The Impact of Caching in a Loosely-coupled Clustered Software DSM System

Luciana Arantes, Pierre Sens and Bertil Folliot

LIP6 Laboratory
University of PARIS VI, Paris, France
{[Luciana.Arantes, Pierre.Sens, Bertil.Folliot]@lip6.fr

Abstract

As interconnection of local-area network of workstation are widely available, the idea of offering a software distributed shared memory (SDSM) layer across them is quite an attractive alternative for compute-intensive applications. However, the higher cost of sending a message over an inter-cluster link than over an intra-cluster can limit applications' performance on a multi-cluster SDSM. In this paper, we present the extensions that we have added to TreadMarks SDSM in order to adapt it to a loosely-coupled cluster-based platform. A logical per-cluster cache was added to the protocol in order to exploit cluster locality. By accessing its local cache, a processor can share data previously requested by another processor of its cluster, hiding, in this way, the cost of inter-cluster communication.

1 Introduction

Software distributed shared memory (SDSM) systems offer an abstraction of shared memory for parallel applications running on top of physically distributed machines. It is a very interesting approach for compute-intensive problems since it combines the programming advantages of shared memory paradigm with the low cost of distributed systems.

The majority of SDSM systems found in literature have been implemented for networks of uniprocessor machines or, more recently, for networks of shared-memory multiprocessors machines (SMPs). Our proposal is to extend the SDSM concept to an interconnection of loosely-coupled clustered machines, as for instance, two or more interconnected LANs. Contrary to SDSM for interconnections of SMP nodes, our approach does not consider any hardware shared-memory support within a cluster. As networks of workstations (NOWs) are widespread accepted, our motivation for proposing a cluster-based SDSM system is the possibility of easily, and at a low cost, increasing the number of machines of the distributed execution platform, or exploiting idle remote machines.

However, in a multi-cluster platform, inter-cluster links are usually slower than intra-cluster ones. The gap in speed may be due to larger communication latency and/or lower bandwidth. This higher cost of sending a message between two machines of different clusters can not be neglected. It may represent the major bottleneck that limits the performance of applications on a multi-cluster SDSM. This is the reason why coarse-grain applications, whose processes hardly communicate, or those that present substantially higher intra-cluster communication than inter-cluster communication (e.g. cooperatives or embarrassingly parallel ones) are surely the most adequate applications for such platform. On the other hand, several studies [IFT 96] [PLA 99] [JIA97] that characterize the behavior of benchmarks' parallel applications show that they usually present some patterns of shared data accesses. Thus, a SDSM layer for a clustered platform could be customized in order to exploit these sharing pattern, specially those related to cluster locality. For instance, in a single producer with multiple consumers pattern, one processor updates a shared data which is afterwards read by some or all the other processors. Considering a multi-cluster SDSM, if the consumers and the producer belong to different clusters, the updated data could be brought closer to the consumers' cluster. All local processors of this cluster would, then, have a faster access to the shared data.

A cluster-based SDSM can also exploit the fact that most DSM systems divide the virtual shared memory into logical units of sharing and that both replication and coherence between the copies are kept at the coarse granularity of these units. Hence, two logically unrelated variables, accessed by two different processors of the same cluster, may happen to be in the same unit of sharing. This is called the false sharing problem. In the case of a multi-cluster SDSM, if the two mentioned processors are aware that the last updated copy of a remote falsely shared page is also available within the cluster, they can both access the latter.

A third important point to consider for the conception of a multi-cluster SDSM is the choice of the protocol that provides consistency of shared data. We have opted for the *Lazy Release Consistency* (LRC), multiple-writer page-based DSM, originally implemented in TreadMarks [AMZ96, KEL94]. Our decision was taken not only because of TreadMarks's good performance on top of LANs, but also because a processor always keeps in memory all the updates (called *diffs*) made by itself as well as the ones made by other processors and which it is aware of. Therefore, other processors within the same cluster can later make use of these updates.

Based on the three above points, we have added some extensions to TreadMarks's LRC protocol in order to implement a logical per cluster cache. Each cluster cache gathers information about shared data available within the cluster. By accessing its local cluster cache, a processor may make use of shared data previously requested by a second processor of its same cluster. If both a cache and a remote processor hold the last updates made to a page, a request of a local processor for these updates will likely be satisfied firstly by its local cache rather than by the remote processor.

