

DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols

Gabriel Antoniu, Luc Bougé

▶ **To cite this version:**

Gabriel Antoniu, Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. Proc. 6th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01), Apr 2001, San Francisco, United States. pp.55-70, 10.1109/IPDPS.2001.925077 . inria-00563583

HAL Id: inria-00563583

https://inria.hal.science/inria-00563583

Submitted on 6 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols

Gabriel Antoniu* and Luc Bougé*

Abstract

DSM-PM2 is a platform for designing, implementing and experimenting multithreaded DSM consistency protocols. It provides a generic toolbox which facilitates protocol design and allows for easy experimentation with alternative protocols for a given consistency model. DSM-PM2 is portable across a wide range of clusters. We illustrate its power with figures obtained for different protocols implementing sequential consistency, release consistency and Java consistency, on top of Myrinet, Fast-Ethernet and SCI clusters.

1 Introduction

In their traditional flavor, Distributed Shared Memory (DSM) libraries [7, 10, 11, 4] 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 computing nodes interconnected by some communication library. The design of the DSM library is often highly dependent on the selected consistency model and on the communication library, and only a few of them are able to exploit the power of modern thread libraries.

In most approaches to DSM programming, it is considered that the DSM library and the underlying architecture are fixed, and that it is up to the programmer to fit his program with them. We think that such a *static* vision fails to appreciate the possibilities of this area of programming. We believe that a better approach is to provide the application programmer with an *implementation platform* where *both* the application *and* the multithreaded DSM consistency protocol can possibly be *co-designed* and tuned for performance. This aspect is crucial if the platform is used as target for a compiler: the implementation of the consistency protocol can then directly benefit from the specific properties of the generated code. The platform should moreover be *portable*, so that the programmer do not have to commit to some existing communication library or operating system.

DSM-PM2 is a prototype implementation platform for multithreaded DSM programming which attempts to meet these requirements. Its general structure and programming interface are presented in Section 2. Section 3 discusses in more detail how to select, define or optimize protocols. We give an overview of the implementation of a number of protocols for various consistency models, including sequential, release and Java consistency. In particular, two implementations of the sequential consistency model are described, a first one based on *page* migration, and the second one using *thread* migration, as enabled by the underlying multithreading library. Finally, we illustrate the portability and efficiency of DSM-PM2 by reporting performance measurements on top of different cluster architectures using various network protocols: BIP/Myrinet, TCP/Myrinet, TCP/FastEthernet, SISCI/SCI.

Our work is related to that of DSM-Threads [8], 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 [5]. Whereas 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.).

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: Gabriel.Antoniu@ens-lyon.fr

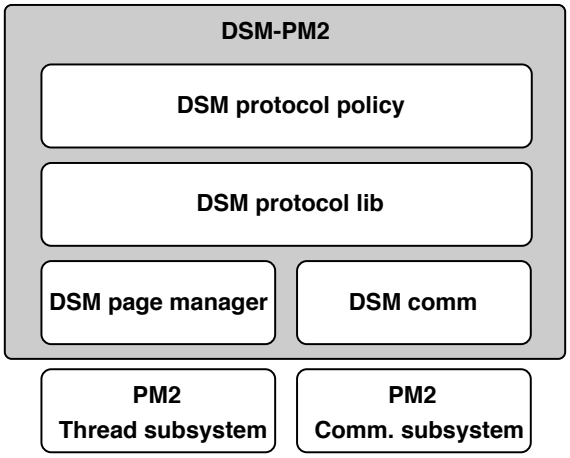


Figure 1. Overview of the DSM-PM2 software architecture

2 DSM-PM2: an overview

DSM-PM2 is a portable implementation platform for multithreaded DSM consistency protocols. It is build on top of the PM2 distributed, multithreaded runtime system (Parallel Multithreaded Machine, [9]). PM2 provides a POSIX-like interface to create, manipulate and synchronize lightweight threads in user space, in a distributed environment. It is available on top of most major Unix brands, including Solaris and Linux. To ensure network portability, PM2 uses an efficient communication library called Madeleine [3], which was ported accros a wide range of communication interfaces, including high-performance ones such as BIP/Myrinet, SISCI/SCI, VIA, as well as more traditional ones such as TCP, and MPI. DSM/PM2 inherits this wide portability, since all DSM communication primitives have been implemented using PM2's RPC mechanism based on Madeleine. An interesting feature of PM2 is its *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. DSM-PM2 provides such mobile threads with the illusion of a uniformly shared memory on top of the distributed architecture of PM2.

