A Case for Dynamic Page Migration in Multiple-Writer Software DSM Systems

Thomas Repantis[1] Christos D. Antonopoulos[2] Vana Kalogeraki[1] Theodore S. Papatheodorou[3]

[1] Dept. of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521, USA
{trep,vana}@cs.ucr.edu

[2] Dept. of Computer Science
The College of William & Mary
McGlothlin-Street Hall,
Williamsburg, VA 23187, USA
cda@cs.wm.edu

[3] Computer Engineering & Informatics Dept.
University of Patras
26500 Patras, Greece
tsp@hpclab.ceid.upatras.gr

Abstract

Software DSMs (SDSMs) are an appealing alternative to message passing, since they facilitate the programmability of clusters. However the ease of programming comes at the expense of performance. Although accesses of data that reside to the memory of remote nodes are transparent to the programmer, they suffer from significantly higher latencies compared to local accesses. As a consequence, it is desirable to move data as close as possible to the nodes that need them most.

In this paper we introduce a protocol for dynamically migrating memory pages in home-based SDSM systems. In these systems each page has a designated home node; yet our protocol allows a node that heavily modifies a page to become its new home. The new protocol targets multiple-writer DSMs, i.e. DSMs that allow multiple nodes to concurrently modify the same page. The process is dynamic and transparent to the applications programmer. Moreover, it does not assume a specific consistency protocol.

Experimental results show that our page migration protocol reduces remote page modifications, decreases the average memory access latency, as well as the overhead for the preservation of memory consistency. The benefit for the end-user is a significant improvement in application performance.

1. Introduction

Computer clusters have been established as a flexible and inexpensive platform for high performance computing. Message passing models, which are often preferred for programming clusters, require all data exchanges between processes to be explicitly specified by the programmer. On the other hand, application development can be significantly easier on top of Distributed Shared Memory (DSM) architectures. DSM architectures provide the notion of a globally shared address space to the programmer, although application threads or processes may execute on different cluster nodes, with physically distributed memory (figure 1). The shared address space notion can be implemented in hardware or software. A Software DSM (SDSM) layer may be offered as either part of the operating system kernel or as a run-time library.

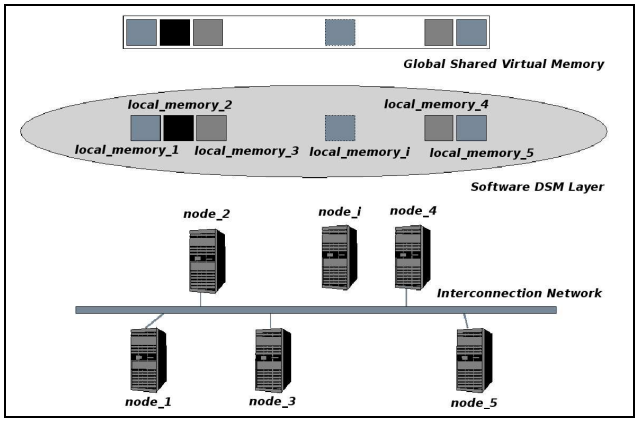

Figure 1. The DSM layer provides the applications programmer with the illusion of a globally shared memory address space.

The programming simplicity that SDSMs offer does however come at a performance cost. The SDSM layer has to preserve consistency of copies of memory pages of the shared address space residing at different nodes. Advances in programming languages for deep multiprocessor systems, such as Unified Parallel C [5] or Co-Array Fortran [18] introduce extensions for selective data sharing, programmer-driven data distribution and association between data and computation. The enriched semantics of those languages can be exploited as hints to an underlying SDSM middleware, in order to effectively reduce the overheads associated with the consistency protocol and improve

the overall performance.

In the simple approach of the sequential consistency model [11], when a shared memory page is requested, the most recent version of that page is fetched. That model, however, delivers poor performance, mainly due to the problem of false sharing of memory pages [9]. If the DSM does not deal with false sharing, it unnecessarily forbids processes to access the same memory pages concurrently, even when the accessed addresses are unrelated. More relaxed memory consistency models, such as Lazy Release Consistency (LRC) [10] deal with this problem. In LRC, modifications of a memory page are only propagated to a process when it acquires a corresponding lock. Even though a protocol like that alleviates the problem of false sharing and allows multiple processes to write on the same shared memory page concurrently, it still has several drawbacks: The memory overhead for keeping consistency-related data in each node is large, as is the communication overhead to gather the modifications of a page from multiple nodes and bring a stale page up-to-date.

Home-based protocols [8] address the aforementioned limitations. Home-based SDSMs associate each shared memory page with a specific node, its home. The home node is responsible for producing and distributing all the copies of the page and for gathering and applying all the modifications (diffs) to it, in order to maintain its consistency. When a page fault occurs, the node attempting to access the page can directly fetch it from the corresponding home node, since the latter is always guaranteed to have a valid copy of the page. Changes on a page by its home node do not produce copies and diffs, whereas changes by other nodes are applied collectively, once, at the home node. Hence, the overheads of diff book-keeping and message exchange are reduced.