We performed some tests with fine-grain and medium-grain applications from benchmark suites [BAI 93] [SIN 92] that show that several of them present cluster locality sharing patterns. These results do not mean that medium-grain or fine-grain applications present a satisfactory performance on top of our multi-cluster SDSM. Specially because, their performance is strongly influenced by how great the gap between inter-cluster and intra-cluster links is. Moreover, for several of our test applications, the better solution would be to restructure them, as described in [PLA99] [JIA97]. On the other hand, our approach automatically, and transparently, offers a mechanism for exploiting cluster locality, which we believe several applications can benefit from.

The remainder of this paper 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 added to TreadMarks in order to integrate a per cluster cache to it. Section 4 discusses our experimental results. In section 5, related works are presented. The contributions of this work is summarized in section 6, while the last section presents the evolution of it.

2 Lazy Release Consistency

In the Lazy Release Consistency model, originally defined by TreadMarks [AMZ 96] [KEL94], ordinary memory accesses are distinguished from synchronization operations. The latter is 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 to signal its end [KEL94]. A consistency view of the shared memory abstraction is provided to one, or all, processes only at the acquire operation. This postponement reduces much of the communication required to make shared data consistent.

LRC systems usually offer two types of synchronization operations: locks and barriers. Locks are used to control accesses to shared memory critical regions, while barriers provide global synchronization. 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 modeled as a release and the departure from it as an acquire operation.

The unit of sharing is a page and virtual memory hardware is used to control invalidated processors' copies of pages. The execution of each process is divided into segments of time called intervals. LRC controls the partial order of intervals by assigning a vector *timestamp* to each interval. A new interval begins at each acquire or release operation. By ensuring the causal ordering of all processes' intervals, the consistency of the shared memory is assured. A process locally controls which pages it has modified at each interval, keeping this information in data structures called *write-notices*. The updates themselves are stored in structures called *diffs*, which are run-length encoded records. A *diff* is created at the end of an interval by comparing the current version of the page with the copy of it that was saved at the first write access within the interval.

In *Lazy Release Consistency*, the propagation of updates (i*ntervals* and *write-notices*) is delayed to a processor until the latter executes an acquire operation. Upon receiving the answer to an acquiring request, a process invalidates its local copies of pages, according to the *write-notices* received. The first access to an invalidated page will cause a page fault. The faulting processor will then ask for the updates (*diffs*), related to this page, which it does not have. It knows which processors keep these *diffs* since it received this information at acquire operations. So, it groups each set of causally related *diffs* into a single list. The set of all these lists corresponds to the updates that the processes have concurrently made to the page (multiple-writers). For each of these lists, the faulting processor requests the *corresponding* diffs to the last processor that modified the page and which is included in the list ("if a processor *p* has modified a page at a timestamp *t*, then *p* has all the *diffs*, including those from other processors, corresponding to timestamps that causally precede it [AMZ96]"). 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 per cluster cache

As only after receiving all the missing *diffs*, a faulting process can go on running, the time that a process waits for a *diff* request has direct influence on the overall performance of the application. If one or more of these requests were issued to remote cluster processors, this time will be higher than if all the requests were issued to local processors.

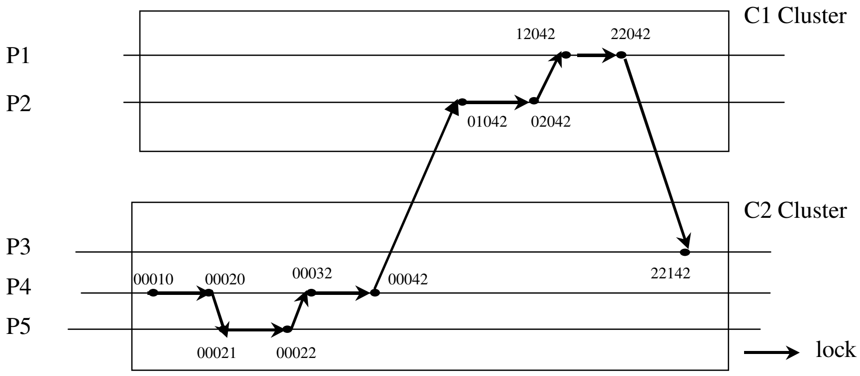

Figure 1: Inherent cache

Thus, 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 closer to the issuing processor, and therefore, more efficiently.

