Implementation of Page Management in Mome, a User-Level DSM

Yvon Jégou

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implementation of Page Management in Mome, a User-Level DSM

Yvon Jégou

N°4999

Novembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4999--FR+ENG

Implementation of Page Management in Mome, a User-Level DSM

Yvon Jégou *

Thème 1 — Réseaux et systèmes
Projet Paris

Rapport de recherche n°4999 — Novembre 2003 — 16 pages

Abstract: This paper describes the implementation of the page management in Mome, a user-level distributed shared memory (DSM). Mome provides a shared segment space to parallel programs running on distributed memory computers or clusters. Individual processes can request for mappings between their local address space and Mome segments. The DSM handles the consistency of mapped memory regions at the page-level. A node can freely select the consistency model which is applied to its own view of a page among two models: the classical strong consistency model and a simple and very basic weak model. Under the weak model, each process of the parallel application must send a consistency request to the DSM each time its view of the shared data needs to integrate modifications from other nodes. Mome targets the execution of programs from the high performance community using an SPMD computation model and the coupling of these simulation codes using an MIMD model.

Key-words: DSM, memory mapping, code coupling, consistency management

(Résumé : tsvp)

* mél : Yvon.Jegou@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Mise en œuvre de la gestion des pages dans Mome, une mémoire partagée répartie en espace utilisateur

Résumé : Ce document décrit la mise en œuvre de la gestion des pages dans Mome, une mémoire partagée répartie (MPR) fonctionnant dans l'espace utilisateur. Mome fournit un espace de segments partagés à des programmes qui s'exécutent sur des calculateurs à mémoire partagée ou des grappes. Un processus peut demander la projection de segments partagés de Mome dans son espace d'adressage. La MPR Mome gère la cohérence des copies des segments avec la granularité des pages virtuelles. Chaque processus peut choisir le modèle de cohérence qui doit être appliqué à sa propre vue de chaque page parmi deux modèles prédéfinis : le modèle de cohérence stricte classique et un modèle de cohérence relachée simple. Sous le modèle relaché, le processus doit expédier une requête de cohérence à la MPR à chaque fois qu'il désire intégrer les modifications des autres processus dans sa vue de l'espace partagé. Mome cible l'exécution de codes numériques du domaine du calcul haute performance suivant un modèle d'exécution SPMD ainsi que le couplage de codes de ce type suivant un modèle MIMD.

Mots-clé : DSM, MPR, projection en mémoire, couplage de codes, cohérence de mémoires

1 Introduction

Software Distributed Shared Memory systems (DSM) have been extensively studied in the past. Many of these DSM provide some form of common address space for parallel programs running on distributed memory architectures. The target parallel execution model is very often close to the widely available Posix thread model. Using these models, codes developed for SMP architectures can run on clusters of workstations and on distributed memory architectures. Many of these DSM relax the sequential consistency model. They take advantage of the semantics of the parallel execution model. Parallel processes need to have identical views of shared data only when they synchronize. TreadMarks [1] and JIAJIA [4] are two typical DSM implementations which manage shared memory consistency on synchronization primitives. On a release consistency software DSM, allocating the data in the shared space is sufficient to enforce the program semantics: the nodes get a valid copy of the data as long as conflicting requests are protected by synchronization primitives.

The Posix memory model allows heterogeneous processes to share memory sections through file mapping and shared segments. But the consistency model applied to these multiple views of the same file is defined only for the case where the processes are under the control of the same operating system (single system image). The consistency model in use when sharing files through NFS, for instance, is not defined. The Mome DSM allows heterogeneous processes running on distributed memory architectures and clusters to share data through the memory mapping of segments into their address space. Each parallel process can freely select the consistency model which must be applied to its own view of the shared data. The behavior of Mome has been evaluated in various configurations: high performance computing using the SPMD execution model, code coupling through segment sharing on clusters, shared data repository or dynamic coupling of parallel codes.

The next section presents the execution models supported by Mome, section 3 the consistency models implemented by the DSM and section 4 the implementation of page management. Section 6 compares the implementations of Mome and some other DSM.