DSM-PM2 is structured in layers (Figure 1). At the highest level, a *DSM protocol policy* layer is responsible for building consistency protocols out of a subset of the available library routines. An arbitrary number of protocols can be defined at this level, which may be selected by the application through a specific library call. Some *classical* protocols are already built-in, as summarized in Table 1. The user can add a new protocol by defining each of its component routines and by registering it using a specific library call (Figure 2). The newly created protocol could then be used as default protocol or specified when shared objects are dynamically allocated. Thus, different protocols can be assigned to different data. Observe also that there is no restriction on the semantics of the protocols: several implementations of the same consistency model may co-exist within the same program, the one to use on a given run being dynamically specified by the application without any re-compiling.

The component routines of a protocol are defined using a *generic* toolbox called the *DSM protocol library* layer. It provides routines to perform elementary actions such as bringing a copy of a remote page to a thread, migrating a thread to some remote data, invalidating all copies of a page, etc. All available routines guarantee thread-safety. This library is 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 page manager.

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 is implemented using PM2's RPC mechanism, which turns out to be well-suited for this kind of task. 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 RPCs are implemented on top of the Madeleine communication library, the DSM-PM2 communication module is portable across all networks supported by Madeleine with no extra cost.

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 to shared data through calls to get/put primitives which explicitly handle consistency. DSM-PM2 copes with this approach as well and both manners were used in alternative implementa-

Protocol	Consistency	Basic features
li_hudak	Sequential	MRSW protocol. Page replication on read access, page migration on write access. Dynamic distributed manager.
migrate_thread	Sequential	Uses thread migration on both read and write faults. Fixed distributed manager.
erc_sw	Release	MRSW protocol implementing *eager* release consistency. Dynamic distributed manager.
hbrc_mw	Release	MRMW protocol implementing *home-based lazy* release consistency. Fixed distributed manager. Uses twins and on-release diffing.
java_ic	Java	Non-standard, home-based MRMW protocol, based on explicit checks for locality. Fixed distributed manager. Uses on-the-fly diff recording.
java_pf	Java	Home-based MRMW protocol, based on on page faults. Fixed distributed manager. Uses on-the-fly diff recording.

Table 1. Consistency protocols currently available in the DSM-PM2 library.

tions of the Java consistency model, allowing DSM-PM2 to be used as a target of the Hyperion Java compiling system for distributed clusters [2].

3 Specifying protocols in DSM-PM2

A protocol is a set of actions designed to guarantee consistency according to a consistency model. In our current implementation, a protocol is specified through eight functions that are automatically called by the generic DSM support. In its current stage of development, DSM-PM2 provides 6 built-in protocols, whose main characteristics are summarized in Table 1.

All these protocols share an important comon feature: the implementation is *multi-threaded* (i.e. uses multiple "hidden" threads) and *thread-safe*: an arbitrary number of user-level threads can concurrently access pages on any node and threads on the same node safely share the same page copy. This distinctive features required that the traditional consistency algorithms (usually written for single-threaded systems) which we used as a starting point be adapted to a multi-threaded context to handle thread-level concurrency. As opposed to the traditional algorithms 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.

Sequential consistency. We provide two protocols for sequential consistency. li_hudak relies on a variant of the dynamic distributed manager MRSW algorithm described by Li and Hudak [7], adapted by Muller [8]. It uses page replication on read fault and page migration on write fault.

Alternatively, DSM-PM2 provides an original protocol for sequential consistency, based on thread migration (migrate_thread). 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). On reaching the destination node, the thread exits the handler and repeats the access, which is now successfully carried out and the thread continues its execution. Note the simplicity of this protocol, which essentially relies on a single function: the thread migration primitive provided by PM2.

This protocol crucially depends on an *iso-address* approach to data allocation: DSM pages are mapped at the *same* virtual address on all nodes. 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. Thread stacks are equally allocated using the same scheme, which thus guarantees the validity of all pointers in the presence of migration [1].

Release consistency. DSM-PM2 also provides two alternative implementations for release consistency. erc_sw is a MRSW protocol for eager realease consistency. It uses page replication on read fault and page migration on write fault, based on the same dynamic distributed manager scheme as li_hudak. Page ownership migrates along with the write access rights and pages in the copyset get invalidated on lock release.

Alternatively, hbrc_mw is a home-based protocol allowing multiple writers (MRMW protocol) thanks to the classical twinning technique. Basically, each page has a home node, where all threads have write

access. On page fault, a copy of the page is brought from the home node and a twin copy gets created. On release, page diffs are computed and sent to the home node, which subsequently invalidates third-party writer nodes. On receiving such an invalidation, these latter nodes need to early compute and send their own diffs (if any) to the home node.

Java consistency. DSM-PM2 provides two protocols which directly implement the specification of the Java Memory Model. Thanks to these protocols, DSM-PM2 is currenly used by the Hyperion Java compiling system [2] and consequently supports the execution of compiled threaded Java programs on clusters. Our DSM-PM2 protocols were co-designed with Hyperion's memory module and this approach enabled us to make aggressive optimizations using information from the upper layers. For instance, a number of synchronizations could thus be optimized out.

