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

Gabriel Antoniu and Luc Bougé

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

Abstract. DSM-PM2 is a platform for designing, implementing and experimenting with 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 [15, 18, 19, 10] allow a number of separate processes to share a common address space using a *consistency protocol* according to a semantics specified by some given *consistency model*: sequential consistency, release consistency, etc. The processes may usually be physically distributed among a number of computing nodes interconnected through some communication library. The design of the DSM library is often highly dependent on the selected consistency model and on the communication library. Also, only a few of them are able to exploit the power of modern thread libraries to provide multithreaded protocols, or at least to provide thread-safe versions of the consistency protocols.

Most approaches to DSM programming assume 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 model through a specific protocol can then directly benefit from the specific properties of the code, enforced by the compiler in the code generation process. The platform should moreover be *portable*, so that the programmers do not have to commit to some existing communication library or operating system, or at least be able to postpone this decision as late as possible.

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 and define protocols. A given consistency model can be implemented via multiple alternative protocols. We give an overview of the implementation of several protocols for various consistency models, including sequential consistency, release consistency and Java consistency (which is a variant of release consistency). In particular, two alternative protocols addressing 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 communication interfaces and interconnection networks: BIP [20]/Myrinet, TCP/Myrinet, TCP/FastEthernet, SISCI/SCI [9].

Related work

The concept of Distributed Shared Memory was proposed more than a decade ago [15]. Important efforts have been subsequently made to improve the performance of software DSM systems and many such systems were proposed to illustrate new ideas. Progresses related to the relaxation of consistency protocols were illustrated with Munin [6] (for release consistency), TreadMarks [1] (to study the impact of laziness in coherence propagation, through lazy release consistency), Midway [4] (for entry consistency), and Brazos [21] (for scope consistency). Recent software DSM systems, such as Millipede [11], CVM [13] and Brazos integrate the use of multithreading.

Our work is more closely related to that of DSM-Threads [16], 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. Millipede [11] also integrates threads with Distributed Shared Memory. It has been designed for a specific execution environment (Windows NT cluster with Myrinet) and focuses on sequential consistency only. CVM [13] is another software DSM system which provides multithreading (essentially to hide the network latency) and supports multiple consistency models and protocols. However, CVM's communication layer targets the UDP protocol only, whereas DSM-PM2 captures the benefits of PM2's portability on a large variety of communication interfaces: it is currently available on modern Myrinet and SCI high-performance clusters run with Linux. The primary goal of DSM-PM2 is to provide a portable platform for easy protocol experimentation. Its customizability makes it also valuable as a target for compilers as the Java Hyperion compiler discussed in Section 3.3.

2 DSM-PM2: an overview

2.1 The PM2 runtime system

PM2 (Parallel Multithreaded Machine) [17] is a multithreaded environment for distributed architectures. It provides a POSIX-like interface to create, manipulate and synchronize lightweight threads in user space, in a distributed environment. Its basic mechanism for inter-node interaction is the *Remote Procedure Call* (RPC). Using RPCs, 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. While threads running on the same node can freely share data, PM2 threads running on distant nodes may 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 minimal latency of a RPC is 6 μs over SISCI/SCI and 8 μs over BIP/Myrinet on our local Linux clusters.

PM2 includes two main components. For multithreading, it uses Marcel, an efficient, user-level, POSIX-like thread package. To ensure network portability, PM2 uses an efficient communication library called Madeleine [5], which was ported across a wide range of communication interfaces, including high-performance ones such as BIP [20], SISCI, VIA [7], as well as more traditional ones such as TCP, and MPI.

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. Such a functionality is typically useful to implement generic policies for dynamic load balancing, independently of the applications: the load of each processing node can be evaluated according to some measure, and balanced using preemptive migration. The key feature enabling preemptiveness is the *iso-address* approach to dynamic allocation featured by PM2. The isomalloc allocation routine guarantees that the range of virtual addresses allocated by a thread on a node will be left free on any other node. Thus, threads can be safely migrated across nodes: their stacks and their dynamically allocated data are just copied on the destination node at the *same* virtual address as on the original node. This guarantees the validity of all pointers without any further restriction [3]. Migrating a thread with a minimal stack and no attached data, takes 62 μs over SISCI/SCI and 75 μs over BIP/Myrinet on our local Linux clusters.

