Enhancing the Cache Strategy of a Cluster-based DSM System Using an Adaptive Approach

Luciana Arantes, Pierre Sens, and Bertil Folliot
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
{*Luciana.Arantes,Pierre.Sens,Bertil.Folliot*}*@lip6.fr*

Abstract

In a previous article, we presented a per cluster logical cache strategy for a Software Distributed Shared Memory (SDSM) system for interconnects of loosely-coupled machine clusters. We assume that inter-cluster links are slower than intra-cluster ones. In the present article, we propose an adaptive strategy that aims at increasing the efficiency of our first cache solution. By exploiting sharing access pattern of applications related to locality within the same cluster, the adaptive cache reduces the number of messages generated by our previous cache solution. Performance evaluation results show that the former outperforms the latter. We also present a new barrier algorithm, which is integrated with the adaptive cache and more adapted to the communication latency hierarchy of the platform.

1. Introduction

Software Distributed Shared Memory (SDSM) systems provide an abstraction of shared memory for parallel applications running on top of physically distributed memory machines. They combine the programming advantages of the shared memory paradigm with the low cost of distributed systems, protecting the programmer from the difficulties of programming that arise with the message-passing paradigm.

Our research interest is in SDSM systems on interconnects of loosely-coupled machine clusters. As clusters of workstations are widespread architectures, our motivation for proposing a SDSM system for interconnection of clusters relies in the possibility of easily and cost-effectively increasing the number of machines of the distributed execution platform. On the other hand, in a multi-cluster platform inter-cluster links are slower than intra-cluster ones. The difference in speed is usually due to larger communication latency or lower bandwidth. This high cost of inter-cluster network communication affects the applications' performance.

Our proposal for a cluster-based SDSM system is based on the **Lazy Release Consistency (LRC)** memory model, originally defined in TreadMarks [2] SDSM from Rice University. We have modified TreadMarks in order to adapt it to a cluster-based distributed platform. Basically, we have introduced to it a per cluster logical cache and we have adapted the synchronization algorithms to the clustered architecture of the platform. The present article is focused on the per cluster logical cache.

Our SDSM solution provides a transparent logical per cluster cache, which aims at increasing the number of shared data requests that can be satisfied close to the process that is requesting a remote updated copy of a shared data. For instance, if the application has a single producer/multiple consumers sharing pattern, with the producer and consumers belonging to different clusters, whenever one of the consumers gets a copy of a shared data for the first time, the copy is also maintained in the logical cache of its cluster. Therefore, all the other consumers of this cluster, by accessing this logical cache, will have a faster access to the copy of the shared data. In this paper, an *intra-cluster locality page* is a page, whose copy is requested for more than one process within the same cluster.

The logical cluster-level is composed of two caches: the *inherent cache* and the *extended cache*. The first one is implemented by changing the way that the LRC protocol requests a list of missing shared updates, while the second one involves an extra process, dedicated to storing remote shared data updates. The *extended cache* is necessay since it may happen that a processor already has an updated copy of a shared data which a second processor of the same cluster is about to request to a remote processor, but the LRC protocol cannot provide this information. However, the architecture of the platform is a loosely-coupled one and intra-cluster latency can not be ignored, when compared to inter-cluster latency. A processor can not wait for a cache miss reply from the *extended cache* before requesting the data from a remote processor. Therefore, the process requests the data from both the remote process and its local *extended cache* process, waiting for the first reply.

Our *extended cache* strategy undoubtedly generates a non-negligible number of extra messages (see section 5.1). Moreover, many of these messages are useless if the requested page does not present intra-cluster locality, which might lead to network message contention and application performance degradation. To minimize the number of useless messages, we propose in this article an adaptive cache, whose strategy is based on the sharing pattern of the application. The adaptive strategy is integrated with the barrier synchronization operation. Hence, we also present in this article a new barrier algorithm that not only includes in its messages information about intra-cluster locality of pages, but also attempts to tolerate the higher costs of inter-cluster latency.

The remainder of this article is organized as follows. Section 2 gives an overview of TreadMarks DSM and the Lazy Release Consistency protocol. In section 3, we describe the extensions we have added to TreadMarks to integrate a per cluster cache. The adaptive cache and the new barrier algorithm are presented in section 4. Section 5 discusses our experimental results and in section 6, we compare our approach to other works. Section 7 summarizes our contributions.