2 Mome run-time model

Parallel programs share data using the Mome DSM through the mapping of Mome segments on their private address space. A Fortran interface as well as a C interface are available in the current implementation.

2.1 Segment creation and memory mapping

Each segment of Mome is referenced using the same unique identifier on all processing nodes supporting the DSM. A new segment is created through a call to the `MomeCreateSegment(size)` routine. The routine returns a valid segment identifier which can be used for mapping requests by all the computation nodes.

A process requests for a mapping between a local memory section and a Mome segment section using the call

```
A= MomeMMap(Addr, Lg, Prot, Flags, Seg, Offset).
```

This routine uses the same parameter list as the classical Posix `mmap` system call where the file descriptor is replaced by a segment identifier. This routine returns the starting address of the mapped region. The process can fix the mapping address through the `Addr` and `Flag` parameters or leave the address selection to the DSM. A non null `Addr` specification suggests a mapping address to the DSM. If a fixed mapping is requested (`MAP_FIXED` flag), the segment offset `Offset` and the address `Addr` must be aligned with the same offset on a page boundary. If the mapping is free and the parameter address is null, the DSM selects a mapping region aligned on the segment offset. In the case where the mapping is free but the mapping address is valid, the segment offset is increased until it is aligned on the requested address before the mapping is applied. This last rule is slightly different from the standard Posix rule. It allows the mapping of statically allocated Fortran arrays or common blocks on the DSM. Using this features, old "dusty-deck" Fortran program can be run without the need to reorganize all array declarations and to remove the common blocks. Before a new mapping is established, all previous mappings of the memory region are removed. On the other hand, nothing prevents the same process to map the same segment region many times, the DSM guarantees the consistency of identically mapped memory regions. Mome does not exploit the presence of these multiple mappings for the implementation of finer grained consistency management as in [5].

The mapping requests of the parallel processes are independent, and the addresses of the mappings need not be the same on all computation nodes. However, through the use of fixed mappings, it is possible to map the same regions at the same addresses and to share pointers inside these regions. Mome segments retain their data even once all mappings have been removed.

2.2 Typical uses of Mome DSM

Running Fortran77 codes

Fortran77 codes can be parallelized and run on distributed memory architectures using the SPMD execution model with few modifications [6]. Statically allocated memory (or common blocks) can be mapped on Mome segments at run time. Shared variables need not be allocated dynamically.

Object sharing

Two (or more) parallel applications can share objects through the mapping of the same DSM segments into their address space. These mappings need not be requested at the same location in the address spaces as long as the shared data does not contain pointers. The run-time system associated to Mome provides intra- and inter-application synchronization primitives through node groups.

Dynamic coupling of parallel applications

When coupling applications, the coupling library must locate the coupled objects in the address space of the applications in order to organize the data transfers. The mapping of the shared objects on a DSM segment removes the burden of locating the elements in the different memories from the coupling library [7]: the DSM segment references are uniform across the coupled applications. The Mome DSM allows multiple mappings of the same segment for a process. Using this feature,

the coupling library can use its own mapping to avoid relying on the memory management of the applications.

3 Shared data consistency levels

Two consistency levels are currently supported by Mome: strong consistency and weak consistency. For a given page, each computation node is free to dynamically request its own consistency management during the program execution.

Strong consistency level

The strong consistency level is implemented in Mome using the classical invalidation scheme: before any node receives the right to modify a page, this page is invalidated on all nodes which have asked for strong consistency management of this page. If all processes request strong consistency on all shared space, Mome implement the classical sequential consistency model.

Weak consistency level

The management of weak consistency by the Mome DSM is based on a scalar distributed clock. This distributed clock is kept consistent by the communication layer across all computation nodes. The major events of the parallel program life can be dated explicitly through clock shift requests or automatically by the DSM run-time system, for instance, on a synchronization barrier or a mutex release. Each computation node keeps track of a *valid-before* date and of a *consistency-constraint* date for each page. The current copy of the page integrates all modifications from all computation nodes up-to the *valid-before* date. The application can update the *consistency-constraint* date using a consistency-request call to the DSM.

```
MomeSyncMem(Address, Length, Date)};
```

All access rights to the current copy of the pages are automatically removed as long as the *consistency-constraint* date is more recent than the *valid-before* date. Any access to an invalidated page generates a page-fault. The page manager can re-validate the page (using a new *valid-before* date) or replace the whole page by a valid (more recent) version.