The performance of home-based DSMs depends highly on the proper distribution of pages across nodes, with respect to the memory access pattern of the application, since remote page accesses incur extra communication cost, whereas local accesses at the home node can be satisfied instantly. The placement of memory pages together with the processes that access them is crucial, in order to reduce the number of remote memory accesses, the size of produced diffs, the cost of the consistency protocol as well as the average memory access latency.

The benefits of a sophisticated page placement motivate dynamic home migration, i.e. explicitly moving the home of a page from some node to another at run-time. The advantages of employing dynamic home migration in SDSMs are numerous: First, increasing the locality reduces the average latency of memory accesses and the amount of network traffic caused by the transfer of both pages and modifications. Second, the system is able to adapt to the applications' memory access patterns, which may not be known at compile-time or may even change throughout the execution. In these cases, an initial home distribution –no matter how clever it might be– cannot deliver optimal performance. Finally, resource utilization can be improved, since variations in memory availability can be taken into account. If a node does not have enough memory, pages can migrate to other nodes. Adaptation to varying resources and application requirements is achieved with lower overhead than that of traditional approaches which rely on thread migration [26].

Even though several home migration protocols for DSMs have been proposed, they either target only single-writer memory access patterns [6, 7], or introduce further communication overhead by considering multiple homes [20]. Yet, in the majority of shared-memory applications memory pages are remotely modified by multiple nodes concurrently [21, 22].

Application	By Single Node (%)	By Multiple Nodes (%)
SOR	0	100
TSP	50.53	49.47
WATER	81.82	18.18

Table 1. Percentage of pages modified remotely by a single node or by multiple nodes within a barrier interval, over the total number of modified pages, for applications running on top of a Software DSM.

Table 1 summarizes the percentage of pages modified by a single and by multiple remote nodes for a variety of applications running on top of a Software DSM. While fine-tuning the application code to minimize the sharing of memory pages among nodes can improve application behavior, these examples serve as motivation for a migration protocol that detects multiple-writer memory access patterns. A migration protocol that relies on detecting a *single* modifier pattern will not be activated for memory pages that are being *concurrently* modified by multiple nodes, and will thus offer no optimization. On the other hand, complicated migration protocols that introduce several new messages and phases to migrate the pages and to keep them coherent may incur too high of an overhead, not outweighed by the benefits of migration.

In this paper we introduce DPMIG, a low-overhead, efficient, dynamic page migration mechanism. DPMIG dynamically distributes shared memory pages to home nodes and is effective in the presence of both single- and multiple-writer memory access patterns. Each page has an initial, designated home node, however nodes that heavily modify pages become their new homes. The migration decision is taken locally, at the home of each page, based on the sizes of the diffs that have been generated by the nodes that modified

it. To avoid redundant page transfers, we perform migration only when the extent of modifications of a page by a remote node becomes higher than a threshold. Local accesses to a page are also taken into account to avoid migrating a page that is locally accessed by its home. Ping-pong of pages is tackled by detecting pages that have already migrated. The migration information is piggybacked on the existing synchronization messages and thus the communication overhead is minimized.

We have implemented our mechanism on top of the JIAJIA software DSM [1, 23]. The choice of JIAJIA as an implementation platform does not influence the generality of our approach. We evaluated our mechanism using application benchmarks. We quantified its benefits by comparing it with the single-writer migration protocol of JIAJIA (JIAMIG), as well as to the execution without migrations (NOMIG). Our experimental results indicate that the new mechanism significantly reduces remote page modifications and hence network transfers, reduces the overhead of the consistency protocol and achieves lower overall execution time than its competitors.

The rest of this paper is organized as follows: In Section 2 we introduce our implementation platform and we describe in detail the new migration protocol. Section 3 presents the performance evaluation results and analysis. Section 4 discusses related work and finally Section 5 concludes the paper.

2. The Multiple-Writer Dynamic Page Migration Protocol (DPMIG)

2.1. The JIAJIA Software DSM

JIAJIA is a home-based SDSM system, based on a scope consistency protocol. Each node acts as the host of a portion of the shared memory. Memory accesses are local when they resolve to locally homed pages. Remote pages need to be fetched from the respective home nodes and cached locally for subsequent accesses. JIAJIA uses three basic page tables at each node for the management of shared memory. The *home* page table registers pages that have the particular node as a home. The *cache* page table keeps track of the non-homed pages that are cached in the particular node. If the footprint of cached pages gets larger than the available cache, old pages must be evicted to make room for new pages. Finally, the *global* page table, common for all nodes –though physically implemented and maintained as an independent copy on each node– is used to keep track of the location of each page in the home and cache page tables.

JIAJIA implements a scope consistency protocol. The processor that acquires a lock makes its shared memory pages consistent only with respect to previous changes relating to the specific lock. The changes, propagated as write notices relating to the particular lock, enable the acquiring host to identify obsolete cached pages. Barriers, on the other hand trigger consistency operations which result to a totally consistent shared memory image across all participating nodes. Each barrier, as well as the consistency operations associated with it, are handled by a specific host, the barrier manager. The barrier manager changes between consecutive barriers in a round-robin fashion, in order to avoid overloading specific nodes.