2 Lazy Release Consistency

In the Lazy Release Consistency model, ordinary memory accesses are distinguished from synchronization operations. The latter are divided into *acquire* and *release* operations. An acquire operation is used to signal the start of a set of accesses to shared data, while a release operation signals its end. A consistent view of the shared memory abstraction is provided to one, or all processes, only at the acquire operation.

LRC-based systems usually offer two types of synchronization variables: locks and barriers. Both can be mapped onto acquire/release operations. A lock operation corresponds to an acquire, an unlock to a release, while an arrival at a barrier can be treated as a release and the departure from it as an acquire operation. The unit of sharing is a page.

To implement LRC protocol, the execution of each process is divided into segments of time called *intervals*. A new *interval* begins at each *acquire* or *release* operation. An interval contains the set of shared memory accesses that a process has performed between two synchronization operations.

The partial order of intervals is controlled by assigning a vector timestamp to each acquire or release operation. Each process P_j keeps a local logical vector clock v_j of N entries, where N is the total number of the system's processes.

At a *remote* acquire, P_j sends its current logical clock value to the release process P_r. P_r sends back all the intervals it itself is aware of, but not P_j. It also includes in the message a list with the identification (*write-notice*) of the pages that have been modified in each of these *intervals*. The updates themselves are stored in structures called *diffs*. A *diff* is created at the end of an interval by comparing the current version of the page with the copy of it (called *twin*) that was saved at the beginning of the interval.

Upon receiving a *write-notice*, the acquiring processor P_j invalidates its local copy of the corresponding page (write-invalidate protocol). The first access to an invalidated page will cause a page fault. The faulting processor P_j will then ask for the updates (*diffs*), related to this page, which it does not have. P_j knows which processors keep these *diffs* since it received this information at acquire operations. It then groups the identification of causally related missing *diffs* into a single list. For each of these lists of *diff* identifications, the faulting processor requests the corresponding *diffs* themselves from the last processor that modified the page and that is included in the list. When receiving the *diffs*, they are individually applied by the faulting processor to its local copy of the page, respecting their causal relation.

3 The Intra-Cluster Cache

By extending the LRC protocol in order to add a per cluster cache, our goal is to increase the number of *diff* requests that can be satisfied in a shorter time. Each logical cluster cache gathers information about the *diffs* available within the cluster. Our system provides one logical *diff* cache per cluster, which is composed of two others: the **inherent cache** and the **extended cache**.

3.1 The Inherent Cache

In the LRC protocol, when a process receives a *diff*, it does not discard it, but keeps in its memory, even after applying it to its local copy of the page. By storing the *diffs* in the processors' local memory, the LRC protocol already provides a kind of cluster cache. Processors within a cluster can benefit from this cache, if the SDSM layer changes the way LRC protocol asks for the missing *diffs*. This is the reason why we have called it the *inherent cache*.

Upon getting a page fault, a process needs to obtain the lists of causally related *diffs* and apply them to the corresponding page, as explained in section 2. In the original LRC protocol, the faulting processor always requests a list of causally related missing *diffs* from the last process that modified the page and whose identification is included in the list in question. In our level-cache solution, whenever this last process, p_r, belongs to a cluster other than the faulting process q, the latter splits the list in two. It identifies which is the last processor p_l of its local cluster to have modified the page and whose corresponding identification is included in the list. Thus, processor q asks this local processor p_l for the *diffs* that it holds, and asks the distant cluster processor p_r for the others, i.e. the *diffs* not available within its cluster. With this approach, the amount of *diffs* sent across the inter-cluster link is reduced when the application presents migratory data, i.e., when the same shared data are sequentially updated by a set of processors. More details about the inherent cache are given in [3].

