Generic Distributed Shared Memory: the DSM-PM2 Approach

Gabriel Antoniu, Luc Bougé, Raymond Namyst

Laboratoire de l'Informatique du Parallélisme

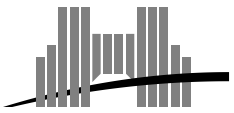
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

Generic Distributed Shared Memory: the DSM-PM2 Approach

Gabriel Antoniu
Luc Bougé
Raymond Namyst

March 2000

Research Report N° RR-2000-19



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Generic Distributed Shared Memory: the DSM-PM2 Approach

Gabriel Antoniu
Luc Bougé
Raymond Namyst

March 2000

Abstract

This paper describes DSM-PM2, a generic, multi-protocol distributed shared memory library built for PM2, a multithreaded runtime system with preemptive thread migration. DSM-PM2 allows threads running on different nodes to communicate via a virtually shared address space and supports multiple consistency models. For a given model, the user can select among several alternative implementations, based for instance on page migration, thread migration, or an adaptative combination of them. Moreover, new consistency models or new implementations of existing models can be easily added using the available library routines. DSM-PM2 is available on top of several UNIX systems and can use a large variety of network protocols (BIP, SCI, VIA, MPI, TCP, etc.). We report performance figures for platforms using different network protocols: SISCI/SCI, BIP/Myrinet and TCP/Myrinet.

Keywords: Generic DSM, Thread migration, Multi-protocol DSM, Iso-address migration, PM2, multithreading.

Résumé

Nous décrivons DSM-PM2, une librairie générique et multi-protocole pour la gestion de mémoire distribuée virtuellement partagée. Cette librairie a été concue et implantée pour PM2, un environnement multithread distribué qui propose la migration de processus légers. Grâce à DSM-PM2, des processus légers qui s'exécutent sur différents noeuds peuvent communiquer via un espace d'adressage commun. DSM-PM2 supporte plusieurs modèles de cohérence et, pour un modèle donné, plusieurs implantations alternatives, basées sur la migration de pages, sur la migration de processus légers, ou bien sur une combinaison des deux. De plus, de nouveaux modèles de cohérence peuvent être facilement rajoutés, ainsi que de nouvelles implantations des modèles existants. DSM-PM2 est disponible pour plusieurs systèmes de type UNIX et peut utiliser un grand nombre de protocoles reseaux (BIP, SCI, VIA, MPI, TCP, etc.). Nous présentons de mesures de performances pour 3 platformes: SISCI/SCI, BIP/Myrinet and TCP/Myrinet.

Mots-clés: MDVP générique, migration de processus léger, MDVP multi-multiprotocole, migration, migration iso-adresse, PM2, multithreading.

Generic Distributed Shared Memory: the DSM-PM2 Approach

Gabriel Antoniu* Luc Bougé* Raymond Namyst*

Contents

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: {Gabriel.Antoniu,Luc.Bouge,Raymond.Namyst}@ens-lyon.fr. This work has been supported by the INRIA ResCapA Research Coordinated Action, the NSF/INRIA *C*IT* Cooperative Research Grant, the CNRS ARP Research Program and the ReMaP Project, INRIA Rhône-Alpes.

1 Introduction

Distributed Shared Memory (DSM) libraries have been available for a dozen of years. In their traditional flavor [7, 11, 12], they allow a number of separate processes to share a common address space according to some fixed consistency model: sequential consistency, release consistency, etc. The processes may usually be physically distributed among a number of distant computing nodes interconnected by some communication library. The design of the DSM library is usually highly dependent on the selected consistency model and on the communication library.

Most existing DSM libraries are bound to one specific consistency model. Recently, it has been observed that the availability of multiple consistency models within the *same* library would allow the designer to tune applications more efficiently. Also, the interaction of the DSM library with the underlying communication library are often poorly structured. Adapting DSM libraries to new optimized communication libraries based on recent high-performance networks such as Myrinet or SCI has shown difficult. Finally, the advent of multithreading facilities in modern operating systems has raised the problem of the interaction between threads and DSM [8, 9].