JIAJIA offers a single-writer home migration algorithm (JIAMIG) [7], that is conservative in its migration decisions and limited in its applicability. In JIAMIG, the home of a page is migrated to a specific node if that node was the only writer of the page between two consecutive barriers and still has an accurate copy of the page locally cached. Then the migration can occur without an actual page transfer being required. JIAMIG takes advantage of write notices to identify pages that have been singly-written since the last barrier and apparently targets software DSMs that use scope consistency. The migration protocol we introduce, on the contrary, is capable of targeting pages modified by multiple hosts. Moreover, even though our implementation has been optimized to exploit the existing synchronization messages, the protocol itself does not rely on any particular consistency protocol assumptions.

2.2. DPMIG Design and Implementation

We propose a multiple-writer dynamic page migration protocol, which improves the performance of applications by optimally placing pages, even if they are concurrently modified by multiple nodes. If the main page modifier is not the current home of the page, the migration of the page to the main modifier can improve the locality of the application and thus reduce the time-consuming remote memory accesses, diff creations, transfers and applications.

The basic phases of our generic page migration mechanism are the following: *(a) On-line monitoring of the remote diff sizes*, *(b) Migration decisions*, *(c) Propagation of the migration decisions*, and *(d) Page table updates*. An example of the page migration procedure is shown in figure 2. We describe the different phases in detail in the following paragraphs.

2.2.1. On-Line Monitoring of Remote Diff Sizes. All the changes made to a page by any node are sent to the home of that page in the form of diffs. The home of each page is the most appropriate node to monitor the amount of modifications applied to the page and, thus, to identify the node that modified it most heavily. The home extracts the size –in bytes– of the modifications directly from the messages carrying the diffs. In other words, the number of bytes in a page that a node modified is recorded as the size of the

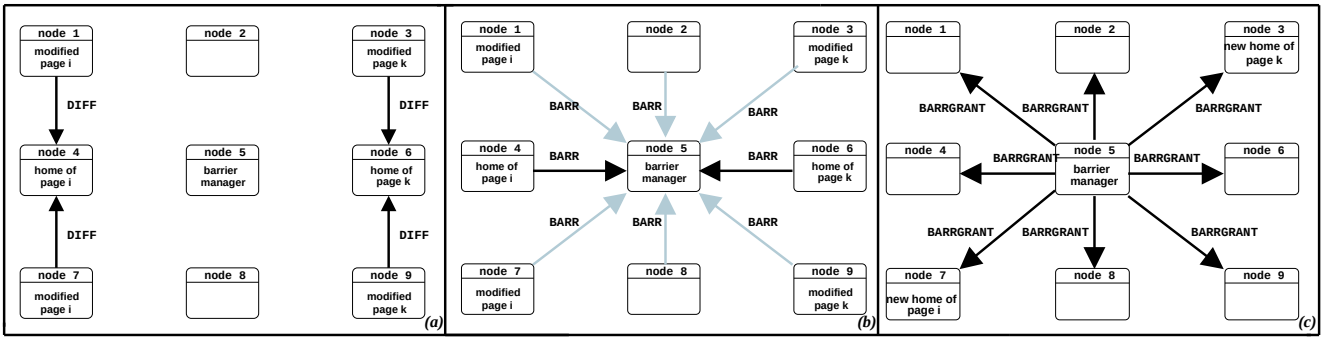

Figure 2. An example of the page migration procedure. Nodes 1 and 7 modified page i, whereas nodes 3 and 9 modified page k (a). According to the sizes of the modifications, page i is migrated from node 4 to node 7 and page k is migrated from node 6 to node 3. The migration decisions are propagated from the homes of the pages to the barrier manager (b) and from the barrier manager to all the nodes (c).

diff coming from that particular node. Modifications by a node on its home pages are done directly and no diffs are created. Those local memory write accesses are however detected and taken into account in the migration decisions.

2.2.2. Migration Decisions. When a host reaches a barrier, it identifies –for each page that it accommodates as a home– the main modifier node, as well as the extent of the modifications contributed from that node. The main modifier of a page is defined as the node that has produced the largest amount of modifications (diffs) for that particular memory page so far. Based on this information, *the home* decides to migrate the page to the main modifier after applying all diffs, only if all following three conditions are satisfied:

1. The modifications are above a certain threshold.
2. The page has not been modified by its current home since the last barrier.
3. The page was not migrated at the previous barrier.

Condition 1: Triggering the migration protocol even for minor modifications would result to excessive migrations. In order to avoid that, the extent of modifications must first reach a certain threshold before the migration procedure is triggered for a page. That threshold depends on the application, the interconnection network, the memory and the processing power of the nodes. Currently the value of the threshold is specified by the application programmer using an API call. For our experiments, we heuristically set this threshold to 512 bytes. Page migration is beneficial even if the threshold is left to the default value of 0; however we are in the process of investigating automated ways to estimate it at run-time.

Condition 2: Since local page modifications are performed without the intervention of the SDSM in the sense that they do not produce diffs, we cannot evaluate their extent and take them into account when reaching migration decisions. It is however reasonable not to risk migrating a page that is being modified by its current home, even if another node has also been modifying it. Hence we detect such pages and do not migrate them. Detecting local page accesses is possible, because even the home of a page has to acquire permission from the SDSM before writing it.