2.2 DSM-PM2: towards a portable implementation platform

DSM-PM2 provides the illusion of a common address space shared by all PM2 threads irrespective of their location and thus implements the concept of Distributed Shared Memory on top of the distributed architecture of PM2. But DSM-PM2 is not simply *a* DSM layer for PM2: its goal is to provide a portable implementation platform for multithreaded DSM consistency protocols. Given that all DSM communication primitives have been implemented using PM2's RPC mechanism based on Madeleine, DSM-PM2 inherits PM2's wide network portability. However, the most important feature of DSM-PM2 is its *customizability*: actually, the main design goal was to provide support for implementing, tuning and comparing several consistency models, and alternative protocols for a given consistency model.

As a starting remark, we can notice that all DSM systems share a number of common features. Every DSM system, aimed for instance at illustrating a new version of some protocol, has to implement again a number of core functionalities. It is therefore interesting to ask: What are the features that need to be present in *any* DSM system? And then: What are the features that are specific to a *particular* DSM system? By answering these questions, we become able to build a system where the core mechanisms shared by the existing DSM systems are provided as a *generic*, common layer, on top of which specific protocols can be easily built. In our study, we limit ourselves to *page-based* DSM systems.

Access detection. Most DSM systems use page faults to detect accesses to shared data, in order to carry out actions necessary to guarantee consistency. The generic core should provide routines to detect page faults, to extract information related to each fault (address, fault type, etc.) and to associate protocol-specific consistency actions to a page-fault event.

Page manager. Page-based DSM systems use a page table which stores information about the shared pages. Each memory page is handled individually. Some information fields are common to virtually all protocols: local access rights, current owner, etc. Other fields may be specific to some protocol. The generic core should provide the page table structure and a basic set of functions to manipulate page entries. Also, the page table structure should be designed so that new information fields could be added, as needed by the protocols of interest.

DSM communication. We can notice that the known DSM protocols use a limited set of communication routines, like sending a page request, sending a page, sending diffs (for some protocols implementing weak consistency models, like release consistency). Such a set of routines should also be part of the generic core.

Synchronization and consistency. Weaker consistency models, like release, entry, or scope consistency require that consistency actions be taken at synchronization points. In order to support these models, the generic core should provide synchronization objects (locks, barriers, etc.) and enable consistency actions to be associated to synchronization events.

Thread-safety. Modern environments for parallel programming use multithreading. All the data structures and management routines provided by the generic core should be *thread-safe*: multiple concurrent threads should be able to safely call these routines.

A closer study of page-based consistency protocols enables to list up a small number of events which should trigger consistency actions: page faults, receipt of a page request, receipt of the requested page, receipt of an invalidation request. Additionally, for weak consistency models, lock acquire, lock release and barrier calls are events to be associated with consistency actions. In the current version of DSM-PM2, there are 8 actions. The detailed list is given in Table 1.

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
lock_acquire	Called after having acquired a lock
lock_release	Called before releasing a lock

Table 1. DSM-PM2 protocol actions.

Once the generic core has been delineated, we can consider building consistency protocols on top of it. Designing a protocol in DSM-PM2 consists in providing a set of 8 routines, one for each action identified above. These routines are designed using on the API of the generic components. They are automatically called by DSM-PM2, and nothing more has to be done by the programmer. According to our personal experience, the code for the routines is quite manageable: a few hundreds of lines for the whole set of routines of a typical protocol. A key feature of DSM-PM2 is that all the mechanisms provided by the generic core are *thread-safe*. The task of the protocol designer is thus considerably alleviated as most (if not all!) subtle synchronization problems are already addressed by the core routines.