The Java Memory Model [6] allows threads to keep locally cached copies of objects. Consistency is provided by requiring that a thread's object cache be flushed upon entry to a monitor and that local modifications made to cached objects be transmitted to the central memory when a thread exits a monitor. The concept of *main memory* is implemented via a *home-based* approach. The home node is in charge of managing the reference copy. Objects (initially stored on their home nodes) are replicated if accessed on other nodes. Note that at most one copy of an object may exist on a node and this copy is shared by all the threads running on that node. Thus, we avoid wasting memory by associating caches to nodes rather than to threads.

Since Hyperion uses specific access primitives to shared data (get and put), we can use explicit checks to detect if an object is present (i.e., has a copy) on the local node, thus by-passing the page-fault mechanism. If the object is present, it is directly accessed, else the page containing the object is locally cached. This scheme is used by the java_ic protocol. Alternatively the java_pf protocol uses page faults to detect accesses to non-local objects. Thanks to the put access primitives, the modifications can be recorded at the moment when they are carried out, with object-field granularity. All local modifications are sent to the home node of the page by the main memory update primitive, called by the Hyperion run-time on exiting a monitor.

Defining new protocols. New protocols can be added by giving the eight component routines, as illustrated in Figure 2. These could include user-defined routines, but also library routines used otherwise then in the built-in protocols. For example, one may 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. However, the user is responsible for using these features in a consistent way to produce a valid protocol.

```
void main ()
{
/* Define a new protocol by specifying
   the component routines */

 int prot;
 ...
 prot = dsm_create_protocol
  (read_fault_handler, write_fault_handler,
   read_server, write_server,
   invalidate_server, receive_page_server,
   acquire_handler, release_handler);

 /* Set this protocol as default. */
 pm2_dsm_set_default_protocol(prot);
 ...
}
```

Figure 2. Defining a new protocol in DSM-PM2.

4 Performance evaluation

We present the raw performance of our basic protocol primitives on three different platforms. The measurements were carried out 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 and by a Fast Ethernet network under TCP[1] .

Table 2 reports the time (in μs) taken by each step involved when a read fault occurs on a node, assuming that the corresponding protocol is *page transfer* based (which is the case for all built-in protocols, except for migrate_thread). First, the faulting instruction leads to a signal (*page fault*), which is caught by a handler that inspects the page table to locate the page owner and then requests the page to this owner (request page). Once its arrival is detected on the owner node, the request is processed and the asked page is sent to the requester (*page transfer*). The *protocol overhead* includes the request processing time on the

[1]The final version will equally provide figures for an SISCI/SCI platform with PII 450 MHz nodes

owner node and the page installation on the requesting node.

Operation	BIP/ Myrinet	TCP/ Myrinet	TCP/ Fast Ethernet
Page fault	21	21	21
Request page	46	325	426
Page transfer	215	460	3809
Protocol overhead	50	50	50
Total (μs)	**332**	**856**	**4306**

Table 2. Processing a read-fault under page-migration policy: performance analysis

As one can observe, the protocol overhead of DSM-PM2 is only up to 15% of the total access time, as most of the time is spent with communications. The *protocol overhead* basically consists in updating page table information and setting the appropriate access rights.

In Table 3 we report the cost (in μs) for processing a read fault assuming a *thread migration* based implementation of the consistency protocol. The *protocol overhead* is here insignificant (less than 1 μs), since it 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 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, and may be closely related to the load balance, as illustrated below. This is a research topic we plan to investigate in the future.

Operation	BIP/ Myrinet	TCP/ Myrinet	TCP/ Fast Ethernet
Page fault	21	21	21
Thread migration	75	438	1035
Protocol overhead	-	-	-
Total (μs)	**96**	**459**	**1056**

Table 3. Processing a read-fault under thread-migration policy: performance analysis

To illustrate DSM-PM2's ability to serve as an experimental platform for comparing consistency protocols, we have run a program solving the *Travalling Salesman Problem* for 14 cities, with random inter-city distances, using one application thread per node. Figure 3 presents run times for our 4 protocols implementing seqential and release consistency, on the BIP/Myrinet platform. Given that the only shared variable intensively accessed in this program is the current shortest path and that the accesses to this variable are always lock protected, the benefits of release consistency over sequential consistency are not illustrated here. But we can still remark that all protocols based on page migration perform better than the protocol using thread migration. This is essentially due to the fact that all computing threads migrate to the node holding the shared variable, which thus gets overloaded. We could expect a better behavior for this protocol with applications where shared data are evenly distributed accross nodes and uniformly accessed.