Condition 3: A common problem of page migration algorithms is the ping-pong phenomenon, manifesting itself as pages bouncing between two nodes. In order to deal with this side-effect, we do not allow a home to migrate twice during two consecutive barriers. This allows a page to build some history on the node. Even though this approach just reduces the possibility of ping-pong, we chose not to eliminate the phenomenon by locking pages in certain nodes or prohibiting pages from migrating again to their earlier homes [15]. This way the protocol is still able to adapt to dynamically changing application memory access patterns. In the case of page ping-pong, whether i) allowing a page to migrate, ii) locking it in its current home, or iii) delaying the migration will provide the best performance, depends on the future memory access pattern of the application and is therefore difficult to predict. Delaying repeating migrations, in combination with forbidding locally accessed pages to migrate, helps us in dealing with ping-pong in a non-intrusive way, leaving room for adaptability.

Following the migration of a page, the diff statistics related to it are reset, so that old access information does not affect future migration decisions.

Figure 3. The modified barrier request message, incorporating the addresses of the pages that will migrate from this home, as well as the hosts with the maximum sizes of applied diffs, which will become the pages' new homes.

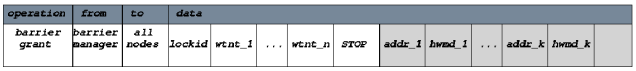

Figure 4. The modified barrier grant message, incorporating all hosts' migration decisions, which were gathered by the barrier manager.

2.2.3. Propagation of Migration Decisions. JIAJIA implements scope consistency, hence the consistency preservation operations take place at synchronization points. Barriers usually mark important synchronization points in parallel applications, therefore we chose to take and apply migration related decisions at barriers as well. In JIAJIA, the "barrier manager" is responsible for collecting and propagating the migration information. The barrier manager changes from barrier to barrier, thus no single host becomes a bottleneck in the procedure. By propagating migration decisions only at barriers, the overhead of the migration mechanism is also minimized.

Each host in the system is responsible for producing information on the migration decisions for the pages it is the home of. The addresses of the pages to be migrated away from the node are piggybacked together with the identifiers of the new homes on the "barrier request" message sent to the barrier manager (figure 3). As a consequence, the barrier manager actually collects all the migration information. When all requests for a barrier have been received, the barrier manager sends the accumulated migration information to all hosts. That information is attached to the "barrier grant" message (figure 4). Thus, while the home of each page is responsible for taking the migration decision, the barrier manager is responsible for propagating the decisions of all homes to all the nodes.

2.2.4. Page Table Updates. Upon receiving a barrier grant message each host updates its page tables (global, home, and cache). The updates are based on the migration information piggybacked on the message. More specifically, each host checks the addresses of the migrating pages, together with the respective new homes and reacts, according to the following rules:

1. If the host is the new home of the migrating page, then:
   (a) If it was not the sole modifier of the page and hence does not have a valid copy of it, it asks the page's previous home for a valid copy. We keep track of arriving pages in order to ensure that the new home will not attempt to access the page before it is received from the old home.
   (b) If it was the single modifier of the page, and still has a valid copy of the page locally cached, no page transfer occurs. If the page is not cached anymore, it is transferred from the previous home.

   A slot is then allocated for the new page in the *home* page table of the host and its place in the *cache* page table is freed.
2. If the host is the old home of a migrating page, then the page's slot in the *home* page table of the host is freed. The page is either invalidated, if there is no free position in the *cache* page table, or preserved as a valid cached copy if a free position is available. We make sure, however, that if the migrating page needs to be transferred to the new home, the old home will not update its page tables and unmap the page before its actual reception from the new home.
3. The *global* page tables of all hosts in the system are updated to point to the new home.

2.2.5. Communication Overhead. From the previous discussion it is clear that DPMIG keeps the message exchanges required for the migration to a minimum. The migration information is piggybacked to the existing barrier request and grant messages. In the case of a single modifier of a page, the migration takes place without an actual page transfer as long as the single modifier still has a valid cached copy of the page. In the case of multiple modifiers though, the new home inevitably has to get the page from the previous one, since this is the only host guaranteed to have a valid, updated version of the page.

By taking and propagating migration decisions at barrier synchronization points, we limit the applicability of the protocol to applications that use barriers as a synchronization mechanism. Apart from the fact that this is often the case – especially in popular programming models for shared memory, such as OpenMP [19]– we also feel that this choice is well balanced due to the simplicity and minimal overhead it provides: Neither extra synchronization phases, nor extra messages are introduced by the migration protocol. There is some minimal overhead imposed on the barrier manager for collecting and propagating the migration information, however this overhead is amortized among all hosts, since a different node adopts the role of the manager for each barrier.

3. Experimental Evaluation

The performance of our protocol (DPMIG) has been compared with both JIAJIA's single-writer home migration protocol (JIAMIG) and the performance of applications when executed without activating migrations (NOMIG). Throughout this section, the results from the NOMIG execution are used as a reference for the calculation of performance improvements, ratios etc.