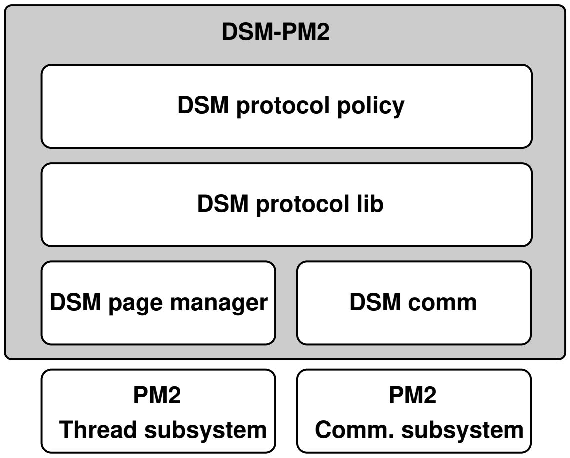

Fig. 1. Overview of the DSM-PM2 software architecture.

As a consequence of our distinction between generic core mechanisms and protocol-specific actions, DSM-PM2 is structured in layers (Figure 1). At the lowest level, DSM-PM2 includes two main components which make up the the main part of the generic core: the *DSM page manager* and the *DSM communication module*. Both are based on the API of PM2: no direct access to the thread low-level structures and to the underlying communication library are made.

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 has been designed to be generic enough so that it could be exploited to implement protocols which need a fixed page manager, as well as protocols based on a dynamic page manager (see [15] for a classification of page managers). Of course, each protocol uses the fields in the page entries of the table as required by its corresponding page management strategy (which is decided at the higher, *protocol library* level). Consequently, a field may have different semantics in different protocols and may be even left unused by some protocols. Also, new fields could be easily added if needed in the future.

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 communication interfaces supported by Madeleine at no extra cost.

The routines which compose a protocol are defined using a 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 the available routines are thread-safe. This library is built on top of the two base components of the generic core: the *DSM page manager* and the *DSM communication module*.

Finally, 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 2, but the user can also add new protocols, as described in Section 2.3 by defining each of the component routines of each protocol and by registering it using specific library calls.

2.3 Using protocols

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	Home-based MRMW protocol, based on explicit *inline checks* (ic)for locality. Fixed distributed manager. Uses on-the-fly diff recording.
java_pf	Java	Home-based MRMW protocol, based on on *page faults*(pf). Fixed distributed manager. Uses on-the-fly diff recording.

Table 2. Consistency protocols currently available in the DSM-PM2 library.

```
#include "pm2.h"

BEGIN_DSM_DATA
int x = 34;
/* ... */
END_DSM_DATA

void main (void)
{
  /* Use the built-in 'li_hudak' protocol */

  pm2_dsm_set_default_protocol(li_hudak);
  pm2_init();

  x++;

  /* ... */
}
```

Fig. 2. Using DSM-PM2 with a built-in protocol.

In DSM-PM2, a specific protocol is a set of actions designed to guarantee consistency according to a consistency model. In our current implementation, a protocol is specified through 8 routines (listed in Table 1) that are automatically called by the generic DSM support as needed. Each protocol is labeled by a unique identifier. This identifier can for instance be used to set it up as the default protocol or to associate it to dynamically allocated shared objects. DSM-PM2 protocols can be specified and used in three different ways.

Using built-in protocols. The easiest way consists in selecting one of the available built-in protocols. In its current stage of development, DSM-PM2 provides 6 such protocols, whose main characteristics are summarized in Table 2 and detailed in Section 3. On Figure 2, the li_hudak protocol is declared as the default protocol for the static shared area.

Building new protocols. The user can also a define a new protocol by providing each of its component routines and by registering it using a specific library call. The newly created protocol can then be used exactly in the same way as built-in protocols.

```
int new_proto;
new_prot = dsm_create_protocol
 (read_fault_handler, write_fault_handler,
  read_server, write_server,
  invalidate_server, receive_page_server,
  acquire_handler, release_handler);

pm2_dsm_set_default_protocol(proto);
```

Building protocols using library routines A mixed approach consists in using *existing* library routines, as provided in the *DSM protocol library* layer, rather than new, user-defined routines, but combine then in some ad-hoc way. One may thus consider *hybrid* approaches such as page replication on read fault (like in the li_hudak protocol) and thread migration on write fault (like in the migrate_thread protocol). 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.