The application can fetch the date associated to synchronization events, for instance, the date of the last synchronization barrier or the date when some mutex lock was released, and use this date in a consistency request.

The strategy in our DSM is different from the strategy adopted by release consistency DSMs in that no information about local modifications is propagated on synchronization points. Our strategy can be compared to the entry consistency strategy: consistency requests are applied when entering a new region. But, using Mome, the consistency request is applied to a memory region and is not associated to any synchronization variable.

The weak consistency level implemented in Mome is consistent with the target computation models: loop parallelism in the high performance computing domain and code coupling. The parallelization process applied during compilation of numerical Fortran code is based on data access

analysis inside loop bodies and dependency computation. It is, in general, possible to predict which sections of the shared arrays are going to be read or modified by each computation node before entering a new parallel loop. This knowledge is exploited when compiling HPF [2] programs for distributed memory architectures: the compiler inserts update requests on the memory objects accessed during the parallel loop execution. If Mome is the run-time target, the compiler must insert calls to the run-time system specifying the sections of the shared memory which must be made consistent. In fact, the Mome based run-time system for HPF can be designed using the same interface as the message based run-time system. No need, in this case, for the development of specialized compilation techniques. Moreover, the presence of a linear address space in Mome avoids the generation of complex index manipulations inside the loop bodies and simplifies the run-time system development.

Loosely coupled applications need not have a permanently consistent view of the shared space: in general it is possible to insert consistency requests after the synchronization calls. In the case where it is difficult to characterize the consistency needs of some application, it is always possible to request for a strong consistency management of the shared space for this application.

3.1 Page management

Mome is a directory-based page management DSM. Each page directory records the status of one page for each node. Currently, the behavior of the page manager is defined using six sets of nodes (one bit per node): the set V of nodes with the current version, the set M of nodes with modifications, the set S of nodes which have asked for strong consistency management, the set I of invalidated nodes, the set F of nodes which have initiated a modification merge and the set H of nodes which have seen the current version (see section 5).

When a copy of the current version of the page needs to be forwarded to some node, the sender is selected from V. It is possible for V to be empty only in the one-writer case and, in this case, M contains exactly one node. M is non-empty if one or more nodes have been allowed to modify the current version. If this set contains two or more nodes, V cannot be empty: the current version of the page must be present when merging the modifications (see section 3.3). Set S records all nodes having requested read or write access right to the page in strong consistency mode. S is a subset of $V \vee M$. The page manager is aware of the consistency level in use on some node only when the page is present on this node. All requests sent from a DSM node to a page manager specify the current consistency mode on this node. As long as this set S is not empty, no other node can be granted write access to the page. S can contain multiple readers or a single writer. In the case where S contains one or more readers, M is empty. If the global manager receives a write-request, all the nodes from S are asked to invalidate their copy of the page and to forward a write acknowledgment to the future writer. The future writer is allowed to proceed only when it has received all the write acknowledgments. Set I records the nodes which have modified the current version and forwarded their modifications. The current version is invalid for these nodes because it does not contain their previous modifications. Any request from an invalidated node generates a new version of the page integrating all modifications.

3.2 Consistency management

A new version of the page must be created each time the current version does not respect the constraint received from some node. Each page directory keeps track of the *valid-before* date of the page. The current version of the page integrates all modifications up-to the date when the first modification rights to the current version were granted. The *valid-before* date is the current date as long as no modifications to the current version have been allowed. All requests from the nodes contain a consistency date. The page manager compares the two dates in order to check if the current version of the page respects the constraint.

A new version of the page is created when pending modifications need to be integrated in order to satisfy a consistency request. Set M records all the nodes which have been allowed to modify the page. The page manager selects one node P in charge of merging the modifications (from $V \wedge M$ when possible) and requests all other nodes from M to forward their modifications to P. Node P must have the current version of the page in order to initiate the merge operation. The result of the merge is a new version of the page and is the base for serving future page faults. After merging the modifications, $V = \{P\}$.