This conjunction of factors has considerably hindered the development of *generic*, *portable* and *efficient* DSM libraries in the past years. Actually, attempting to enhance an existing DSM library with extra consistency protocols while merging it with an existing communication library on top of an existing thread library sounds like a nightmare... To our knowledge, no existing DSM system tackles all these three problems at once [1, 5, 8].

We believe that the right approach is to reconsider the problem from scratch with the primary goals of genericity, portability and efficiency in mind. Our *portable* and *efficient* distributed multithreaded environment called PM2 [10] (*Parallel Multithreaded Machine*) provides an interesting basis to address this challenge. We have therefore designed a new prototype DSM library called DSM-PM2, built on top of PM2. The DSM-PM2 library is designed so as to inherit from PM2 its wide spectrum of portability across platforms, and the efficiency of its thread management. Also, using the portable communication library of PM2 guarantees optimized performance across a variety of networking protocols. We have done one additional step by designing the library as a *generic* toolbox. It implements various consistency models, and even provides multiple implementations (thereafter called *protocols*) for the same model. A number of protocols are built-in. The user can easily add extra models or specific implementations of some existing model. This is an extremely valuable feature if the DSM library is to be targeted by compilers (Java, OpenMP, HPF, etc.): it is possible to easily tune the implementation so as to take advantage of the specific semantic features of the generated code.

Our work is related to that of DSM-Threads [8, 9], a system which extends POSIX multithreading to distributed environments by providing a multithreaded DSM. Our approach is different essentially by the generic support and the ability to support *new*, user-defined consistency protocols. Another system integrating threads with Distributed Shared Memory is Millipede [6]. While Millipede is designed for a single execution environment (Windows NT cluster with Myrinet) and focuses on sequential consistency, DSM-PM2 proposes multiple consistency models and protocols on top of several UNIX systems and can use a large variety of network protocols (BIP, SCI, VIA, MPI, TCP, etc.).

In this paper, we introduce our DSM-PM2 prototype library. Its general structure and programming interface are presented in Section 2. Section 3 discusses in more detail how to select/define protocols. Two implementations of the sequential consistency model are described, a first one based on *page* migration, and the second one using *thread* migration. Finally, we illustrate the portability and efficiency of DSM-PM2 by reporting its performance on top of different platforms using different

network protocols: SISCI/SCI, BIP/Myrinet and TCP/Myrinet.

2 DSM-PM2 overview

PM2 (Parallel Multithreaded Machine) [10] is a multithreaded environment for distributed architectures. Its programming interface is based on *Remote Procedure Calls* (RPC). Using RPC, the PM2 threads can invoke the remote execution of user defined *services*. Such invocations can either be handled by a pre-existing thread or they can involve the creation of a new thread. Threads running on the same node can freely share data. Threads running on distant nodes can only interact through RPC. This mechanism can be used either to send/retrieve information to/from the remote node, or to have some remote action executed. The latency of an LRPC is 6 μs on SISCI/SCI and 8 μs on BIP/Myrinet.

PM2 has two main components. For multithreading, it uses Marcel, an efficient, user-level, POSIX-like thread package. For communication, PM2 relies on Madeleine [4], an efficient and portable library implemented on top of high-performance protocols such as BIP, SCI, VIA, as well as on traditional protocols such as TCP, MPI and PVM.

PM2 provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such a functionality is typically useful to implement dynamic load balancing policies. In the original version of PM2 [10], no provision was made to maintain the relationship between a migrating thread and the data it is accessing: this was left to the designer, and only ad-hoc hook functions were provided. Coupling thread migration and data migration becomes a pretty complex issue in this context. The DSM-PM2 library addresses namely this point: it provides migrating threads with the illusion of a uniformly shared memory on top of a distributed architecture efficiently exploited by PM2.