Observe that no pre-processing of the code file is used. Consequently, it is possible to define a number of protocols in a program and to dynamically select one of them according to the arguments provided by the user without any recompilation:

```
int proto1, proto2;
proto1 = dsm_create_protocol(...);
proto2 = dsm_create_protocol(...);

if (...)  pm2_dsm_set_default_protocol(proto1);
else pm2_dsm_set_default_protocol(proto2);
```

Again, the built-in protocols are just pre-defined protocols, so they can freely be included in such a selection.

On Figure 2, a protocol is associated to a static memory area. DSM-PM2 also provides dynamic allocation for shared memory. Each such dynamically-allocated shared area can be managed with a specific protocol, which can be specified through its creation attribute as illustrated. (Otherwise, the default protocol set by pm2_dsm_set_default_protocol is used.) Consequently, different DSM protocols may be associated to different DSM memory areas within the same application.

```
#define N 128
int *ptr;
dsm_attr_t attr;

dsm_attr_set_protocol(&attr, li_hudak);
ptr = (int*)dsm_malloc(N*sizeof(int),&attr);
```

In the current version of the system, DSM-PM2 does not provide any specific support to dynamically *switch* the management of a memory area from one protocol to another one within the same run. However, this can be achieved if needed through a careful synchronization at the program level (e.g. through barriers). Essentially, one has to keep the corresponding memory area from being accessed by the application threads during the protocol switch, since this operation involves modifications in the distributed page table on all nodes.

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, when DSM-PM2 is used as a compiler target, accesses to shared data may be carried out through specific runtime primitives like get and put (and not through direct assignment). The implementation of these primitives may then explicitly check for data locality and handle consistency accordingly. DSM-PM2 thus provides a way to bypass the page fault detection and to directly activate the protocol actions.

3 Built-in protocols available with DSM-PM2

Currently, DSM-PM2 provides 6 built-in protocols, whose main characteristics are summarized in Table 2. All these protocols share two important common features. 1) Their implementations are *multithreaded*: it uses multiple "hidden" threads to maintain the internal data structures and to enhance reactivity to external events such as message arrival. 2) They are *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. These distinctive features required that the traditional consistency protocols (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 protocols 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.

3.1 Sequential consistency

We provide two protocols for sequential consistency. The li_hudak protocol relies on a variant of the dynamic distributed manager MRSW (multiple reader, single writer) algorithm described by Li and Hudak [15], adapted by Mueller [16]. It uses page replication on read fault and page migration on write fault. Note that in a multithreaded context, the *single* writer refers to a node, not to a thread, since all the threads on the 'writer' node share the same copy. They may thus write it concurrently.

Alternatively, DSM-PM2 provides a new protocol for sequential consistency based on thread migration (migrate_thread), illustrated in Figure 3. 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. The counterpart is that the pages are not replicated in this protocol (i.e., for each page, there is a unique node where the page can be accessed both for read and write), so that all threads accessing a non local page will migrate to the corresponding owning node. Though the migration cost is generally very low, the efficiency of this protocol is highly influenced by the distribution of the shared data, which has a direct impact on the load balancing (since the threads migrate to the data they access). This point is discussed in Section 4.

The protocol described above crucially depends on an *iso-address* approach to data allocation [3]: not only static, but also dynamically allocated DSM pages are mapped at the *same* virtual address on all nodes, using the isomalloc allocation routine of PM2.

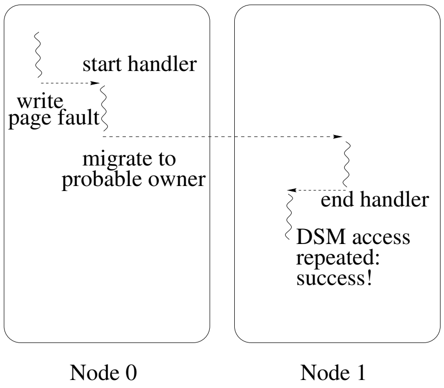

Fig. 3. Sequential consistency using thread migration: on page fault, the thread migrates to the node where the data is located.

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.