3.3 Multiple writers

Merging the modifications from multiple writers costs CPU, memory and communication resources. Many projects such as TreadMarks [1] consider that the communication layer is the most critical resource and compact the modifications through diffing. But diffing consumes CPU and memory: a twin of the page must be created by each writer, the twin and the modified page must be compared when creating the diff message.

The current implementation of Mome follows a different strategy. The interconnection network (SCI, Myrinet and GigabitEthernet) is not the critical resource of our experimental platform. On the other hand, the optimization strategies for numerical codes try to improve the locality of memory accesses: optimizing the memory hierarchy behavior reduces the need for multiple writers. The cost of twinning and diffing should be avoided when only one node modifies a page.

The Mome page managers use the following strategy:

1. The first writer is allowed to modify its local copy of the page without twinning (no extra CPU nor memory cost). The case where this node is the sole writer is favored. If this was the unique copy of the current version of the page, V becomes empty, the page manager enters the single writer mode and a new version will be created as soon as another node requests read or write access to the page.

2. Read page-faults are served using existing copies in V.

3. If a second node requests write access to the page and the current version respects the consistency constraints, this second node is allowed to modify a copy of the page (it is asked to keep a twin of the original version): this node is added to sets V and M.

4. Additional writers can be allowed to modify the page without twinning.

5. When modifications are merged, one of the nodes with a twin is selected by the page manager and becomes the merger. The other nodes from M are requested to forward their modifications to the merger. If the number of modified page is above some predefined threshold, two or more intermediate mergers are selected for a partial merge.

In the current implementation, the modified pages are sent to the mergers without diffing. The modifications are merged in Mome using a simple bit-wise exclusive or ($\oplus$) of the modified pages. If the number of modified pages is odd, and if each bit has been modified by a single node, the original version of the page is not needed.

$$n\ odd : R = P_1 \oplus P_2 \oplus ...P_n$$

In the case where the number of modified pages is even, the original version V is added to the list of pages.

$$n\ even : R = V \oplus P_1 \oplus P_2 \oplus ...P_n$$

The proof of this simple algorithm is straight-forward. The exclusive or $\oplus$ operator is associative and commutative. Exclusive OR-ing an even number of equal bits produce bit value 0. Exclusive OR-ing of bit 0 with the remaining bit produces this remaining bit. A slightly more complex algorithm is available for the case where multiple nodes store the same value into the same location.

$$R = ((V \oplus P_1) \vee (V \oplus P_2) \vee \ldots \vee (V \oplus P_n)) \oplus V$$

The first part of this expression evaluates which bits have been modified. The second part complements the bits which have been modified. This second algorithm needs the presence of the base version of the page. Although this algorithm is more complex than the first version, the current implementation shows equivalent performance: the two algorithms make the same memory accesses. An efficient implementation of these algorithms should be possible using SIMD extensions to instruction sets such as MMX, SSE, SSE2 or AltiVec. Note also that these basic algorithms do not depend on the granularity of the modifications. A slightly more expensive implementation has been developed for the detection of conflicting modifications to the same location. Conflicting writes produce random results. Early detection of these conflicts is necessary for parallel program debugging. The granularity of the writes must be known for this detection. It is possible to tolerate conflicting but identical writes using the second merge algorithm.

4 Mome implementation

4.1 Page management implementation

In the current implementation of Mome, the state of each segment page is described globally by a page manager and locally to each node by a page automaton. The main purpose of the page automatons is to allow the processing of concurrent transactions on the same or on different pages and to serialize the requests. A request queue is associated to each automaton.

All requests received by a page directory are always handled immediately, even when the processing of one request results in many transactions sent to the different nodes. A page directory is never locked while waiting for an acknowledgment from a node. All requests sent by the page manager are considered as being handled immediately by the destination automaton. In fact, each page automaton is in charge of reordering and processing the messages received from various sources: the page directory, other automatons through the communication layer, the application on explicit request to the run-time library – consistency request, prefetch request ... – or on page-fault, the memory manager etc. For instance, it is in charge of handling the case where some DSM node is asked to forward a copy of the page although it has not yet received this page.