3.2 The Extended Cache

The inherent cache mechanism is based on the fact that a process can identify, at a page fault, which missing *diffs* can be found within its cluster based on the history of causality information provided by LRC at *acquire operations*. On the other hand, it may happen that a processor does not know that a second process of its local cluster has already requested the same remote *diffs* that it is going to ask for. Examples of such a scenario are applications that have a *"one producer and several consumers"* sharing pattern with the producer and the consumers belong to different clusters, or those that present falsely-shared pages (the false sharing problem arises when two logically unrelated variables accessed by two different processors are placed in the same page). It may happen then that faulting processors of the same cluster will redirect the same *diff* requests to a remote processor. However, if there is a local process in the cluster of those processors, dedicated to storing any *diff* received from outside clusters, the two latter could also ask this process if it could provide the *diffs* they need. If both this local process and a remote process hold the latest *diffs* of the page, a request for these *diffs* will likely be satisfied earlier from the former than from the latter. This is the idea behind the *extended cache*. An extra process that does not take part in the application execution, is added to each cluster with the aim of storing remote *diffs*, i.e. those that a processes are not able to find in the *inherent cache*.

The *extended cache* mechanism works as follows. Whenever a processor issues a request for a list of causally related *diffs* to a processor outside its cluster, it also sends a copy of this request message to its local extended cache process. Upon receiving the request, the remote processor sends back to the requester a response message containing the corresponding *diffs*. However, if it is the first time that the remote processor receives a request for such *diffs*, it also sends a copy of them to the *extended cache* process of the remote faulting processor's cluster. Thus, any other request for some or all of these *diffs* made by a second processor of this cluster can be satisfied by the extended cache process in question. It is worth remembering that the fact that intra-cluster latencies are not inexpensive (although smaller

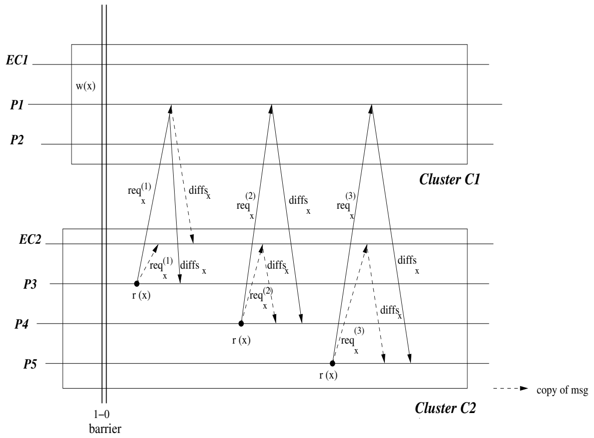

Figure 1. The extended cache

than inter-cluster ones) justifies our design choices for the *extended cache* strategy.

Figure 1 depicts the extended cache mechanism when a "one producer and several consumers " scenario takes place. It shows a two-cluster platform with five processors P_i and their respective *extended cache* process (EC_i). Processor P_1 (the producer) writes on variable x, which will be later read by P_3, P_4 and P_5 (the consumers). Notice that when P_4 or P_5 tries to read the variable x, EC_2 has already received the copy of the *diffs* related to x, and it can, therefore, provide them to the requesting process. EC_2 may then satisfy the request for *diffs*$_x$ earlier than the remote processor P_1, as it happens in the figure.

An *extended cache* process may receive several requests for the same *diffs* before the arrival of the *diffs* themselves, specially considering that inter-cluster links are slower than intra-cluster ones. In order to minimize the effect of this lack of synchronization, we have added a mechanism to the extended cache that stores pending *diff* requests. We called it the *pool of outstanding diff requests*. Whenever an extended cache process receives a request for a *diff* that it does not have, it keeps the request in this pool. Then, when afterwards it receives the *diff* itself from a remote processor, it checks the pool, looking for all the outstanding requests that can be satisfied. For a detailed description of the *extended cache* and the mechanism of the *pool of outstanding diff requests* see [3].

4 An Adaptive Intra-cluster Cache Approach

The *inherent cache* splits a request containing a list of *diffs* only in the case of migratory data. Consequently, when the *inherent cache* is activated, the number of *diff* requests and *diff* answers increases only for applications whose shared data accesses present such behavior. On the other hand, the *extended cache* approach generates a considerable number of extra requests or *diff* messages. Furthermore, many of them may be useless. If the page related to the requested *diffs* does not present cluster locality of access or is not a falsely-shared one, the extended cache will never be able to satisfy the request. Therefore, to minimize this number of messages, we propose an adaptive *extended cache* approach, which works as follows.