In our cluster-based SDSM, each per cluster cache is composed of two others: the *LRC inherent cache* and the *extended cache*. The first one is implemented by changing the way LRC protocol requests a list of causal related *diffs*, while the second one involves an extra process which is dedicated to store remote *diffs* and requests. Both of them are described below.

3.1 The inherent cache

In 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. It keeps the *diff* because a third process may request it. By storing the *diffs* in the processors' local memory, LRC protocol provides a kind of cluster cache. This is the reason why we have called it the *inherent cache*. Processors within a cluster can benefit from this cache, if the multi-cluster SDSM layer changes the way the protocol asks for the missing *diffs.*

As explained above, when a processor gets a page fault, it needs to obtain the lists of causally related *diffs* and apply them to the corresponding page. In the original LRC, the faulting processor, *q*, always requests each list of causally related *diffs* to the last processor *p* that have modified the page. In the case of our multi-cluster SDSM, whenever *p* belongs to a cluster other than *q*'s (the faulting processor), the latter splits each list of causally related *diffs* in two. for each list, it identifies which is the last processor of its local cluster that is included in the list in question. Processor *q* asks to this processor for the *diffs* that it holds and asks to the distant cluster processor *p* for the others, i.e., the ones not available within its local cluster. Therefore, the list of *diffs* requested to a remote cluster only includes the ones that have not been created by any of the local processors and that *q* supposes that were never previously requested by any of the other local processors. With this approach, the amount of *diffs* sent across the inter-cluster link is reduced. Besides, in the case of long list of causal related *diffs*, i.e. migratory data pattern applications that present the problem of *diff* accumulation [AMZ99], the overlap of the 2 *diff* requests can minimize the time that the faulting processor *q* will wait for the *diffs*.

Figure 1 illustrates the *inherent cache* idea. It shows five processors divided into two clusters. The application uses a lock to protect accesses to shared data. Timestamps assigned to each acquire or release operation are indicated. For instance, if after acquiring the lock (at 22142), P_3 attempts to access an invalidated page, it will get a page fault and will need to request the missing *diffs*. If we consider that all processors that previously acquired the lock modified the page, in the original LRC protocol, P_3 would request to P_1 the whole list of missing *diffs* (causally related diffs corresponding to intervals from 00010 to 22042). However, at this point P_3 knows that P_4 was the last processor of its cluster which updated the page. Hence, P_3 splits the list of *diff* requests in two: for P_4 it asks for the *diffs* until the timestamp 00042 (when P_4 lost the lock), and for P_1 the remaining *diffs*.

3.2 The Extended Cache

The inherent cache mechanism is based on the fact that each processor knows, at a page fault, which are the causal relation *diffs* that are missing. However, it may be the case when a processor is not aware that another process of its local cluster has already requested the same *diffs* that it is going to ask for. For instance, after a barrier operation some or all the processors may access the same local invalidated copy of a page.

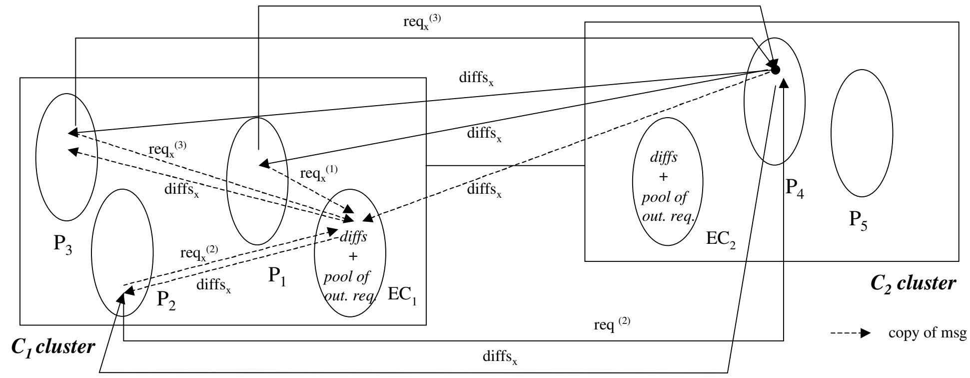

Figure2: The extended cache approach

If it is a remote processor that holds the latest *diffs* of this page, the processors will concurrently redirect the requests to this remote processor. Examples of such scenario are falsely shared pages or pages with one producer and several consumers sharing pattern. On the other hand, if there was a local process, dedicated to store any *diff* received from any outside cluster, the requesting process could check if this second process could not provide the diffs it needs. 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. It can run on any processor and has the objective of keeping remote *diffs*, i.e., those that a process was not able to find in the *inherent cache* at the moment of the request. In other words, processors can benefit from the extended cache whenever they request the same non local *diffs* within concurrent intervals.