Given that existing DSM applications require different consistency models, it makes little sense to dedicate a DSM library to a specific model. Therefore, DSM-PM2 has been designed to be generic and to support multiple consistency models. Moreover, thanks to the thread migration facility provided by PM2, DSM-PM2 can propose alternative mechanisms to implement a given model. The user can for instance choose to migrate the thread to the data it accesses, or to migrate the pages containing the accessed data to the thread, or any suitable combination of both. This is done by selecting the appropriate library protocols. The user can even dynamically switch from one policy to another. Moreover, new protocols can be built using existing library components.

2.1 General features

The main features characterizing DSM-PM2 are the following ones.

Multiple consistency models DSM-PM2 has been designed so as to support a number of consistency models. Sequential consistency and Java consistency are now operational, but other consistency models can be supported using the existing internal DSM mechanisms.

Multithreaded features DSM-PM2 has been designed so as to be compatible with PM2's programming interface, which allows threads to be created locally or remotely (via Remote Procedure Calls). Since all threads may access shared data, regardless of their location, all available protocols and their supportive data structures guarantee thread-safety. To ensure this property, classical protocols (such as Li and Hudak's dynamic distributed manager algorithm) had to be adapted to a multithreaded execution context. On the other hand, thread-related features

provided by PM2 (such as thread migration) have served as a basis for the design of the consistency protocols themselves.

Generic multi-protocol support We call *consistency protocol* a possible implementation of a given *consistency model*. DSM-PM2 proposes alternative mechanisms to implement the same consistency model. For instance, we currently provide two built-in protocols implementing sequential consistency: the first one uses *page replication* and *page migration*, whereas the second one relies on *thread migration* exclusively. The user can explicitly associate protocols with shared data.

Adaptive protocols Users may also choose hybrid approaches by building new protocols out of existing library routines. Protocols may mix page replication, page migration and thread migration. Moreover, one may build protocols able to dynamically switch from one policy to another depending on external factors such as load information or access histories.

Access detection DSM-PM2 provides a multithreaded DSM interface: static and dynamic data can be shared by all the threads in the system. Since the programming interface is intended both for direct use and as a target for compilers, no pre-processing is assumed in the general case and accesses to shared data are detected using page faults. Nevertheless, an application can choose to bypass the fault detection mechanism by controlling the accesses through calls to `get`/`put` primitives which explicitly handle consistency. In some cases, the resulting cost may be smaller than the overhead of the underlying fault handling subsystem. DSM-PM2 copes with this approach as well, as illustrated by the implementation of the Java consistency model discussed in Section 3.3.

Portability DSM-PM2 requires no specific system support, is fully operational on top of Linux and Solaris and is portable over a large set of communication networks and communication protocols (such as BIP, SCI, VIA, MPI, TCP) thanks to Madeleine, PM2's communication library. The implementation contains only a few OS/architecture-dependent instructions for segmentation fault information (address, fault type, etc.).

Efficiency DSM-PM2 adds only a very small overhead (10-15%) to the cost of the communication operations required to take consistency actions (which gives a lower bound for the overall cost). This is illustrated and discussed in Section 4 with figures for 3 different platforms. Hence we can claim DSM-PM2 is well-suited as a target DSM runtime system for high-performance applications.

2.2 Architecture

DSM-PM2 is structured in layers (Figure 1). At the high level, a *DSM protocol policy* layer is responsible for building consistency protocols out of a subset of the available library routines and for associating each application data with its own consistency protocol. The library routines (used to bring a copy of a remote page to a thread, to migrate a thread to a remote page, to invalidate all copies of a page, etc.) are grouped in the lower-level *DSM protocol library* layer. Finally, these library routines are built on top of two base components: the *DSM page manager* and the *DSM communication module*. The *DSM page manager* is essentially dedicated to the low-level management of memory pages. It implements a distributed table containing page ownership information and maintains the appropriate access rights on each node. This table can be exploited to implement protocols which need a fixed page manager, as well as protocols based on a dynamic manager [7].