When the application starts, every process supposes that the *extended-cache* process of its cluster can service any request for remote *diffs* of any page. Hence, whenever a process issues a remote *diff* request, it also sends a copy of the request to the *extended cache* process of its cluster, as described in section 3.2. Nevertheless, during program execution, a process p monitors, for each page x, from which process it receives remote *diffs* related to x, i.e. for each page, p keeps a counter to control the number of successive remote *diff* responses received from a remote processor and not from the *extended cache* process of its cluster. If p gets a *diff* from the latter, it resets the counter of page x. However, if this counter reaches a pre-defined threshold, process p considers that it is not worth insisting on requesting *diffs* of page x to its *extended cache* process. It assumes that the page is not accessed by any of the other processes of its own cluster. Thereby, the process drops the identification of page x from the set of pages whose *diffs* it considers are worth requesting from its local *extended cache* (*set of intra-cluster locality pages*), and then it stops sending *diff* requests for this page to its *extended cache*. In addition, in the following *diff* request messages related to page x that are sent to a remote process, p includes the information that a copy of the respective *diff* response should not be sent anymore to p's *extended cache* process.

Process p can re-include page x in its *set of intra-cluster locality pages*) if it receives the information, at a barrier operation, that the page has recently begun to present a cluster locality of sharing access pattern within its cluster. It gets this information from its *extended cache*, which takes part in a barrier operation, as explained in the next section.

A Cluster-based Barrier Algorithm

In the original TreadMarks, a barrier has a centralized implementation. One of the processes is elected as the barrier manager. When arriving at a barrier, processes that are not the barrier manager send to the barrier manager all the consistency information that the latter is not aware of. Those processes then block themselves waiting for a reply from the manager. The barrier manager process, in its turn, gathers all the consistency information messages (that basically contain *intervals* and *write-notices*) that it receives from those processes. Once the barrier manager receives such messages from all processes, it sends back a second one to each of them with consistency information of which they were not aware.

By decentralizing the barrier algorithm, our aim is not only to alleviate the work of the barrier manager process and tolerate to some extent inter-cluster latency, but specially to make *extended cache* processes participate in the barrier execution, propagating information about remote accesses.

The algorithm we propose is the following. Arriving at a barrier, every processor sends to every *extended cache* process, including the one of its cluster, the same consistency information message that it would send to the barrier manager in the centralized algorithm. It then blocks itself. Every *extended cache* process, in its turn, plays the role of the barrier manager, but only to the processors of its cluster. After gathering the messages received from all processes, each *extended cache* process releases the processes of its own cluster, by sending them a message containing consistency information of which they were not aware. Figure 2 illustrates our barrier approach.

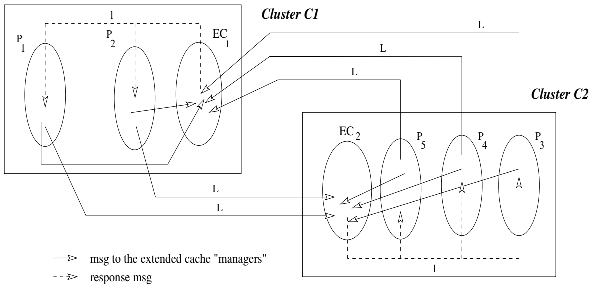

Figure 2. Cluster-based barrier algorithm

If we consider that the average time to send a message between two processors of different clusters is T and between two intra-cluster ones is t, in a centralized algorithm, a remote processor takes $2 * T + \Delta t'$ (where $\Delta t'$ is the barrier's processing time at the processor side plus the processing time at the manager side) to execute a barrier, if all processors start the execution of the barrier at the same time. On the other hand, using our cluster-based algorithm, all processes wait $T + t + \Delta t'$, including the ones of a different cluster than the manager's.

When the adaptive cache approach is activated, barrier messages contain not only consistency information, but also information identifying the pages that recently presented *intra-cluster locality* and the *diffs* that *extended cache* processes already keep. Between two synchronization operations (a *barrier-interval*), each process p controls all the *diff* requests that it receives from the processors of a remote cluster C_r. If during a *barrier-interval*, p receives more than one request for the same *diff* of a particular page, this means that this page presents *intra-cluster locality*. Hence, at the next barrier operation that concludes the current *barrier-*

interval, p includes in its barrier message to the extended cache process EC_r of C_r the information that this page exhibits such behavior. The extended cache EC_r, in its turn, will gather all the information received from remote processes about pages that presented such a sharing pattern and will propagate it to the processes of its cluster by including it in the barrier message response to those processes. It also includes in the message information about the *diffs* that it recently received.