3.2 Release consistency

DSM-PM2 also provides two alternative implementations for release consistency. The erc_sw protocol is a MRSW protocol for eager release 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. Pages in the copyset get invalidated on lock release.

Alternatively, the hbrc_mw protocol is a home-based protocol allowing multiple writers (MRMW protocol) by using the 'classical' twinning technique described in [14]. Essentially, 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 compute and send their own diffs (if any) to the home node.

3.3 Java consistency

DSM-PM2 provides two protocols which directly implement consistency as specified by the Java Memory Model [12] (we refer to this consistency using the term "Java consistency"). Thanks to these protocols, DSM-PM2 is currently 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 thereby be optimized out.

The Java Memory Model 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. Gontmakher and Schuster [8] have shown that the JMM provides Release Consistency for synchronized access to non-volatile variables and stricter forms of consistency for the other cases. That is, Java Consistency is equivalent to Release Consistency in most cases.

The concept of *main memory* is implemented with DSM-PM2 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 brought to the local cache. This scheme is used by the java_ic protocol (where ic stands for inline check). Alternatively the java_pf protocol uses page faults to detect accesses to non-local objects (hence the pf suffix). Through 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.

4 Performance evaluation

We present the raw performance of our basic protocol primitives on four different platforms. The measurements were first carried out on a cluster of 450 MHz PII 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. Then, the same measurements were realized on a cluster of PII 450 MHz nodes interconnected by a SCI network.

Table 3 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). The request is processed on the owner node and the required page is sent to the requester (*page transfer*). The time reported here corresponds to a common 4 kB page. Finally, the *protocol overhead* includes the request processing time on the owner node and the page installation on the requesting node.

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 communication. The *protocol overhead* essentially consists in updating page table information and setting the appropriate access rights.

Operation	BIP/Myrinet	TCP/Myrinet	TCP/Fast Ethernet	SISCI/SCI
Page fault	11	11	11	11
Request page	23	220	220	38
Page transfer	138	343	736	119
Protocol overhead	26	26	26	26
Total (μs)	**198**	**600**	**993**	**194**

Table 3. Processing a read-fault under page-migration policy: Performance analysis.

In Table 4 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 run-time to migrate the thread to the owner node. In PM2, migrating a thread means moving the thread stack and the thread descriptor to the destination node, possibly together with some private dynamically allocated data (which is not the case in this example).

Operation	BIP/Myrinet	TCP/Myrinet	TCP/Fast Ethernet	SISCI/SCI
Page fault	11	11	11	11
Thread migration	75	280	373	62
Protocol overhead	1	1	1	1
Total (μs)	**87**	**292**	**385**	**74**

Table 4. Processing a read-fault under thread-migration policy: Performance analysis.

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. 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.

To illustrate DSM-PM2's ability to serve as an experimental platform for comparing consistency protocols, we have run a program solving the *Traveling Salesman Problem* for 14 randomly placed cities, using one application thread per node. Figure 4 presents run times for our 4 protocols implementing sequential 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

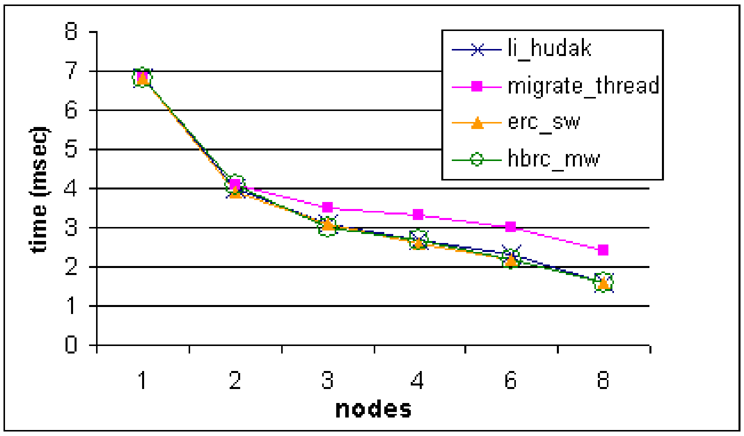

Fig. 4. Solving TSP for 14 cities with random inter-city distances: Comparison of 4 DSM protocols.

gets overloaded. We could expect a better behavior for this protocol with applications where shared data are evenly distributed across nodes and uniformly accessed.