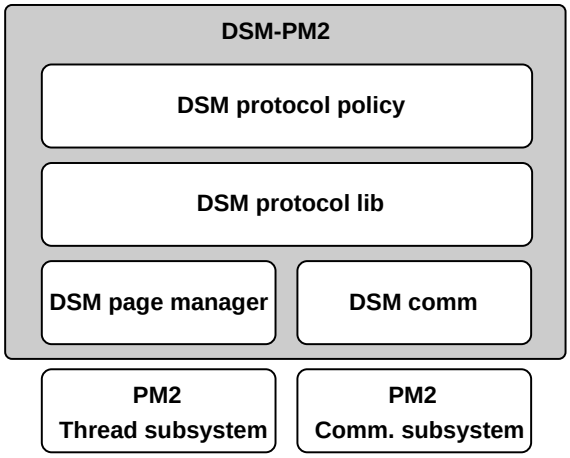


Figure 1: Overview of the DSM-PM2 software architecture

The *DSM communication module* is responsible for providing elementary communication mechanisms, such as delivering requests for page copies, sending pages, invalidating pages or sending diffs. This module has been implemented using PM2's RPC mechanism, which meets very well our needs. For instance, requesting a copy of a remote page for read access can essentially be seen as invoking a remote service. On the other hand, since the RPC are implemented on top of the Madeleine communication library, the DSM-PM2 communication module is portable across all networks supported by Madeleine (SCI, BIP, MPI, TCP, PVM) with no extra cost.

2.3 Using DSM-PM2: the programming interface

Figure 2 shows a sample DSM-PM2 code. The program uses a single shared variable (c) which is shared by all threads on all nodes. Static shared data are declared within a special section, bracketed by the macros BEGIN_DSM_DATA and END_DSM_DATA. The program is SPMD and is intended to run on two nodes. First, the default DSM protocol is selected (in this case, a protocol implementing sequential consistency using a variant of Li and Hudak's dynamic distributed manager algorithm [7], adapted by Mueller [8, 9] for a multithreaded context). Note that we use a library protocol in this case, but custom protocols can be used, too, as explained in Section 3.

On each node, the function my_func() is invoked (through a direct call on node 0 and through a RPC on node 1). The function increments 1,000,000 times the shared variable c, whose final value diplayed by node 0 will be 2,000,000. As a more complex code exemplifying multithreading, we could create multiple threads on each node, each of them calling my_func() and incrementing c. For simplicity, we do not illustrate this feature here.

3 Specifying protocols

A protocol is a set of actions destined to guarantee consistency according to a consistency model. In our current implementation, a protocol is specified through six functions that are automatically called by the generic DSM support, as described in Table 1. These functions are grouped in a structure (dsm_protocol_t) which needs to be passed as an argument to the DSM protocol initialization primitive: void pm2_init_dsm_protocol(dsm_protocol_t *prot).

Protocol function	Description
read_fault_handler	Called on a read page fault
write_fault_handler	Called on a write page fault
read_server	Called on receiving a request for read access
write_server	Called on receiving a request for write access
invalidate_server	Called on receiving a request for invalidation
receive_page_server	Called on receiving a page

Table 1: DSM-PM2 protocol actions

Protocol	Description
dsmlib_ddm_li_hudak_prot	Implements sequential consistency using page replication on read access and page migration on write access. Relies on a variant of the dynamic distributed manager MRSW algorithm described by Li and Hudak [7], adapted by Muller [8, 9].
dsmlib_migrate_thread_prot	Implements sequential consistency using thread migration on both read and write faults.
dsmlib_hyp_java_prot	Implements Java consistency. Non-standard MRMW protocol, based on explicit checks for locality, not on page faults.