4.2 Memory mapping implementation

Figure 1 shows a possible organization of the Mome DSM memory on one computation node after the creation of one shared segment and the mapping of two memory sections M1 and M2 on this segment. All the memory pages managed by the Mome DSM are allocated in the DSM memory. This memory section is allocated at initialization time and then mapped on a local temporary file, the DSM file. Each page of the DSM memory corresponds to a block of the DSM file. A page of the DSM memory is made accessible to the application through the mapping of the corresponding block of the DSM file at the address selected in the application address space. This separation between the DSM mappings and the application mappings allows to distinguish the access rights to the same data from the DSM and from the application.

In Figure 1, the local application has requested two mappings between its address space and the DSM space:

- a mapping of six pages between memory section M1 and the DSM segment at offset 0,
- a mapping of four pages between memory section M2 and the DSM segment at offset 8.

Memory section M1 is managed by items 0 to 5 of the segment descriptor. Memory section M2 is managed by items 8 to 11 of the segment descriptor.

Pages 1, 2 and 5 of memory section M1 as well as pages 0 and 3 of section M2 have no associated memory: the first memory access to these pages by the application results in a page fault. Page 0 of section M1 is mapped on block 6, page 3 on block 8 and page 4 on block 14. The current mappings of section M2 are: page 1 on block 10 and page 2 on block 11. Note that it is possible for two memory sections to share mappings on the DSM memory if the mappings of these memory sections on the same DSM segment overlap.

In figure 1, the size of the DSM memory is 22 pages. This memory section must be large enough to contain all pages present simultaneously during the execution. On the other hand, this DSM memory should not be too large in order to avoid disk paging on the computation node. A segment descriptor is created on each node when a new segment is created. In our example, the segment contains 12 pages. Each block of a segment descriptor corresponds to one page and contains the local page automaton and three page references to the local DSM memory:

- V points to the current version,

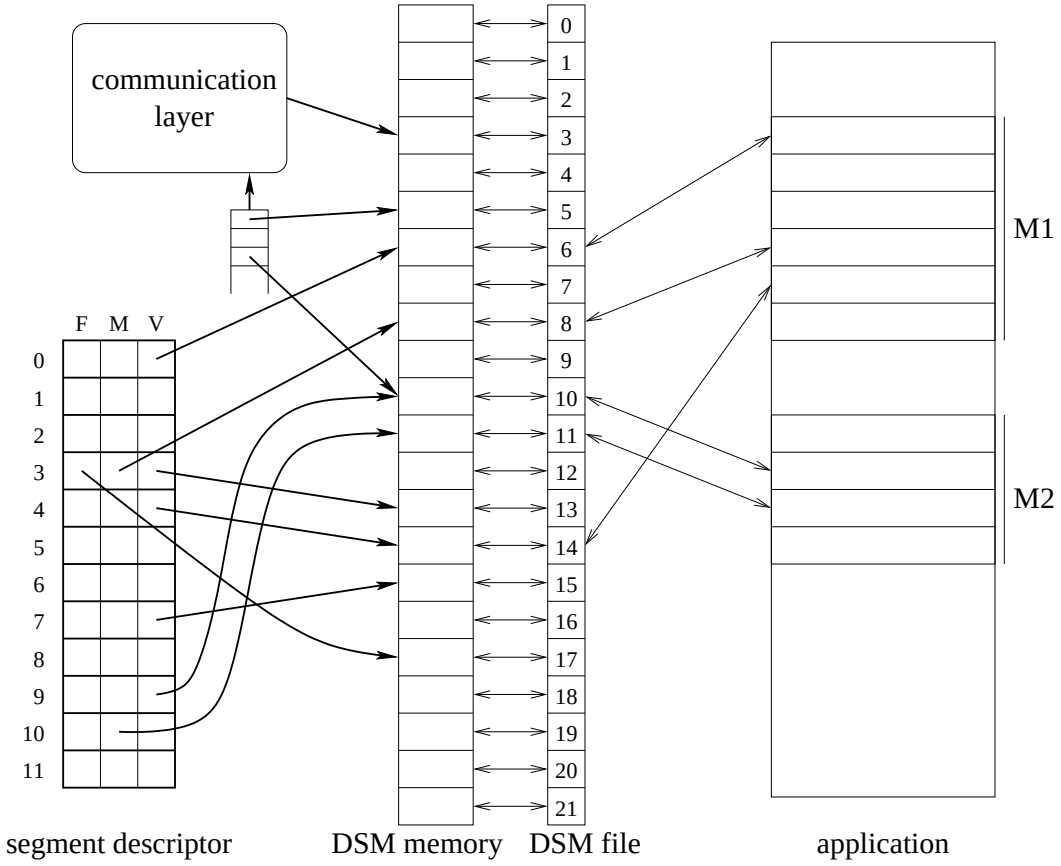

Figure 1: Mappings

- M points to the modified version,
- F points to the next version.

All three pointers are null when the page is not present. No data for segment pages 1, 2, 5, 6, 8 and 11 are present. The current version of pages 0, 3, 4, 7, and 9 are present in blocs 6, 13, 14, 15 and 10 respectively. Page 0 of the segment corresponds to page 0 of memory section M1 in the application. Because this page is the current page version, the access right of the application is limited to read right. This is also the case for page 4 of the segment which corresponds to page 4 of section M1 through a common mapping on block 14 of the DSM file. Page 2 of section M2 corresponds to the current version of page 9 of the segment. Pages 3 and 10 have a modified version in blocs 8 and 11. Segment page 3 corresponds to page 3 of section M1, and segment page 10 to page 2 of section M2. These two memory pages are writable by the application. When the page automaton receives a page with modification rights from the communication layer, the reference to this page is stored in pointer M. When a modification right is granted to the page automaton for an already present page by the global page manager, the reference to this page is first moved from V to M. Then, if the automaton was requested to keep a twin of the pages, this page is copied into a free page before the modification right is inherited by the application. The reference to this copy is stored into V.

The merging of modifications has been initiated on segment page 3: pointer F is linked to bloc 17. All the modifications already received by the node are stored into this block. The modifications can be integrated into this block upon arrival. A more efficient algorithm consists to keep all modified pages until all the modifications have been received and then to merge all the pages inside the same loop (see section 3.3). In the current state, bloc 17 does not yet record the local modifications which are still in block 8.

4.3 Address space management

The current implementation of Mome considers three forms of interactions between an application and the DSM: the *address space sharing* form, the *shared memory* form and the *distributed* form.

Using the *address space sharing* form, the DSM and the application run into the same address space (threads of the same Unix processes). The DSM threads can directly manipulate the access rights of the application using the `mmap` and `mprotect` system calls. The page automaton is synchronous: its state represents the exact state of the page in the application. The DSM and the application have the same life-time, the life-time of the process.

In the *shared memory* form, the DSM and the application run inside two different address spaces but share access to the DSM file through shared memory (single system image). The DSM has no direct control on the access rights of the application: they communicate through pipes and signals. The page automaton and the application communicate through messages and synchronize using acknowledgments. This form has been experimented on Mach micro-kernels using memory objects. It is currently the base for the implementation of a persistent data repository for parallel applications: programs can dynamically connect to the DSM, map existing segments on their memory, read and modify the data and leave this data in the repository for further use by other programs.

In the *distributed* form (currently under development), the page automaton asynchronously controls a *virtual node*. The virtual node integrates a Mome page manager in charge of a group of

computation node: the virtual node is itself a Mome DSM running on a cluster. Using these virtual nodes, it is possible to adapt the DSM structure to the structure of the applications and to the interconnection topology of the computation nodes. When two or more applications are coupled to the DSM, each virtual node is in charge of managing the page inside a single application. The top level of the hierarchy manages the interactions between the applications. Virtual nodes can also be used in order to reflect the topology of the computation nodes in the DSM topology: clusters of clusters, heterogeneous interconnection networks. The lower level groups locally or strongly connected nodes: the hierarchy favors local and more efficient transactions.