Figure 2 shows a two cluster platform with their respective application-processors P_i and *extended cache* processes (EC_i). Consider, for instance, that processor P_4 (the producer) wrote on variable x, which is, afterwards read by P_1, P_2, and P_3 (the consumers). Thus, when a processor of cluster C_1, e.g. P_1, issues a request to P_4 for the list of *diffs* causally related to the updates made on the variable x ($diffs_x$), it also sends a copy of this request to its local *extended cache* process EC_1. Upon receiving the request, the remote processor P_4 sends back to P_1 a message with the corresponding $diffs_x$. Besides, as it is the first time that P_4 receives a request for such *diffs*, it also sends a copy of them to the *extended cache* of the remote cluster (EC_1). So, if we consider that P_1's request was issued before P_2's and P_3's, the former will not be satisfied by EC_1, but the other two requests have a chance of being. If these processors receive $diffs_x$ from EC_1 they will go on running, discarding the further $diffs_x$ message that they will receive from P_4. Moreover, as the *extended cache* is persistent, if there was a fourth processor in cluster C_1, a further request for these same *diffs* from this processor would be satisfied by the *extended cache*. The *extended cache* is only cleared out when Treadmarks's garbage collector is triggered.

Notice that a processor (P_4 in the figure) controls which are the *diffs* that it has already sent to a remote *extended cache*, sending them only once. Besides, it may be the case that, at a page fault, a single processor issues concurrent *diff* requests to different non-local processors. Thus, instead of sending the copies of these requests in separate messages to its *extended cache* process, the SDSM layer combines them into a single message. This aggregation of messages reduces intra-cluster messages as well as possible *extended cache*'s contention. Applications like Barnes-Hut from SPLASH-2 suite [STE92], that intensively issue concurrent *diff* requests, can strongly benefit from it.

The *extended cache* process just saves remote *diffs,* supplying them to local processors. It operates quite asynchronously and messages that are sent to or received from it are unreliable, not requiring acknowledgement. The loss of messages does not affect correctness of program execution, since the *extended cache* operates with copies of messages. The handling of original messages lost is guaranteed by the LRC protocol itself. As the *extended cache* does not take part in application's execution, it does not spend CPU time in coherence actions such as creating twin of pages, generating *diffs* or applying them to pages. These tasks involve substantial software overhead [IFT 96].

The pool of outstanding diff requests

If we consider that inter-cluster links are slower than intra-cluster ones, the *extended cache* process may receive several requests for the same *diffs* before the arrival of the *diffs* themselves. This is the reason why the *extended cache* process stores in a pool every request, whose *diffs* it does not have. Hence, whenever it receives one or more *diffs* from a remote processor, it checks this pool, looking for all the outstanding requests that can be satisfied. For instance in figure 2, if processor P_2 and P_3 had issued their requests request, $req_x^{(2)}$ and $req_x^{(3)}$

respectively, when $diffs_x$ were on the fly, the *extended cache* process EC_1 would save these requests in the *pool of outstanding diff requests.* As soon as EC_1 receives the copy of $diffs_x$, it will swept through the pool looking for all the requests that could be satisfied, in this case $req_x^{(2)}$ and $req_x^{(3)}$, forwarding a copy of $diffs_x$ to the respective processors. EC_1 does not send the *diffs* to process P_1, since it knows that the remote processor, P_4, has directly sent the original $diffs_x$ message to it.

Remark that, if N_i is the number of processors of cluster i and m the number of total clusters of the platform, an EC_i process has to keep only $N_i\ x\ (N_1 + \ldots N_{i-1} + N_{i+1} \ldots + N_m)$ entries in the *pool of outstanding requests*, since, at each page fault, a processor of cluster EC_i can issue at utmost $N_1 + \ldots N_{i-1} + N_{i+1} \ldots + N_m$ concurrently requests for remote missing *diffs*. Besides, when receiving a request for *diffs* of a second page from a processor, the *extended cache* process can discard this processor's previous outstanding requests. It knows that the latter have been satisfied since the processor is asking for *diffs* concerning a new page.