5 Experimental Results

We have performed some tests on a collection of six applications in order to evaluate our cache strategy. All the applications present *cluster locality of sharing accesses*, as we described in our earlier work [3]. The applications are fine-grain or medium-grain applications. In general, this kind of application does not yield such a good performance on a non-uniform latency architecture. For several of these applications, the best solution would be to restructure them, adapting them to a non-uniform latency platform. However, since authors often use such applications to show the efficiency of their DSM systems, they are adequate for showing that several applications exhibit cluster locality of shared memory accesses, and that the cache we propose automatically and transparently exploits this locality. The input problem size of each application is shown in Table 1.

Applications	size	iterations
Barnes-Hut	16K	3
3D-FFT	64x32x32	30
Gauss	2048	-
MG	64x64x64	-
TSP	19 cities	-
Water	343	10

Table 1. Input size of applications

- **Barnes-Hut** from the SPLASH [10] benchmarks performs a N-body simulation under the influence of gravitational forces over a number of time-steps. The update-force calculation phases are inherently one producer/several consumers sharing patterns. The application also exhibits high write-write false sharing.

- **3D-FFT** from the NAS benchmark executes a three-dimensional Fast Fourier transformation. It is an intensive-bandwidth application that exhibits a one-producer/one-consumer sharing pattern [4].

- **Gauss** application solves the Gauss Elimination problem on linear equations with partial pivoting. The sharing pattern for the pivot row is one-producer/several-consumers. For the problem size we have chosen, we observe that one of the pages is strongly requested by consumers.

- **MultiGrid (MG)** from the NAS benchmark [4] performs an iterative multi-grid method to compute the solution of the three-dimensional Poisson problem. MG presents a high write-write false sharing.

- **TSP** uses a branch-and-bound algorithm to solve the traveling salesman problem. A centralized queue keeps partial tours from which processes get jobs. TSP exhibits migratory data as well as a producer/consumers behavior.

- **Water** from the SPLASH benchmark simulates a system of water molecules in an imaginary box [10]. Sharing patterns are both migratory and “one producer/several consumers”.

5.1 Simulation Environment

We have performed our tests on top of a single switched 100Mbs/s Ethernet network of Celeron- and K6-based PC computers.

We divided the machines logically into two clusters, each one having six machines, composed of five Celeron machines that execute the application code, plus one K6 machine. Since the latter is slower than the former, it is dedicated to execute the *extended cache* process. The Celeron-machines are called *computation-processors* or *computation-processes*.

To emulate higher inter-cluster latencies some extra Celeron-based or K6-based machines, called routing machines, were added to the platform. If a processor wants to send a message to a processor of the second cluster when a particular latency is simulated, it sends the message to one of the routing machines. Each routing machine, in its turn, forwards it to a second routing machine or to the target processor of the remote cluster, depending on which latency is being simulated. Hence, by varying the number of extra machines, we have managed to simulate 2 different latency values (*lat1* and *lat2* in ascending order).

The impact of the aforementioned two simulated inter-cluster latency values on the performance of applications was evaluated by measuring the average round-time to get a remote *diff* request. In a platform with no latency simulation, the average time to obtain a remote *diff* varies from 386 to 3,594 microseconds, depending on the application. For each application, we have compared the time to get a remote *diff* with and without any simulation of inter-cluster latency. When latency *lat1* is simulated, a remote diff request is, on average, slowed down of a factor of 2 and when latency *lat2* is simulated, the slow down factor for a remote diff request is on average of 10. Notice that with such simulated latencies, the cost of an intra-cluster *diff* message can not be ignored when compared to inter-cluster ones, which is in accordance with the assumptions we have made for our design choices.