Table 2: DSM-PM2 library protocols

There are three ways to specify protocols:

Using library protocols Users can select one of the pre-initialized dsm_protocol_t structures available in DSM-PM2. Their specific features are summarized in Table 2.

Building protocols using library routines Users can choose to build protocols by defining dsm_protocol_t structures grouping library routines in approaches not provided by the library protocols. One may thus choose hybrid approaches such as page replication on read fault and thread migration on write fault. One may even embed a dynamic mechanism selection within the protocol, switching for instance from page migration to thread migration depending on ad-hoc criteria.

Building protocols using DSM-PM2 lower-level blocks Finally, users may build their own protocols using the programming interface of the lower-level DSM-PM2 libraries: dsm_page_manager for page management and dsm_comm for communication. Of course, this approach is more difficult and more complex. The user is responsible for using these libraries in a consistent way to produce a valid protocol.

In the following two subsections we describe our two built-in protocols for sequential consistency.

3.1 Sequential consistency using a classical approach

Li and Hudak [7] propose several algorithms for sequential consistency (fixed centralized manager, fixed distributed manager, dynamic distributed manager, etc.). Though all these protocols could be implemented using our two base modules (*DSM page manager* and *DSM communication module*),

```
#include "dsm-pm2.h"
#define N 1000000

BEGIN_DSM_DATA
int c = 0;
END_DSM_DATA

dsm_mutex_t L;                  /* Global mutex */
int DSM_SERVICE;                /* service id */
pm2_completion_t end;           /* completion object */

void my_func () {
    int i;
    for (i = 0; i < N; i++) {
          dsm_mutex_lock (&L);
          c++;
          dsm_mutex_unlock (&L);
      }
}

void main () {
    /* Select the DSM protocol: */
    pm2_set_dsm_protocol (&dsm_lib_li_hudak_prot);

    /* Register a user-level RPC: when DSM_SERVICE
       is invoked on a  remote node,
       the function my_func is called: */
    pm2_rpc_register (&DSM_SERVICE, my_func);

    /* Initialize the global mutex: */
    dsm_mutex_init (&L, NULL);

    /* Call PM2's init procedure */
    pm2_init ();

    /* The following code is executed on node 0 only */
    if (pm2_self () == 0) {
          /* Invoke DSM_SERVICE on node 1. This will result is a call
             to my_func on node 1. The completion is then signalled
             to node 0 using the object "end" */
          pm2_rpc (1, DSM_SERVICE, &end);

          /* Call my_func on node 0 */
          my_func ();

          /* Wait for completion on node 1 */
          pm2_wait (&end);
          printf("Final value = %d\n", c);
          pm2_halt ();
       }
}
```

Figure 2: Sample DSM-PM2 program.

we have chosen to implement the third one only, which shows good properties as regards scalability. This algorithm has been adapted by Mueller [8, 9] in his implementation of DSM-Threads, in order to take into account the extra complexity due to multithreading. As opposed to the simple case presented by Li and Hudak, where all page faults on a node are processed sequentially, concurrent requests may be processed in parallel in a multithreaded context, should they concern the same page or different pages.

The algorithm uses page replication on read and page migration (after invalidation of all copies) on write. We illustrate page migration on Figure 3. It is carried out when a thread needs to write to a page and does not have the appropriate access right. The access is detected through a page fault and the thread executes the handler corresponding to the fault type (i.e., the second function in the protocol structure, for a write access). The function consults the page table and requests the page and its ownership to the supposed owner of the page (here, Node 1). Node 1 may no longer be the owner of the page and in this case it forwards the request to the owner pointed by its page table (i.e., Node 2). On receiving the request, Node 2 (which we assume to be the current owner) creates a thread which calls the write server (4th function in the protocol structure), which invalidates all page copies (here, only the local copy) and sends the page directly to Node 0, together with the page ownership. On receiving the page, Node 0 creates a thread which executes the receive page server (last function in the protocol structure): it installs the page, sets the appropriate access rights in the page table, and eventually wakes up the waiting thread. Note that other threads may have tried to access the page in the meantime, but in this case they have been blocked, waiting for the request to be served. Therefore, all the threads waiting for the page are waked up when the page arrives. On the other hand, multiple page requests may be concurrently issued by threads running on the same node if they concern different pages.