We carried out our experiments on a cluster of 8 nodes, with AMD Athlon XP processors, clocked at 1674 MHz and equipped with 256 KB of outer-level (L2) cache and 1GB main memory each. Nodes were interconnected with a 100 MBps Ethernet network. The operating system was Linux (kernel version 2.6.7) and the application binaries were created using the gcc compiler (version 3.3.3).

We used the following applications as benchmarks: Water from the SPLASH-2 suite [25], IS from the NAS Parallel Benchmarks [13], a synthetic matrix multiplication (MM) and EM3D from the JIAJIA software DSM distribution [24].

Water simulates forces between 288 different molecules. It uses an array of data structures, each corresponding to a molecule. The array is statically divided into equal parts, each of which is assigned to a processor. Processors use locks to protect the update of force values relating to the molecules. The calculation proceeds in 100 time steps. Barriers are used to ensure that all processes perform calculations corresponding to the same time step, as well as to guarantee global memory consistency at the beginning of each step.

IS applies bucket sort to sort a sequence of 2^{15} keys. Keys are assigned to processors and each one has a local bucket, while there is also a global bucket for all processors. Initially, each processor counts the keys in its local bucket. Afterwards the counts are accumulated in the global bucket during a critical section protected by a lock. Then keys are sent to the local buckets they correspond to, taking advantage of the information collected in the global bucket. Finally each processor sorts its local bucket. The different phases of the algorithm are separated by barriers.

MM performs the calculation $A=(B\times C)^{100}$ with matrices of dimension 1024x1024 each. Matrices A, B and C are shared by all processors. The $A\times B$ matrix multiplication does not require any synchronization, since each processor computes a different band of lines of the resulting matrix. However, barriers have been introduced between the consecutive matrix multiplications used to raise the result of $A\times B$ to the power of 100.

EM3D is a parallel, finite difference, time domain algorithm that computes the resonant frequency of a waveguide loaded 3D cavity. The main shared data structures are three electronic and three magnetic field components as well as eight coefficient components. The electronic and magnetic field components are updated alternatively in the x- and y-directions, whereas parallelism is exploited in the z-direction. The computation progresses in 100 time steps, with barriers between consecutive time steps to enforce memory consistency. The application is hand-optimized for the JIAJIA SDSM, so that the initial data allocation is optimal.

The first performance metric used for the comparison among the page migration strategies is the *execution time* of applications, expressed as the performance improvement over the execution without migration. This metric is a good indication of the overall end-user experience. The results are depicted in figure 5.

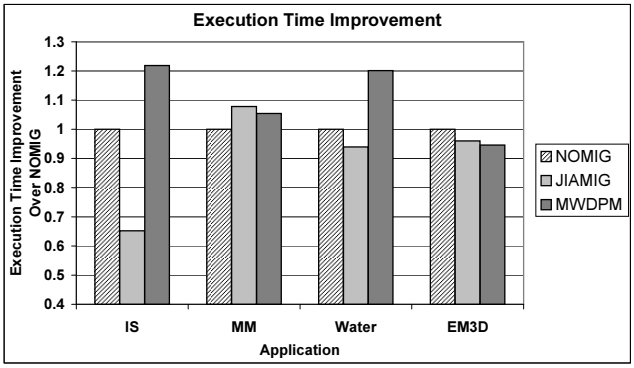

Figure 5. Execution time improvement attained by the activation of the DPMIG and JIAMIG page migration policies, over the execution without migrations (NOMIG).

Figure 5 shows that the DPMIG policy achieves performance improvement of up to 18% (as shown in the case of IS). JIAMIG's policy on the other hand results to slowdowns on all applications except MM. MM is the only application where all pages are singly-written by the same node throughout the execution, and therefore JIAMIG manages to improve performance. DPMIG behaves significantly better. All applications, with the exception of EM3D benefit. EM3D suffers a minimal slowdown under both DPMIG and JIAMIG. Its performance is practically indistinguishable from that of the NOMIG execution. The reason, as it is further explained in the following paragraph, is that the code has already been optimized by the application programmer.

Figure 6 depicts the *ratio of home migrations over the total number of pages* under the DPMIG and JIAMIG protocols. This ratio is not equivalent to the percentage of pages having their home migrated, since the home of the same page may migrate to more than one nodes throughout the execution. However, it is a good indication of how often

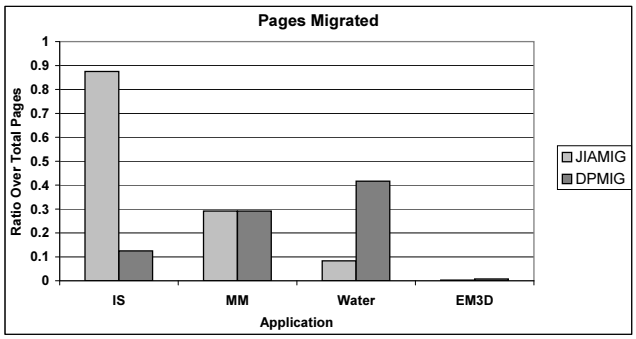

Figure 6. Ratio of home migrations triggered by the two protocols, over the total number of pages.