To compare our two protocols for Java consistency, we have run a multithreaded 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 API and solves the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Figure 5 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 locality in java_ic, whereas 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.

Of course, we are aware that the performance evaluation reported above can only be considered as preliminary. A more complete analysis is necessary to study the behavior of the DSM-PM2 protocols with respect to different classes of applications illustrating various sharing patterns, access patterns, synchronization methods, etc. This is part of our current work.

Finally, we can 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.

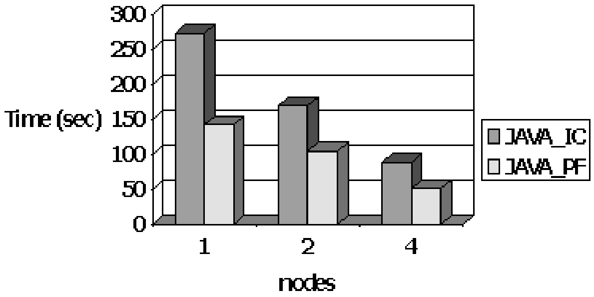

Fig. 5. Comparing the two protocols for Java consistency: page faults vs. in-line checks.

5 Conclusion

DSM-PM2 is a platform for designing, implementing and experimenting with multithreaded DSM consistency protocols. It provides a generic toolbox which facilitates protocol design and allows for experimentation 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: 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 platforms* to experiment with alternative designs, in a generic, customizable environment, while providing tools for performance profiling, such as post-mortem analysis. We are convinced that many interesting ideas in DSM protocols could be more easily experimented using such *an open platform*: implementing everything from scratch is simply too hard! Also, such a platform enables *competing protocol designers* to compare their protocols within a common environment, using common profiling tools. Switching from one protocol to another, or switching from one communication library to another, can be done without changing anything to the application. No recompiling is even needed if all the necessary routines have been linked beforehand. Finally, such a platform 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 2.2.x and Solaris 6 or later. Extensive testing has been done on top of SISCI/SCI, TCP/Myrinet and BIP/Myrinet. All the 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 a more thorough performance evaluation using the SPLASH-2 [22] benchmarks, which will be helpful to guide an efficient protocol use in applications.

Acknowledgments

We are grateful to Frank Mueller for his helpful explanations about the design of the DSM-Threads system. We thank Phil Hatcher for our fruitful collaboration on Java consistency. Last but not least, we thank Vincent Bernardi for his help with the design and implementation of the two protocols for release consistency within DSM-PM2.

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. In *Euro-Par 2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1039–1052, Munchen, Germany, August 2000. Springer-Verlag.
3. 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.
4. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.
5. 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 468–482, San Juan, Puerto Rico, April 1999. Springer-Verlag.
6. J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29:219–227, 1995. Special issue on distributed shared memory.
7. Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne-Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–75, March 1998.
8. A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizatons for Java memory behavior. In *Proc. of the Workshop on Java for High-Performance Computing*, Rhodes, June 1999.
9. IEEE. *Standard for Scalable Coherent Interface (SCI)*, August 1993. Standard no. 1596.
10. L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, 87(3), March 1999.
11. 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.
12. B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java language specification*. Addison Wesley, Second edition, 2000.
13. P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th Intl. Conf. on Distributed Computing Systems*, Hong Kong, May 1998.

14. P. Keleher, A.L.Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software based release consistent protocols. *J. Parallel and Distrib. Comp.*, 26(2):126–141, September 1995.
15. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
16. 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.
17. R. Namyst. *PM2: an environment for a portable design and an efficient execution of irregular parallel applications*. PhD thesis, Univ. Lille 1, France, January 1997. In French.
18. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE computer*, 24(8):52–60, September 1991.
19. J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Paralel and Distributed Technology*, pages 63–79, 1996.
20. Loïc Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485. Springer-Verlag, April 1998.
21. E. Speight and J.K. Bennett. Brazos: A third generation DSM system. In *Proc. of the USENIX Windows/NT Workshop*, pages 95–106, August 1997.
22. 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.