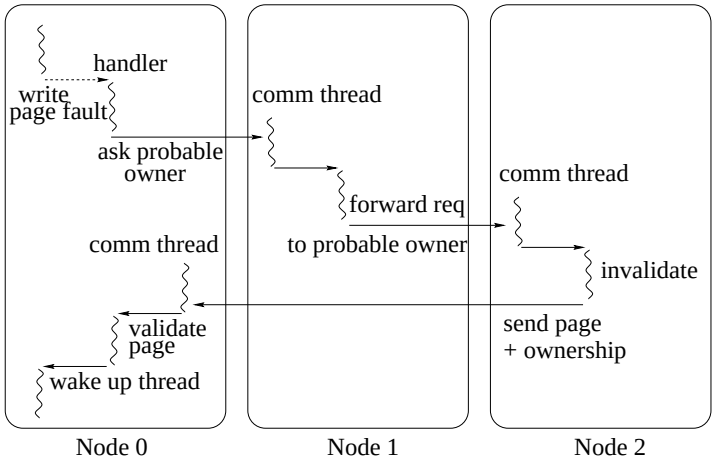


Figure 3: Sequential consistency (classical solution): on a write page fault, the page and its ownership are transfered to the requesting node

3.2 Sequential consistency using thread migration

For sequential consistency, we propose an alternative protocol based on thread migration. The protocol structure is quite simple in this case, it only references PM2's migrate_thread function in the first two fields of the protocol structure (i.e., for read fault and write fault), whereas the other fields are set to NULL. When a thread accesses a page and does not have the appropriate access

rights, it executes the page fault handler which simply migrates the thread to the node owning the page (as specified by the local page table). Observe that it is a remarkable feature of PM2 to be able to migrate a thread which is just executing a fault handler! On reaching the destination node, the thread exits the handler and repeats the access. If the access rights are appropriately set on this node, the access is successful and the thread continues its execution. Otherwise, the repeated access results in a new segmentation fault and the the thread is then migrated to the owner pointed by the local page table. Note the simplicity of this protocol, which essentially relies on a single function: the powerful thread migration primitive provided by PM2.

Let us also emphasize that the protocol works well thanks to our iso-address approach to data allocation. DSM data is located at the same virtual address on all nodes and, consequently, on exiting the fault handler after migration, the thread automatically repeats the access at the *same* address, which does correspond to the same piece of data.

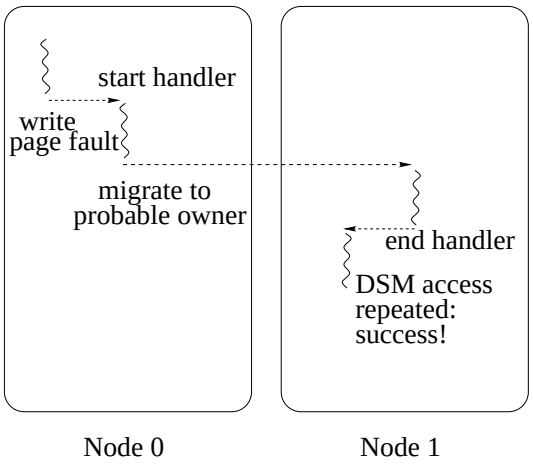


Figure 4: Sequential consistency using thread migration: on page fault, the thread migrates to the node where the data is located