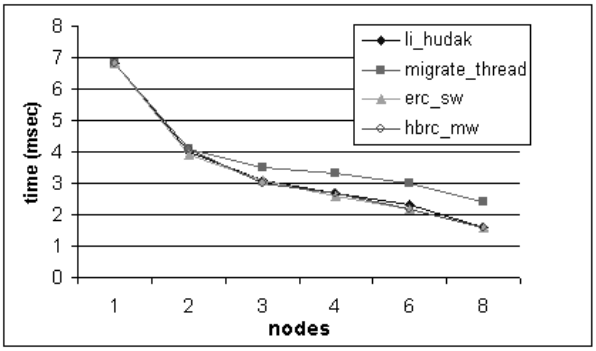


Figure 3. Solving TSP for 14 cities with random inter-city distances. Comparison of 4 DSM protocols

To compare our two protocols for Java consistency, we have run a multi-threaded Java program implementing a branch-and-bound solution to the minimal-cost map-coloring problem, compiled with Hyperion [2]. The program was run on a four-node cluster of 450 MHz Pentium II processors running Linux 2.2.13, interconnected by a SCI network using the SISCI protocol and solves the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Figure 4 clearly shows that the protocol using access detection based on page faults (java_pf) outperforms the protocol based on in-line checks for locality (java_ic). This is due to the intensive use of objects in the program: remember that every get and put operation involves a check for local-

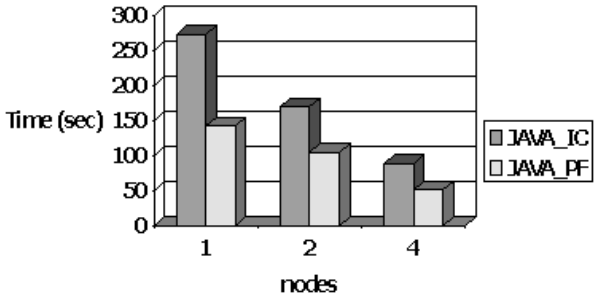


Figure 4. Comparing the two protocols for Java consistency: page faults vs. in-line checks.

ity in java_ic, wheras this is not the case for accesses to local objects when using java_pf. The overhead of fault handling appears to be significantly less important than the overhead due to checks, also thanks to a good distribution of the objects: local objects are intensively used, remote accesses (generating faults for java_pf) are not very frequent.

Finally, we mention that very precise post-mortem monitoring tools are available in the PM2 platform, providing the user with valuable information on the time spent within each elementary function. This feature proves very helpful for understanding and improving protocol performance.

5 Conclusion

DSM-PM2 is a platform for designing, implementing and experimenting multithreaded DSM consistency protocols. It provides a generic toolbox which facilitates protocol design and allows for experimentating with alternative protocols for a given consistency model. DSM-PM2 is portable across a wide range of cluster architectures, using high-performance interconnection networks such as BIP/Myrinet, SISCI/SCI, VIA, as well as more traditional ones such as TCP, and MPI. In this paper, we have illustrated its power by presenting different protocols implementing sequential consistency, release consistency and Java consistency, on top of different cluster architectures using various network protocols: BIP/Myrinet, TCP/Myrinet, TCP/FastEthernet, SISCI/SCI.

DSM-PM2 is *not* just yet another multithreaded DSM library. It is aimed at exploring a new research direction, namely providing the designers of such protocols with *portable plateforms* to experiment alternative designs, in an environment as open, controlled and monitored as possible. We are convinced that many interesting ideas in DSM protocols have not yet been studied simply because of the lack of such *an open plateform*: implementing everything from scratch is simply too hard! Also, such a plateform enables *competing protocol designers* to compare their protocols within a common environment, using common profiling tools. Actually, switching from one protocol to another, or switching from one communication library to another, can be done without changing anything to the application. No re-compiling is even needed if all the necessary routines have been linked beforehand. Finally, such a plateform opens a large access to the area of *co-design*: indeed, the application and the protocol can then be designed and optimized *together*, instead of simply tuning the application on top of a fixed, existing protocol. This idea seems of particular interest in the case of compilers targeting DSM libraries, as demonstrated by the Hyperion Java compiler project reported above.

Currently, DSM-PM2 is operational on Linux and Solaris. Extensive testing has been done on top of SISCI/SCI, TCP/Myrinet and BIP/Myrinet. All the protocols mentioned in Table 1 are available and hybrid protocols mixing thread migration and page replication can also be built out of library functions. We are currently working on a deeper performance evaluation using the SPLASH-2 [12] benchmarks.

References

[1] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 496–510, San Juan, Puerto Rico, April 1999. Springer-Verlag.

[2] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. Compiling multithreaded Java bytecode for distributed execution. In *Euro-Par 2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1039–1052, Munchen, Germany, August 2000. Springer-Verlag.

[3] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *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. Springer-Verlag.

[4] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, 87(3), March 1999.

[5] 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.

[6] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java language specification, second edition*. Addison Wesley, 2000.

[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.

[9] 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.

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

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

[12] 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.