each migration protocol is triggered. The home migration ratio of EM3D is less than 0.01 in both cases (155 and 63 out of 21176 pages for DPMIG and JIAMIG respectively). As aforementioned, the application is hand-optimized for execution with JIAJIA. It has a predictable memory access pattern, which allows an optimal initial allocation of shared data to be applied off-line. None of the migration protocols manages to get triggered enough to allow home migrations to have measurable effect in the execution time. For MM, on the other hand, the migration ratio is 0.29. The experimental results from the execution of IS and Water indicate that the ratio of home migrations over the total number of pages can not alone offer a safe indication of a protocol's effectiveness. JIAMIG reaches a migration ratio of 0.88 for IS, since many pages have a single modifier some time during the execution. However, the modifications are minor and do not justify the migration, nor does the access pattern generally repeat. DPMIG, which takes into account the extent of modifications before triggering a migration is more conservative and limits the ratio to 0.13, resulting however to better performance than JIAMIG. The behavior of the two protocols is the opposite for Water. The relatively irregular modifications pattern of Water limits the opportunities for page migrations under the JIAMIG protocol. DPMIG looks for migration candidates beyond singly-written pages and manages to perform more migrations, attaining better performance compared with both JIAMIG and NOMIG.

Table 2 summarizes the *network traffic*, in MBytes, produced by the four benchmarks under the three different migration strategies. The sophisticated page placement achieved by DPMIG results to an average 30.51% reduction of network traffic, compared with the execution without migrations (NOMIG). JIAMIG also reduces network traffic. It moves 9.95% less data than NOMIG. Despite the fact that DPMIG may require page transfers during home migrations –should the new home not be the single modifier of the

	IS	MM	Water	EM3D
NOMIG	5266.05	816.52	320.16	169.98
JIAMIG	5264.35	81.57	320.46	222.14
DPMIG	4065.37	81.64	223.09	174.64

Table 2. Total network traffic (in MBytes) caused by each application under the three different page migration strategies.

page between the last two barriers– it produces less overall network traffic than JIAMIG. The better home placement it achieves compensates for the additional page transfers. DPMIG results to more data traffic than NOMIG only in EM3D. The additional network traffic is 2.74% or just 4.66 MBytes. The minimal overhead indicates that, even in cases the DPMIG migration protocol has no room to improve the locality of memory accesses, it will not have adverse effects on application performance. It should be pointed out that in the case of EM3D JIAMIG causes even higher network overhead than DPMIG.

In the case of MM it is also noteworthy that, although both migration protocols manage to reduce the network traffic by a factor of 10, this reduction does not translate into a significant improvement of the execution time. As aforementioned, all pages of the result matrix (C) are singly written and, after the first step, the modifier nodes have valid copies of them. In other words, the overhead of the consistency protocol is limited to the calculation of diffs and their transfer to the corresponding home nodes. However, since no node ever requests a page from its home node, this procedure is actually not in the critical path of the execution and overlaps with the wait time at barriers for the most part.

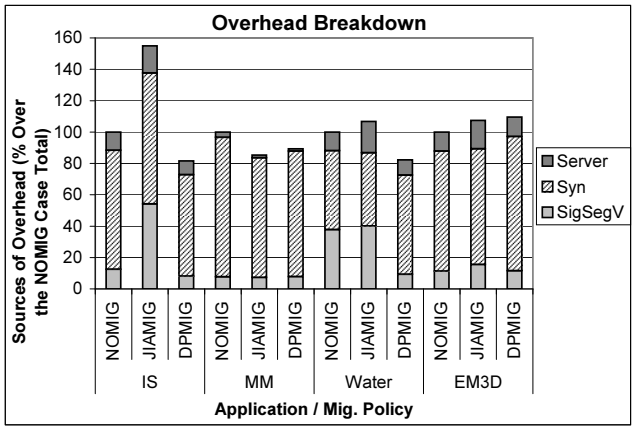

Figure 7. Overhead breakdown for each application under the three page migration strategies.

Figure 7 depicts the *overhead* caused by the memory consistency protocol when it is combined with the three migration strategies. The total overhead is, in all cases, normalized with respect to that of the execution of the same application without migrations (NOMIG). Given that the page migration mechanism is integrated with the memory consistency operations, the measurements capture the overhead caused by the migration protocol as well.

Three different sources contribute to the total overhead: (i) The Segmentation Violation Handler (*SigSegV*), which resolves memory accesses that cannot be satisfied inside the node without DSM intervention, either because the accessed page is not locally homed or cached, or because the node does not have the correct access privileges. (ii) The Memory Server (*Server*) of each node responds to requests from the SigSegV handlers of remote nodes concerning locally homed pages. (iii) The Synchronization Server (*Syn*) is responsible for handling synchronization requests among different nodes, in the form of locks or barriers. However, since the triggering points of JIAJIA's memory consistency protocol are closely related to synchronization operations, the Syn server is also responsible for memory consistency preservation as well. Moreover, page migration protocols also execute in the context of that server.