3.3 Using DSM-PM2 as a runtime for compiled Java

As explained above, the DSM-PM2's API for protocol specification consists of a set of functions automatically called by the generic DSM mechanism. This approach is well suited for implementations using page faults to detect accesses to shared data, like our two library protocols implementing sequential consistency. We have also implemented a protocol for Java consistency [3] to demonstrate the genericity of the internal structure of DSM-PM2. This Java protocol relies on explicit checks for data location. Accesses to shared data are controlled using specific get/put primitives and consistency actions are partly taken automatically (on receiving a page request or on receiving a page), partly using explicit calls to some extra protocol primitives (loadIntoCache, updateMainMemory, flushCache) built on top of our low-level DSM-PM2 libraries: dsm_page_manager and dsm_comm. This has allowed us to use DSM-PM2 as a target for the Hyperion [2] distributed Java system. See [3] for a detailed description of our protocol for Java consistency.

4 Performance evaluation

We have evaluated the raw performance of our basic protocol primitives on three different platforms. The first column in Table 3 and in Table 4 corresponds to measurements carried out on a cluster of 450 MHz Pentium II nodes running Linux 2.2.10 interconnected by a SCI network using the SISCI protocol. The figures in the next two columns have been obtained on a cluster of 200 MHz Pentium Pro nodes running Linux 2.2.13 interconnected by a Myrinet network using the BIP and TCP protocols respectively. (Unfortunately, we could not use two platforms with the same processors.)

Table 3 reports the time (in μs) taken by each step involved when a read fault occurs on a node, assuming that the associated implementation of the protocol is *page transfer* based. First, the execution of the faulting instruction raises a signal (*page fault*). This signal is caught by a handler that inspects the page table to locate the real owner on the page and protects the associated table entry from being further accessed by other threads (*fault handling*). Then, a request message is sent (*request transmission*) to this owner. Once its arrival is detected on the owner node, the request is processed (*request service*) and the asked page is sent back to the other node (*page transmission*). Finally, the page is installed, the appropriate access rights are set on the corresponding address area and the faulting instruction is restarted (*page installation*).

Note that we had to use two distinct *ping-pong* test programs in order to measure the exact respective transfer times for page request and page transfer.

Operation	PII 450 MHz SISCI/SCI	PPro 200 MHz BIP/Myrinet	PPro 200 MHz TCP/Myrinet
Page fault	18	56	56
Fault handling	1	2	2
Request transmission	17	30	190
Request service	1	2	2
Page transmission	85	134	412
Page installation	12	24	24
Total (μs)	**134**	**248**	**686**

Table 3: Performance analysis of the processing of a read-fault under page-migration policy.

As one can observe, the processing overhead of DSM-PM2 with respect to the raw transmission time is only 10–15%. The overhead related to page installation includes a call to the mprotect primitive to set the *write access right* on the page and a call to malloc to allocate the page bitmap necessary for recording local modifications. This latter cost could be further improved using a custom malloc-like primitive.

In Table 4 we report the cost (in μs) for processing a read fault assuming a *thread migration* based implementation of the consistency protocol. As one may guess, the *page fault* processing time is the same as with the page-transfer based implementation. However, the *fault handling* time is now insignificant, because the associated processing is just an access to the page table, to determine the owner of the page. The last step merely consists of a call to the underlying runtime to migrate the thread to the owner node. That's all!

We can observe that this migration-based implementation outperforms the previous one, because thread migration is very efficient. Note however, that this migration time is closely related to the stack size of the thread, because it has to be entirely migrated on the remote node along with the thread descriptor. In our test program, the thread's stack was very small (about 1 kB), which is

Operation	PII 450 MHz SISCI/SCI	PPro 200 MHz BIP/Myrinet	PPro 200 MHz TCP/Myrinet
Page fault	18	56	56
Fault handling	-	-	-
Thread migration	24	75	257
Total (μs)	**42**	**131**	**313**

Table 4: Performance analysis of the processing of a read-fault under thread-migration policy.

typically the case in many applications, but not in all applications. Thus, choosing between the implementation based on page transfer and the one based on thread migration deserves careful attention. Moreover, it may depend on other criteria such as the number and the location of the threads accessing the same page, etc. This is a research topic we plan to investigate in the future.