4 Experimental Results

The cache extensions described in the previous sections were integrated in TreadMarks DSM, version 0.10.1. We used the following medium-grained or fine-grained applications: Red-Black SOR and TSP which are distributed with TreadMarks [COX 99]; Water and Barnes-Hut from SPLASH-2 benchmark suite [SIN92]; IS, 3D FFT and Multigrid (MG) from NAS benchmark suite [BAI 93]. The input problem size is shown in Table 1.

Applications	Size/Iterations
TSP	19 cities
SOR	1000x1000, 200
IS	16x16, 10
3D-FFT	256x256x16,30
Barnes-Hut	4K, 3
Water	343,10
MG	64x64x64

Table 1. Input problem size of applications

TSP solves the traveling salesman problem, using a branch-and-bound algorithm. As it protects with locks the access to a centralized job queue, TSP exhibits migratory data. It also presents the problem of false-sharing since some of their shared structures are kept in the same page.

SOR solves partial differential equations, using two matrixes. The matrix are equally divided by the number of processors and statically assigned to them. Shared data communication happens only between neighbor processors when they access boundary rows [COX 99].

Barnes-Hut application performs a N-body simulation using the hierarchical Barnes-Hut method [SIN 92]. It presents for some data and phases multiple-producers/multiple-consumers data referencing patterns. As the read and write accesses are both fine grained, it presents high false sharing [JIA 98].

IS sorts a sequence of keys, using bucket sort. It presents migratory data pattern and the problem of *diff* accumulation [BAI 93] [AMZ 99].

Water simulates a system of water molecules [SIN92]. Sharing patterns are both migratory and producer-consumers. The array of molecules is equally divided among processors. The problem of false sharing exists for boundary pages [COX99].

3DFFT executes a three-dimensional Fast Fourier transformation. It is an intensive-bandwidth application and presents 1 producer- 1 consumer sharing pattern [BAI 93] [JIA 98].

MG [BAI 93] uses a multi-grid method to compute the solution of the three-dimensional scalar Poisson solution. It presents a high write-write false sharing.

4.1 Simulation Platform

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

We logically divided the machines into two distinct clusters. Each cluster has up to six machines, where one of them is dedicated to execute the *extended cache* process. The others are called application-processors.

For simulating a small higher inter-cluster latency, we used an extra machine per cluster. Thus, every message to a remote processor is not directly sent to it. It is routed to one of these machines that, in its turn, forwards it to the target processor.

The average time, in milliseconds, to get a remote *diff* request, a copy of a remote page, and all the missing *diffs* of an invalidated page (handling of a page fault), is shown in Table 2, when an inter-cluster latency is not simulated (uniform latency) and when it is (higher inter-cluster latency). The applications were executed running 5 application-processes per cluster.

The number of *diff* bytes per message sent over inter-cluster links is summarized in Table 3, considering an uniform latency two-cluster platform with 5 application-processors per cluster. The *inherent* cache is the one responsible for the reduction of inter-cluster bytes/message ratio. Applications with migratory data pattern, such as IS, TSP and Water, can benefit from it. This reduction explains why applications with a strong problem of diff accumulation, as the IS, presents a better speedup when the per cluster cache is activated (see Figure 3).

Applic.	Avg. diff request time		Avg. Page request time		Page fault handling	
	Uniform latency	Higher inter-cluster latency	Uniform latency	Higher inter-cluster latency	Uniform latency	Higher inter-cluster latency
Barnes-Hut	**0.90**	**1.21**	**2.77**	**2.96**	**0.79**	**0.95**
3D- FFT	**1.36**	**2.05**	**2.15**	**2.41**	**1.10**	**1.34**
MG	**0.95**	**1.80**	**2.03**	**3.31**	**1.92**	**2.23**
IS	**3.95**	**5.26**	**2.06**	**3.29**	**3.6**	**6.8**
SOR	**0.83**	**1.03**	**2.90**	**3.15**	**0.71**	**0.71**
TSP	**0.37**	**0.63**	**2.37**	**2.89**	**0.39**	**0.52**
Water	**0.87**	**1.76**	**2.20**	**2.45**	**0.67**	**0.92**

Table 2: Average time for remote data requests in msecs.