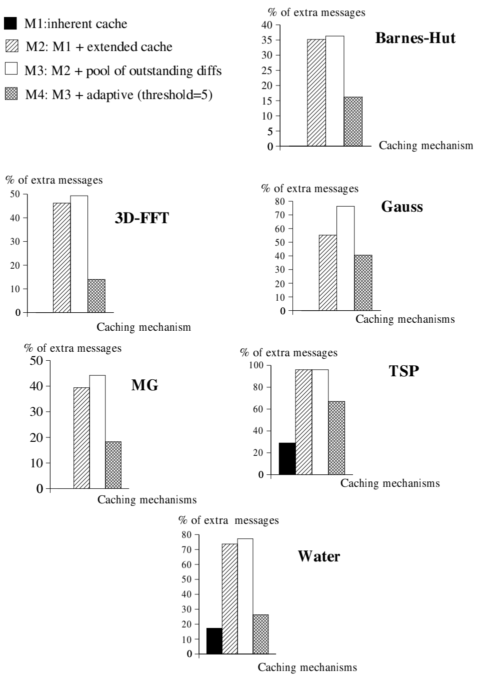

Figure 3. Percentage of extra diff messages

To evaluate the increase of network traffic and show the advantage of adopting an adaptive approach for the *extended cache*, we measured the percentage of extra *diff* messages that our per cluster cache generates on top of the two logical-cluster platform. This network traffic study also goes to prove that the number of extra messages depends on the behavior and sharing pattern of each application. We considered a non-uniform platform with inter-cluster simulation latency equal to *lat2*. Figure 3 shows the percentage of extra *diff* messages when the different mechanisms are activated. The following configurations (M_i) are depicted in the figure:

M_1: only the *inherent cache* is activated;

M_2: the *inherent* and *extended* cache are activated, but not the mechanism of the *pool of outstanding diff requests*;

M_3: M_2 configuration plus the *pool of outstanding diff requests*;

M_4 : M_3 configuration plus the adaptive mechanism with page threshold of 5.

The figure confirms that the *inherent cache* (configuration M_1) has no effect in Barnes-Hut, 3D-FFT, MG and Gauss, since these applications do not present migratory data. On the other hand, for applications like TSP and Water which exhibit migratory data, the number of extra *diff* messages increases. For all applications, the number of extra *diff* messages rises considerably when the *extended cache* mechanism is activated (configuration M_2). The *pool of outstanding diff requests* is responsible for generating a small percentage of extra *diff* messages on the network (configuration M_3 compared to M_2) for the majority of applications. However, for applications like TSP, this mechanism does not induce any extra messages. The reason for this null value is due to the lock-protected shared data behavior of the application. When an extended cache *diff request hit* occurs, it is because another process of the same cluster had read the same lock-protected *diff* when it got the lock in a previous interval. On the other hand, applications that suffer from high write-write false sharing, such as MG, or those that exhibit concurrent consumers for a single producer, such as Gauss, take strong advantage of the *outstanding diff request pool* mechanism.

As we can observe, the number of extra *diff* messages suffers a significant reduction when the adaptive mechanism is adopted (configuration M_4 compared to M_3). We have activated the adaptive mechanism with a threshold of 5 (maximum number of consecutive *diff* requests that a process issues to its *extended cache* process). For the majority of applications, the percentage of extra messages goes down to 15%-20%, representing a substantial decrease.

In order to evaluate the effect of clustering and the cluster locality of shared data accesses of each application, we measured the average percentage of *diff request hits* of an *extended cache* (EC) process. It corresponds to the percentage of remote *diff* requests that each *extended cache* process has been able to service either because it already caches the *diff* (*prompt diff request hit factor*) or because of the *pool of outstanding diff* mechanism (*delayed diff request hit factor*).

Table 2 summarizes the average percentage of extended cache *diff request hits* when applications are executed on top of the platform when the highest inter-cluster latency (*lat2*) is simulated. We measured the *diff request hit factor* with no adaptive mechanism at all, as well as with the adaptive mechanism, separately considering a *page threshold* of 5 and 10 successive *diff* requests.