4.4 Communication layer

On each computation node, the DSM initiates one (or more, depending on the underlying communication layers) reception thread(s) in charge of receiving inter-node DSM messages. This thread dispatches the received messages into various waiting queues. There are two types of messages, the protocol messages (currently 16 bytes) and the page messages. A page message associates one protocol message to one memory page of data. The pages are sent directly from, or received into one block of the DSM memory: memory copies can be avoided between the low-level communication layer and the DSM layer. In Figure 1, the next page of data received by the communication layer will be stored directly into block 3 of the DSM memory. Each time a page has been received, a pointer to the received page is appended to the associated protocol message, and the message is dispatched on the relevant queue. The receive thread gets free pages from the free page list.

When a page is forwarded through the communication layer, a pointer to the corresponding DSM memory is appended to the corresponding protocol message. The message is then stored into the input queue of the communication system. In Figure 1, this input queue contains three messages. Messages 0 and 2 have off-line associated pages 5 and 10.

Note that the same block in the DSM memory can be referenced simultaneously by different page descriptors, by messages in the input queue of the communication layer or through mapping in the application space. Mome keeps track of all mappings using reference counters. In the case where some node receives modification rights on some page, the reference counter of this page is first checked. If other references exist, the application receives a fresh copy of the page (copy-on-write). A block of the DSM memory is linked to the free page list when all references have been removed.

4.5 DSM memory management

The memory foot-print of the Mome DSM on each node is limited by the size of the DSM file. It is possible to extend the DSM memory when the free page list becomes empty. However, the size of the physical memory of the computation nodes is limited, and allocating too large DSM memories can result in disk swapping. The DSM memory management thread is in charge of managing the usage of the DSM memory: it implements a classical house-keeping algorithm. Three occupation levels of the DSM memory are considered: green, yellow and red. In the green level, the memory management thread sleeps, no house-keeping is necessary. In the yellow level, the management thread and the application run in parallel. In the red zone, only a few free pages remain, the DSM stops resolving

page faults until enough space becomes free. The memory management thread randomly scans the DSM page descriptors. If the page descriptor contains a hidden page (see 5), it is immediately freed. If the page is accessible by the application, it is protected. If the page is already protected (the page has not been accessed since the last visit), the memory manager sends a non-blocking `free-page` (yellow zone) or `flush-page` (red zone) request to its page manager. The page-manager must imperatively free the page (using page migration if necessary) upon the reception of `flush-page`. The reaction to a `free-page` request is limited to the removal of duplicates.

5 Optimizations

The Mome DSM integrates features such as *hidden pages* to improve the memory management, *manager migration* to reduce the average distance between nodes and page managers, *prefetching* to reduce the latency of page-faults or *double-faulting* which limits the number of page-faults.

5.1 Hidden pages

When using the weak consistency level implemented in Mome, a DSM computation node is not automatically informed on the validity of its copy of the page: the node must send explicit consistency requests. But the DSM node must keep this copy in its memory as long as its presence is recorded in set V of the page manager. For the page manager point-of-view, set V records nodes which have requested notifications (strong consistency) and nodes which can participate to page forwarding. It is not necessary to have too many nodes in this set: only a few of them will be involved in page forwarding. A hidden page is a page present on a DSM node but ignored by the page manager. Such a page can be freely flushed by a memory manager. The page manager keeps track of the hidden pages using set H: each time the current version of the page is forwarded to some node, this node is added to set H. All requests sent by a DSM node to a page manager indicate if the last version of the page received by the node is still present. Such a page can be re-validated by the manager using set H in reply to a consistency request. The use of hidden pages improves the comportment of Mome: the memory manager can eject more pages without transaction with the page managers, hidden pages need not be invalidated.

5.2 Manager migration

The Mome DSM can exploit the data access analysis capacity of modern compilers through page manager migration. Without modifying the page management strategy, it is possible to reduce the transaction latency on the shared pages when the page directory is located on the node most frequently using the page. A node can request for the management of a page. Mome also implements the *ProbOwner* scheme of IVY [8]. Manager migration requests are handled without synchronizing the nodes. A typical use of manager migration is the application of the owner compute rule of HPF: a node requests the management of the pages containing the data it is in charge of.

5.3 Read or write prefetching, double faulting

Mome can also handle read and write prefetch requests on mapped memory sections. A prefetch request initiates a non-blocking page fault in the background. In double-faulting mode, all page faults are handled as write page faults. This mode reduces the number of page faults when shared variables are first read and then written.