Applications	No cache	Per cluster cache
Barnes-Hut	465	464
3D- FFT	1800	1800
MG	1286	1286
IS	5817	4147
SOR	721	721
TSP	542	404
Water	485	435

Table 3. Number of inter-cluster diff bytes/msg

In order to evaluate the effect of clustering and applications' cluster locality, we have measured the percentage of remote *diff* request hits of the *extended cache* process. In Table 4, we can observe that increasing the number of applications-processors of the cluster, the *diff* hit factor increases as well. Applications whose processes rarely share the same page, as SOR, does not present any hit factor at all. Therefore, it can not take advantage of the *extended cache*.

Table 5 summarizes, for the 5 application-processors per cluster configuration, the percentage of *diff* hits that the extended cache could satisfied using the *outstanding diff request pool* mechanism (delayed hits) and the percentage of remote *diff* request answers that a faulting processor received from its respective *extended cache* process and not from a remote processor (coverage factor). Both numbers increase for platforms with higher inter-cluster latency, as expected. The *delayed hit factor* proves that, with a higher inter-cluster latency, the probability that the *extend cache* receives concurrent requests for on-the-fly diffs increases. The *coverage factor* represents the effectiveness of the *extended cache*, since it is defined as the percentage of total requests that the *extended cache* was able to reply in time to the faulting processor.

Applic.	Uniform latency			Higher inter-cluster latency		
	3 nodes	4 nodes	5 nodes	3 nodes	4 nodes	5 nodes
Barnes-Hut	**56%**	**63%**	**71%**	**55%**	**62%**	**72%**
3D- FFT	**15%**	**18%**	**39%**	**15%**	**18%**	**39%**
MG	**9%**	**18%**	**20%**	**10%**	**17%**	**21%**
IS	**23%**	**37%**	**42%**	**24%**	**35%**	**38%**
SOR	**0**	**0**	**0**	**0**	**0**	**0**
TSP	**31%**	**38%**	**46%**	**32%**	**39%**	**44%**
Water	**35%**	**44%**	**53%**	**34%**	**42%**	**52%**

Table 4. The Effect of clustering

Applications	Uniform latency		Higher inter-cluster latency	
	Delayed hit factor	Coverage factor	Delayed hit factor	Coverage factor
Barnes-Hut	**3%**	**42%**	**6%**	**44%**
3D- FFT	**16%**	**18%**	**25%**	**29%**
MG	**3%**	**12%**	**8%**	**15%**
IS	**12%**	**10%**	**15%**	**15%**
SOR	**0**	**0**	**0**	**0**
TSP	**2%**	**20%**	**1%**	**44%**
Water	**3%**	**20%**	**20%**	**42%**

Table 5: Extended cache effectiveness

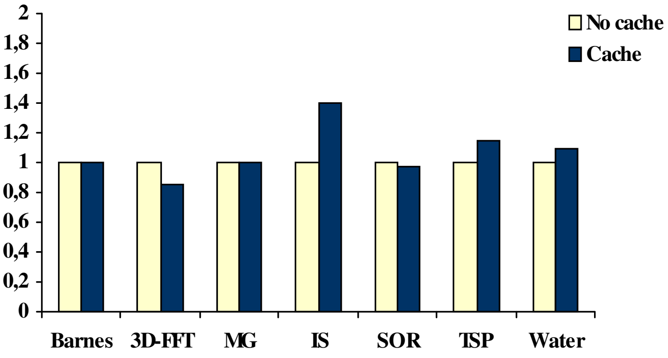

Figure 3: Speedup of applications on top of the non-uniform latency platform

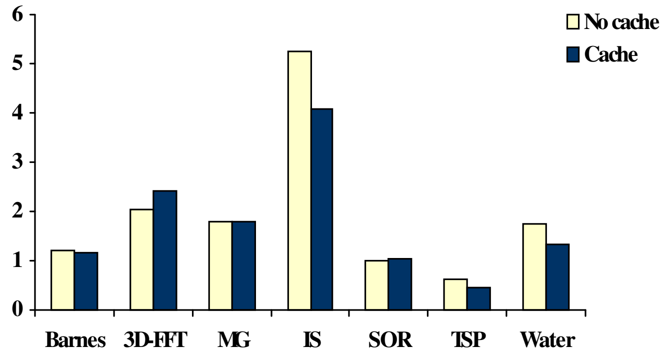

Figure 4: Average remote diff request waiting time on top of non-uniform latency platform (msec.)