For many of the applications, the adaptive approach results in an increase in the *diff request hit* factor, when the page threshold is equal to 5. The explanation for such an increase is the reduction of useless *diff* requests and of network traffic as a whole. However, the adaptive approach does not have a significant impact on Gauss. The reason for this lies in both the functioning of the adaptive cache

Appl.	no adap	thresh=5	thresh=10
Barnes-Hut	26.5%	29.1%	27.2%
3D-FFT	32.5%	41.7%	32.8%
Gauss	77.3%	76.3%	76.9%
MG	21.9%	33.1%	31.9%
TSP	44.1%	59.2%	61.4%
Water	44.7%	46.3%	45.7%

Table 2. EC diff request hit percentage

mechanism and the behavior of the application itself. In Gauss, for the problem size we have chosen, only one page, the pivot, presents a "one producer/ several consumers behavior". Hence, if a faulting processor stops requesting this page, in its following request it warns the remote process to stop sending a copy of the corresponding *diffs* to the extended cache process in question, as explained in section 4. However, an *extended cache* process, at every barrier operation, informs the computation-processes of its cluster about the set of pages that recently presented a cluster locality of sharing accesses, which in the case of Gauss is the pivot page. Thus, the computation-processes will re-include this page in the set of pages whose *diff* it will also request from the local extended cache.

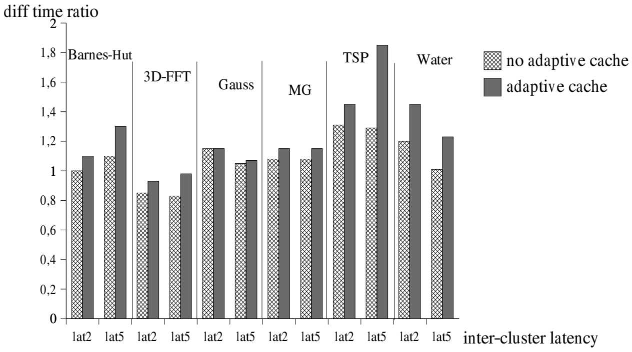

Figure 4. Remote diff time ratio

We also evaluated the average time that a processor is stalled waiting for an answer to a remote *diff* request, when the adaptive mechanism is activated and not. Figure 4 shows, for each application the "remote *diff* time speedup", i.e. the ratio *average time that the application is stalled waiting for a remote diff with no cache mechanism at all/ the same time with our intra-cluster cache mechanism*, both when the adaptive mechanism is activated and when it is not. We consider a page threshold of 5. If the cache mechanisms improve performance of the application, this ratio should be greater than 1.

When the adaptive mechanisme is not activated, we can observe that the time that a process waits for a remote *diff* request was reduced for most applications, except 3D-FFT. On the other hand, for the other applications, except Gauss for the reasons previously explained, the adaptive approach reduces even more the time that a faulting process waits for a remote *diff*, which confirms the efficiency of our adaptive approach. In the case of 3D-FFT, which is an intensive-bandwidth application, even if the adaptive mechanisme reduces the number of extra messages, the latter remains still high, being responsible for degrading the application.

Figure 5 shows the gain in performance of the applications when the adaptive cache is adopted. It gives the ratio *application execution time when the cache is not activated / application execution time when the adaptive cache is activated, with a page threshold of 5.*

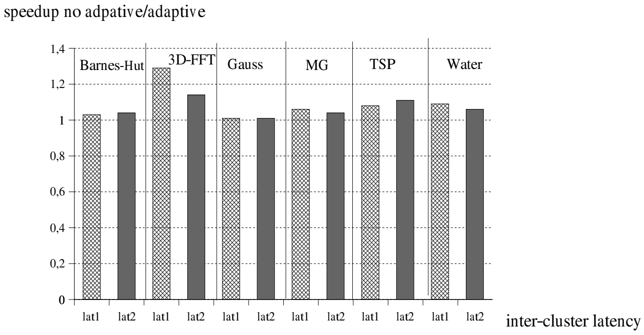

Figure 5. Application speedup

As we can observe in the figure, our adaptive approach entails a gain in performance for all applications, except Gauss. This gain is a consequence of the reduction of the time that a process is blocked, waiting for a remote *diff*, as shown in figure 4. Bandwidth-intensive applications, like 3D-FFT, strongly benefit from the adaptive cache since it entails a large reduction on the number of messages over the network, lessening the problem of network contention. However, even with this gain in performance, our adaptive cache still degrades the performance of 3D-FFT when compared to the execution of the application on top of the original LRC protocol, which shows that our cache strategy is not adequate for this kind of application.