DPMIG reduced the overall overhead by 16.12% in average. JIAMIG, on the contrary, increased the overhead by 38.65%, in comparison with the execution without page migrations (NOMIG). The sophisticated page placement achieved by DPMIG results to better locality and as a consequence lowers the cost for the preservation of memory consistency. The rigid JIAMIG protocol, on the other hand, did not manage, at least for the specific applications, to trigger the appropriate page migrations. It often proved oversensitive, as in the case of IS, or undersensitive, as in the case of Water. As a result, the overall overhead is higher than that of NOMIG, due to both the cost of executing the migration algorithm and the overhead of inopportune page migrations.

Interesting conclusions can also be drawn from the breakdown of contributions of the three individual sources to the total overhead. DPMIG improves the locality of memory references, which results to a measurable reduction of the execution time of the SigSegV handler and the memory server. The average reduction, in comparison with NOMIG, is 38.84% and 29.06% respectively. This is not the case for JIAMIG, which results to 227.10% and 45.74% higher SigSegV and memory server overheads than NOMIG. The performance problems of JIAMIG are more evident in the case of IS. However, even if IS is excluded, the SigSegV handler and memory server are again more time consuming than in the case of NOMIG, this time by 5.73% and 29.83% respectively.

Despite the fact that the synchronization server overhead is inflated by the cost of the migration protocol, DPMIG manages to reduce that overhead by 11% over NOMIG, since the educated distribution of page homes to nodes results to less memory consistency operations at synchronization points. Again though, JIAMIG is not successful at compensating for the cost of the home migration protocol through better locality. The synchronization server consumes in average 4.09% more cycles than when home migrations are not activated.

4. Related Work

Since the introduction of Ivy [12], the first Software DSM, many techniques have been proposed to improve SDSM performance. Due to space limitations, we focus mainly on previous efforts related to dynamic home migration.

Fang *et al* [6] introduce an adaptive home migration protocol that detects only single-writer patterns and is implemented in a distributed Java Virtual Machine. They employ a per-object threshold that is adjusted continuously to guide migration decisions. The threshold is tuned according to feedback generated from previous home migration decisions made at run-time.

Peng and Speight [20] propose a home-based lazy release consistency protocol, in which the number of home nodes for each shared page can vary. The protocol relies on multicast –if available– to propagate page updates to all the homes simultaneously and reduce the communication cost. Home nodes can change dynamically according to predictions based on the memory access patterns. Predictions are based on the comparison of the time needed to update a page versus the time needed to fetch it.

In Moving Home-Based Lazy Release Consistency (MHLRC) [3], a page migrates to another node if: i)There have been no modifications to it by the home node in the current interval, and ii) the page was not fetched in the current interval. This page migration protocol targets only limited memory access patterns, such as single-writer, where the new home already has the latest version of the page. To avoid broadcasting home changes, a linked list that points to the current home is generated along the migration route and multicasting is used. At every home transfer the previous home node creates a pointer to the new home node. Apparently traversing through the linked list can have a non-negligible overhead.

A home migration protocol has also been implemented as part of the Orion SDSM [14]. Again, it targets only single-writer (producer) memory access patterns. Furthermore, page migration decisions are not combined with any existing synchronization mechanism. Hence, broadcast messages are needed to announce a page migration. Moreover, page updates and requests may be directed to the old home during a transient period and need to be forwarded to

the new home.

JUMP [2] is another page migration protocol, where the processor asking for a page becomes the new home of it, regardless of whether it was the single writer or not. When two processors ask to become the home for the same page, the one that asked first becomes the new home. Migration notices are used to inform nodes about the home change, that may introduce traffic overhead. No effort is made neither to try to migrate a page to the node that actually needs it the most, nor to prevent unnecessary migrations.

We have already described the home migration protocol implemented in the JIAJIA SDSM [7], in which a singly written page during a barrier interval is recognized by the barrier manager and is migrated to the processor that modified it. Migration messages are piggybacked on barrier messages, thus, no extra communication between the nodes is required. The authors illustrate that they can reduce diffs significantly and improve application performance, compared to statically distributing the pages to the nodes. Our work extends that approach towards an adaptive multi-writer home-based migration protocol that is less conservative in its migration decisions and applicable to more types of memory access patterns. The basis of our page migration algorithm together with preliminary results were introduced in [21, 22]. In the current paper we present our complete protocol together with an extensive experimental evaluation.

Nikolopoulos *et al* have shown that user-level dynamic page migration at run-time can improve the performance of parallel applications on NUMA hardware DSMs. Monitoring of memory page references [15] is combined with feedback from the operating system kernel scheduler [17] to keep threads together with their memory affinity sets. The authors have also shown that the OpenMP run-time environment can benefit from page migration in hardware DSMs to offer implicit data distribution and redistribution schemes without programmer intervention [16]. The benefits of dynamic page migration are more profound in SDSMs, due to the significantly higher remote / local memory access latency ratio involved. At the same time, page migration is more challenging in SDSMs, because SDSMs do not – and technically can not – provide any support for monitoring local monitory accesses, similar to the functionality available in certain hardware DSMs and exploited in [15–17].