6 Related works

Many software DSM have been developed in order to offer an execution model close to the simple and successful SMP model to programmers. TreadMarks [1] and JIAJIA [4] offer a programming interface close to the Pthread interface. Software codes built on this parallel programming model can be executed on these DSM with few modifications. The consistency of the shared data is managed automatically at run-time. As long as the accesses to the shared variables are race-free, the programs should behave correctly. The data consistency is managed at system page granularity.

The Mome DSM was first defined as a base for HPF run-time systems. The DSM inherited its first SPMD execution model from HPF. The explicit control of the data consistency was also derived from this data parallel model: HPF code generation is guided by static data access analysis at compile-time. The main advantage of this model is the total absence of page movement in the case of false sharing: as long as no data dependency has been detected by the compiler, modifications to the same page need not be merged. This is not the case for the release consistency model which automatically update the pages as soon as the nodes synchronize. This scheme works fine as long as the data dependency analysis can be applied with enough precision. When this is not possible, for instance in the presence of indirections in the data accesses, the compiler might request consistency management for large amounts of memory sections.

Mome always transfers whole pages between the nodes and does not try to reduce message lengths using diffs. In the current implementation, the throughput of the communication networks was not considered to be critical. The solution adopted for Mome avoids the need to always save a twin of a page before it can be modified and avoids encoding the diffs before the modifications are propagated. TreadMarks reduces the volume of data exchanged by the nodes, but at the cost of a more complex memory management and more CPU usage. A trade-off between processing power, memory usage and communication performance must be found. Mome clearly depends on the presence of a high performance communication system. On the other hand, because the consistency management must be controlled by the software, Mome uses a very simple protocol: all messages have a fixed length of 16 bytes. The release consistency model used in TreadMarks necessitates to exchange write notices when the nodes synchronize. These write notices increase the volume of communications and produce a more complex communication buffer management.

JIAJIA is a home-based DSM. Home-based protocols have the advantage that each access fault requires only one round trip to the home page. But, on the other hand, if many nodes try to access the same page, the home node is in charge of serving all requests. The directory scheme used by Mome allows to balance this load on many nodes. The difference is measurable, for instance, in the presence of broadcast schemes. On the other hand, Mome allows to dynamically migrate the page

directories, for instance on the node most frequently faulting on the page. If the data access patterns exhibit enough spatial locality, directory migrations remove inter-node communications and reduce the latency of fault resolution. For similar reasons JIAJIA allows for the migration of the page homes in order to reduce the communication needs on page updates ([3]).

7 Conclusion

In this paper, we presented the implementation of Mome, a user-level software DSM. Mome is a weak consistency multiple-writers DSM. Using Mome, a node must explicitly send a consistency request to the DSM each time its own view of a memory section must integrate modifications made by other nodes. It is possible for programmers to explicitly insert consistency management requests in their programs. Mome was first developed as a support for high level language run-time systems. The code generation optimizations applied by modern compilers are based on static data access analysis. From the same information, it is possible to characterize which array sections must be made consistent before some code section is executed. Consistency requests can be inserted by the compilers at code-generation time.

In a second step, the application domain of Mome has been extended to the coupling of parallel codes and for the implementation of a persistent data-repository.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] High Performance Fortran Forum, Department of Computer Science, Rice University. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[3] 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*, 1999.

[4] W. Hu, W. Shi, and Z. Tang. Reducing System Overheads in Home-based Software DSMs. In *Proc. of 13th Int'l Parallel Processing Symposium*, pages 167–173, April 1999.

[5] A. Itzkovitz and A. Schuster. Multiview and millipage fine-grain sharing in page-based dsms. In *3rd Symposium on Operating Systems Design and Implementation (OSDI99)*, pages 215–228, February 1999.

[6] Yvon Jégou. Controlling Distributed Shared Memory Consistency from High Level Programming Languages. In *Proceedings Parallel and Distributed Processing, IPDPS 2000 Workshops*, pages 293–300, May 2000.

[7] Yvon Jégou. Coupling DSM-Based Parallel Applications. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 82–89, May 2001.

[8] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399