6 Related Work

Some systems, such as Cashmere-2L [11], Shasta [9] have extended the DSM approach across a network of symmetric multiprocessors (SMP). In these systems, data sharing within a SMP node is supported by the intra-node physical memory, and the software DSM protocol is invoked for data that is shared across the nodes. This feature of exploiting the intra-node physical shared memory is not possible in our loosely-coupled cluster-based solution.

The proposal of some systems such as InterWeave [6], PerDis [7] is to provide shared memory support for distributed wide-area (WAN) environments, as for instance Internet. In order to minimize WAN high communication costs, the majority of them support more relaxed shared memory consistency than LRC memory model and more flexible synchronization semantics than the traditional ones offered by DSM systems. A global coherent view of shared data is often too costly for such platforms. Consequently, large-scale distributed shared memory systems tolerate temporal out-of-date copies of shared data (e.g., read-only objects with one or two versions old), if this inconsistency implies faster response time. As on our approach, the main bottleneck of these systems is latency between machines of different clusters.

Adaptive LRC-based DSM systems, proposed by some authors [1, 8], automatically adapt themselves at run-time to the memory access patterns of applications. For instance, based on the application behavior, the LRC protocol can toggle between between invalidate and update protocols. As on our approach, the change of the shared memory support behavior is based on the memory access patterns of the applications and the propagation of information that allows this adaptation takes place at synchronization operations.

In terms of barrier algorithms, the *hierarchical barrier* one, adopted by some SMP-based DSM systems [5] also alleviates the centralized barrier manager, as in our algorithm. In addition, it reduces the amount of data and the number of messages sent over inter-node links. In this algorithm, a processor within the SMP node collects the consistency information from all the processors of its node before sending them to the centralized barrier manager. However, in our platform, intra-cluster latency is not insignificant and such an algorithm would increase even more the time of sending a synchronization message between processors of different clusters.

7 Conclusions

The tests performed with our prototype confirm that DSM applications exhibit cluster locality of sharing accesses and that our per cluster cache mechanisms make applications exploit this locality, minimizing, thereby, the time of inter-cluster requests for shared data. For several applications of our test suite, the time that a process was stalled waiting for an updated copy of a remote shared data was reduced. The adaptive cache strategy enhances the efficiency of the *extended cache*, reducing the number of useless *diff* messages. Therefore, the percentage of *diff* requests satisfied by the *extended cache* increases. The time that a process waits for a remote updated copy of a remote shared data was reduced even more for the majority of the applications with the adaptive approach. The cluster-based barrier algorithm also proved to be adequate for our solution. It is integrated with the adaptive cache approach. Its clustered structure execution allows the propagation of information about *diffs* recently received by the extended cache, as well as information about cluster locality of shared data accesses.

References

[1] C. Amza, A. Cox, K. Dwarkadas, Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *IEEE Proceedings*, 87(3):467–475, Mar. 1999.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of worksations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] L. Arantes, P. Sens, and B. Folliot. The impact of caching in a loosely-coupled clustered software DSM system. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 27–34, Chemintz (Germany), Nov. 2000.

[4] D. Bailey, J. Barton, T. Lansinski, and H. Simon. The NAS parallel benchmark. Technical Report 103863, NASA, July 1993.

[5] A. Bilas, L. Iftode, and J. P. Singhal. Shared virtual memory across SMP nodes using automatic update: protocols and performance. In *Proceedings of the 6th Workshop on Scalable Shared Memory Muliprocessors*, Oct. 1996.

[6] D. Chen, S. Dwarkadas, S. Parthansarathy, E. Pinheiro, and M. L. Scott. InterWeave: A middleware system for distributed shared state. In *Proceedings of the 5th Workshop on Languages, Compilers and Run-time Systems or Scalable Computers*, May 2000.

[7] P. Ferreira, M. Shapiro, and et. al. PerDiS: design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, Feb. 2000.

[8] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of the Fourth International Symposium on High Performance Computer Architecture*, Feb. 1998.

[9] D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, Feb. 1998.

[10] P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[11] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis, Parhasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 170–183, Oct. 1997.