Considering the non-uniform platform with 5 application-processors per cluster, the speedup of the applications and the average time that applications were blocked waiting for a remote *diff* are respectively shown in Figure 3 and Figure 4. The latter justifies the effectiveness of our cache approach, since the just mentioned waiting time was reduce for some applications. As explained above, the significant speedup of IS application is due to the *inherent* cache that considerably reduces inter-cluster *diff* message size. The small gain for TSP and Water is a consequence of both the *inherent* cache and the *extended* one, since the applications presents migratory shared data and multiple writers patterns, which are respectively confirmed by Table 3 and 4.

However, we can argue that the per cluster cache did not yield a substantial performance improvement. This lack of speedup happens either because the number of *diffs* satisfied by the cluster cache was small or because the extended cache did not send them significantly earlier to the requesting application-processor. We believe that with higher inter-cluster latency, the latter tends to happen more frequently. Besides, the use of a prefetching mechanism, as discussed in the last section, can minimize these two problems, making the cache more effective. Another point that can not be neglected is the considerable increase of network traffic due to the sending of diff message copies to the *extended cache* process. For applications as 3D-FFT, which is an intensive-bandwidth application, the extended cache entails network contention delays that influence the application speedup. Similarly, in spite of the fact Barnes-Hut application presented a high remote *diff* hit (Table 4) due to its multiple writers/multiple readers sharing pattern, and a *coverage factor* (Table 5), its intensive concurrent fine-grained accesses cause contention in the *extended cache* process. Aggregation of messages reduces this problem, but for larger problem size than the one used in our tests, lost of messages and data retransmission was observed. Selective requesting of *diffs* to the extended cache, as described in the last section, will strongly reduce this problem.

5 Related Works

Home-based Lazy Release Consistency (HLRC) is a variation of LRC [ZHO 96]. Every shared paged is statically assigned to a node. At a release operation, *diffs* of each modified page are eagerly sent to its respective home node, where they are applied to the local copy of the page. The *diffs* are discarded as soon as they are dispatched to the home node or after been applied at the home's copy. At a page fault, a processor requests the updated version of the whole page from its home node. The similarity to our multi-cluster SDSM, is that in both approaches, *diffs* are sent to an assigned processor. However, the *extended cache* process does not take part in application's execution. It just keeps copies of remote cluster *diffs*, whose originals are kept in the application processor's memory.

In order to provide larger scalable shared virtual memory systems, several works [SAM 98] [SCA 98] [STE97] propose to extend the DSM approach across networks of SMP nodes. Their design choices take advantage of cache coherence hardware to maintain consistency within each node. For instance, in the page-based Cashmere-2L system, pages are kept in the intra-node shared memory, and any update made to them is immediately seen by all intra-cluster processors. Shasta [SCA 98] adopts a fine-grain approach and the underlying hardware is also exploited to keep coherence within a node. In HLRC-SMP [SAM 98], based on HLRC protocol described above, any consistency information received from a remote SMP node is kept in the hardware shared memory, being, then, available to any local processor. This advantage of exploiting intra-node physical memory hardware, invoking the software DSM protocol only for data shared across the SMP nodes, is not possible in our loosely-coupled cluster based-solution. On the other hand, comparison with our solution can be done, since the aim of our logical cluster cache is also to provide availability of all consistency information received from an outside cluster.

Plaat et al. [PLA 99] make a performance study of medium-grain and fine-grain parallel applications running on top of cluster-based systems, where inter-cluster links are slower than intra-cluster ones. Their experimental testbed allows the emulation of a variety latency and bandwidth gap. By restructuring them, the applications were adapted to the non-uniform remote memory access system, presenting therefore, a better performance. Application restructuring is also proposed in [JIA97] in order to migrate applications that perform well on top of physical shared address space multiprocessor towards page-grained distributed shared virtual memory systems. Contrary to our solution, in both cases, the optimizations were introduced to the applications themselves and not to the shared memory support.