5 Conclusion

We have introduced our DSM-PM2 prototype, a DSM library for our PM2 distributed, multithreaded runtime environment. DSM-PM2 provides an integrated framework to handle migrating threads sharing a uniform memory space across a network of distributed nodes. DSM-PM2 inherits the portability and efficiency of PM2. It is available on a large number of platforms and interconnection protocols, from standard, general-purpose ones (PVM, MPI) to high-performance, specific ones (SCI, Myrinet, VIA, etc.). Instead of being dedicated to one consistency model, DSM-PM2 is able to handle multiple consistency models, and even multiple implementations of a single model.

The distinguished feature of DSM-PM2 is its ability to support *user-defined* consistency protocols through an internal programming interface. A number of predefined protocols are available, and the user can freely redefine and/or combine them. Most interestingly, this allows the user to provide alternative implementations for some existing models, which takes advantage of the specific semantic features of the application. For instance, sequential consistency may be implemented by migrating pages to threads, *or* by migrating threads to pages, *or* by dynamically selecting on each access fault which of the alternative is the best one. Some existing implementation may be moreover enhanced by pre-fetching pages according to the behavior of the application, or by modifying on the fly the ownership of the pages. DSM-PM2 sets no limits to the inventiveness of the user! Its interface has proved generic enough to support the Java consistency model.

Currently, DSM-PM2 is operational on Linux and Solaris and our preliminary tests on top of SISCI/SCI, TCP/Myrinet and BIP/Myrinet have been successful. The three protocols mentioned in Table 2 are available and hybrid protocols mixing thread migration and page replication can also be built out of library functions. We are currently working on the dynamic memory allocator and on the distributed synchronization mechanisms.

We intend to further develop our generic low-level components by providing support for other consistency models and protocols (such as release consistency). Using the remote read/write features provided by the SCI communication interface also seems a promising topic to investigate. Finally, we plan to evaluate DSM-PM2 using the SPLASH-2 [13] benchmarks.

Acknowledgments

We thank Frank Mueller for his helpful explanations about the implementation of DSM-Threads. We also thank Phil Hatcher for our fruitful collaboration regarding Java consistency.

References

[1] 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.

[2] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. Research Report RR2000-08, LIP, ENS Lyon, Lyon, France, February 2000. Submitted for publication.

[3] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Implementing Java consistency using a generic, multithreaded DSM runtime system. In *Parallel and Distributed Processing. Proc. Intl Workshop on Java for Parallel and Distributed Computing.*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Held in conjunction with IPDPS 2000, Springer-Verlag. To appear.

[4] L. Bougé, J-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. Held in conjunction with IPPS/SPDP 1999, Springer-Verlag.

[5] A. Itzkovitz and A. Schuster. MultiView and Millipage – fine-grain sharing in page-based DSMs. In *Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 215–228, February 1999.

[6] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its application in distributed shared memory systems. *J. Systems and Software*, 42(1):71–87, July 1998.

[7] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] F. Mueller. Distributed shared-memory threads: DSM-Threads. In *Proc. Workshop on Run-Time Systems for Parallel Programming (RTSPP)*, pages 31–40, Geneva, Switzerland, April 1997. Held in conjonction with IPPS '97.

[9] F. Mueller. On the design and implementation of DSM-Threads. In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 315–324, June 1997.

[10] R. Namyst. *PM2: an environment for a portable design and an efficient execution of irregular parallel applications.* Doctoral thesis, Univ. Lille 1, France, January 1997. In French.

[11] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE computer*, 24(8):52–60, September 1991.

[12] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Paralel and Distributed Technology*, pages 63–79, 1996.

[13] S. C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd Annual Int'l Symp. on Comp. Arch.*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.