Concerning the execution of OpenMP applications on top of software DSMs Costa *et al* have also proposed to encourage cooperation between the SDSM and the OpenMP runtime [4], instead of relaxing the consistency semantics [10].

5. Conclusions

In this paper we introduced DPMIG, a simple yet efficient dynamic page migration mechanism for home-based software DSMs. Unlike previous efforts, our mechanism is applicable in the presence of both single- and multiple-writer memory access patterns. The new mechanism is general and at the same time keeps migration-related communication minimal. It does not presume any specific memory consistency protocol, however its implementation in JIAJIA is closely integrated with the scope consistency protocol, thus minimizing its overhead.

We evaluated DPMIG using application benchmarks. By comparing our multiple-writer migration protocol to a single-writer one (JIAJIA's home migration protocol), as well as to operation without migration, we were able to quantify its benefits. Our experimental results show that our mechanism significantly reduces remote page modifications and hence network traffic, reduces the memory consistency protocol overheads and achieves better performance than its competitors.

We are currently experimenting on the automatic calculation of the migration threshold at run-time. The estimate should ideally take into account static technical characteristics of the system as well as dynamically changing parameters of the execution environment. Moreover, we are evaluating the applicability in the context of SDSMs of techniques traditionally used to reduce the average memory access latency in multiprocessors, such as prefetching or precomputation.

Acknowledgments

We wish to thank Dimitrios Serpanos and Dimitrios Nikolopoulos for their helpful comments during the initial phase of this work.

The second author is supported by NSF grants ITR(ACI-0312980) and CAREER(CCF-0346867) and the European Commission through IST grant No. 2001-33071. The third author is supported by NSF Award 0330481.

References

[1] Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences. Distributed Shared Memory. http://www.ict.ac.cn/chpc/dsm, 1999.

[2] B. W. L. Cheung, C. L. Wang, and K. Hwang. A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, 1999.

[3] J. W. Chung, B. H. Seong, K. H. Park, and D. Park. Moving home-based lazy release consistency for shared virtual memory systems. In *Proc. of the 1999 International Conference on Parallel Processing (ICPP'99)*, pages 282–290, 1999.

[4] J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. In *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 35–42, 2004.

[5] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proc. of the SuperComputing 2002: High Performance Networking and Computing Conference (SC'02)*, pages 1–26, 2002.

[6] W. Fang, C. L. Wang, W. Zhu, and F. C. Lau. A Novel Adaptive Home Migration Protocol in Home-based DSM. In *Proc. of the 2004 IEEE International Conference on Cluster Computing (Cluster'04)*, pages 215–224, 2004.

[7] W. Hu, W. Shi, and Z. Tang. Home Migration in Home-Based Software DSMs. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[8] L. Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, 1998.

[9] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An Evaluation of Software-Based Release Consistent Protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, 1995.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th International Symposium on Computer Architectyre (ISCA'92)*, pages 13–21, 1992.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[12] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 International Conference on Parallel Processing (ICPP'88)*, pages 94–101, 1988.

[13] NAS. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/, 1997.

[14] M. C. Ng and W. F. Wong. Adaptive Schemes for Home-based DSM Systems. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[15] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. of the 2000 International Conference on Supercomputing (ICS'00)*, pages 119–130, 2000.

[16] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in OpenMP? In *Proc. of the SuperComputing 2000: High Performance Networking and Computing Conference (SC'00)*, pages 47–60, 2000.

[17] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In *Proc. of the 2000 International Conference on Parallel Processing (ICPP'00)*, pages 95–104, 2000.

[18] R. W. Numrich and J. L. Steidel. F^{--}: A Simple Parallel Extension to Fortran 90. *SIAM News*, 30(7), Sep 1997.

[19] OpenMP Architecture Review Board. OpenMP API. http://www.openmp.org, 2004.

[20] S. Peng and E. Speight. Utilizing Home Node Prediction to Improve the Performance of Software Distributed Shared Memory. In *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 59–66, 2004.

[21] T. Repantis. *Implementation of Page Forwarding on Clusters.* Diploma thesis, University of Patras, Greece. http://www.cs.ucr.edu/∼trep/tsrDiplThesis.html.

[22] T. Repantis, C. D. Antonopoulos, V. Kalogeraki, and T. S. Papatrheodorou. Dynamic Page Migration in Software DSM Systems. In *Proc. of the 2004 IEEE International Conference on Cluster Computing (Cluster'04) (poster session)*, page 494, 2004.

[23] W. Shi. *Improving the Performance of Software DSM Systems*. PhD thesis, Institute of Computing Technology, Chinese Academy of Sciences, 1999.

[24] W. Shi, J. Ma, and Z. Tang. High Efficient Parallel Computation of Resonant Frequencies of Waveguided Loaded Cavitives on JIAJIA Software DSM System. In *Proc. of the 7th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99)*, 1999.

[25] SPLASH. Stanford Parallel Applications for Shared Memory. http://www-flash.stanford.edu/apps/SPLASH/, 2001.

[26] K. Thitikamol and P. Keleher. Thread Migration and Communication Minimization in DSM Systems. *The Proceedings of the IEEE*, 87(3):487–497, March 1999.