6 Conclusions

This paper proposes a per cluster cache for loosely-coupled cluster-based SDSM systems, based on TreadMarks SDSM. Taking advantage of LRC consistency information persistency and the fact that processors can concurrently request the same unit of sharing, we have added some extensions to TreadMarks in order to implement a logical per cluster cache.

The results present in section 4 confirm the effect of clustering: migratory access pattern applications can benefit from the *inherent cache*, while application with producer/consumers or multiple-writer access pattern can benefit from the *extended cache*. The greatest benefit provided by our approach is the reduction of time that a process waits for *diffs* at a page fault. This reduction may result in performance gain. Since our cache proposal focuses on hiding the cost of inter-cluster latency and exploiting cluster locality, our solution is more effective on higher inter-cluster latency platform.

7 Future works

The effectiveness of the *extended cache* is directly related to the time that it takes to satisfy a request for remote *diffs*. If the *extended cache* does not have a requested *diff*, it should wait for it. This time plus the time to forward the *diffs* to the local processor must be longer than the time of an inter-cluster request. This happens because the *extended cache* depends on a first request to be able to satisfy the following ones. Thus, in order to minimize this waiting time, we are currently working on a new version of our clustered SDSM, where a mechanism for feeding the *extended cache* beforehand is provided. A processor will periodically and asynchronously send to a remote *extended cache* some *diffs* that the processors of this remote cluster might ask for in the future. The good choice of which *diffs* to send requires prediction of application future sharing access pattern. Since temporal locality is proved to be a good heuristic in DSM *diff* prefetch [KAR97] [BIA98], a processor will make its decision of which *diffs* to pre-send to a remote *extended cache,* based on the *diff* requests past history of the processors of this cluster, expecting that the same concurrent sharing access pattern will be repeated in the near future.

Another point that worth discussing is the extra number of request/diff messages that the *extended cache* generates. This network traffic rise can lead to network message contention and application performance degradation. The ideal would be that a processor could choose which are the *diffs* that it desires to request to the *extended* cache. Hence, in our new clustered SDSM version, an adaptive extended cache mechanism will be provided. A processor will send a *diff* request to its *extended cache* only for those pages that it knows that at least one processor of its local cluster already issued a request for it (or will do in the near future) or that it can find in the cache due to the pre-sending mechanism just described.

References

[AMZ 96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony; W. Yu and W. Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *IEEE Computer,* 29(2): 18-28, February 1996.

[AMZ 99] C. Amza, L. Cox, S. Dwarkadas, J. Jin, K. Rajamani; W. Yu and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. In *Proceedings of the IEEE*, 87(3):467-475, March 1999.

[BAI 93] D. Bailey, J. Barton, T. Lansinski, and H. Simon. The NAS parallel benchmark. *Technical Report 103863*, NASA, July 1993.

[BIA 98] R. Bianchini, R. Pinto, C. L. Amorim. Data Prefetching for Software DSMs. In *Proceedings of International Conference on Supercomputings*, July, 1998.

[COX 99] A. Cox, E. de Lara, Y. Hu and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory. In *Proceedings of The Fifth International Symposium on High Performance Computer Architecture*, Feb. 1999.

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

[JIA 97] D. Jiang, H. Shan, and J. P. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* June 1997.

[KAR 97] M. Karllson. and P. Stenström. Effectiveness of Dynamic Prefetching in Multiple-writers Distributed Virtual Shared Memory Systems. *Journal of Parallel and Distributed Computing*, 42(7), July1997.

[KEL 94] P. Keleher, A. L. Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstation and operating systems. In: *The Winter 1994 Usenix Conference,* San Francisco, pages 115-31, 1994.

[PLA99] A. Plaat, H.Bal, R. Hofman. Sensitivity of Parallel Applications to Large Differences in Bandwidth an Latency in two-layer interconnects. In Proceedings of the *Fifth International Symposium on High Performance Computer Architecture*, Feb. 1999.

[SAM 98] R. Samanta, A. Bilas, L. Iftode and J. Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of the 4th Symposium on High Performance Computer Architecture*. February 1998.

[SCA98] D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. *In Proceedings of the Fourth HPCA*, February 1998.

[SIN 92] P. Singh, W. Weber and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5-44, March 1992.

[STE 97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis, S. 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 Systems Principles*, October 1997.

[ZHO 96] Performace Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, October 1996.