Implementing Multithreaded Protocols for Release Consistency on Top of the Generic DSM-PM2 Platform

Gabriel Antoniu, Luc Bougé

Implementing multithreaded protocols for release consistency on top of the generic DSM-PM^2 platform

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-PM^2 is an implementation platform designed to facilitate the experimental studies with consistency protocoles for distributed shared memory. This platform provides basic building blocks, allowing for an easy design, implementation and evaluation of a large variety of *multithreaded* consistency protocols within a unified framework. DSM-PM^2 is portable over a large variety of cluster architectures, using various communication interfaces (TCP, MPI, BIP, SCI, VIA, etc.). This paper presents the design of two multithreaded protocols implementing the release consistency model. We evaluate the impact of these consistency protocols on the overall performance of a typical distributed application, for two clusters with different interconnection networks and communication interfaces.

1 Introduction

Traditionally, Distributed Shared Memory (DSM) libraries [7, 10, 11, 6] allow a number of separate, distributed processes to share a global address space, based on a *consistency protocol* which implements the 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 and to provide multithreaded protocols, or at least to provide thread-safe versions of the consistency protocols.

The main objective of the DSM-PM^2 project is to provide the programmer of *distributed, multithreaded applications* with a flexible and portable implementation platform where the application and the consistency protocol of the underlying DSM can be *co-designed* and tuned together for performance. The programmer can select the consistency protocol best suited for his application, or can even program his own consistency protocol using basic blocks provided by the platform. Multiple consistency models are supported and protocols can be easily implemented for these models through alternative mechanisms available in DSM-PM^2. Comparative experimentations can be carried out on different cluster architectures, since the platform is portable across most UNIX-like systems and supports a large number of communication interfaces: TCP, MPI, BIP [12], SCI [5], VIA [4], etc. Finally, an important feature is that the platform

is operational in a multithreaded context: distributed threads can safely access shared data and concurrency-related problems are adressed directly by the consistency protocols.

In this paper we present the design of the multithreaded versions of two consistency protocols for the *release consistency* model. We describe their integration to DSM-PM^2 and we compare their performance using a multithreaded version of a FFT kernel from the SPLASH-2 [13] benchmark suite.

2 The DSM-PM^2 platform

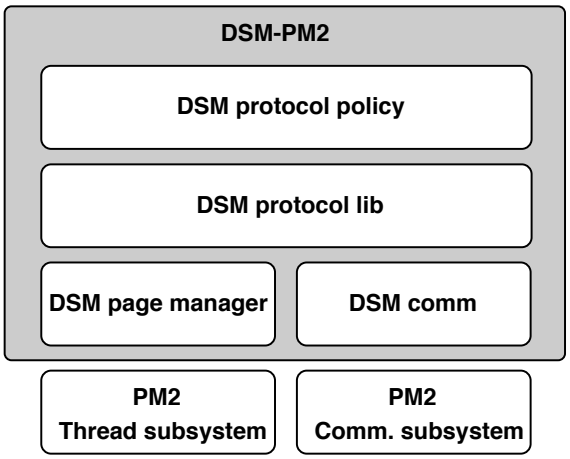

Fig. 1. The architecture of DSM-PM^2.

DSM-PM^2 [1] has been designed as an experimental implementation platform for multithreaded DSM consistency protocols. It relies on the PM^2 (*Parallel Multithreaded Machine*, [9]), a runtime system for distributed, multithreaded applications. PM2 provides a POSIX-like programming interface for thread creation, manipulation and synchronization in user space, on cluster architectures. PM2 is available on most UNIX-like operating systems, including Linux and Solaris. For network portability, PM2 uses a communication library called Madeleine [2], which has been ported on top of a large number of communication interfaces: high-performance interfaces, like SISCI/SCI and VIA, but also more traditional interfaces, like TCP and MPI. DSM-PM^2 inherits this portability, since all its communication routines rely on Madeleine. An interesting feature of PM^2 is that it allows threads to be preemptively and transparently migrated across the cluster nodes. DSM-PM^2 provides these mobile threads with the abstraction of a uniformly shared memory on top of the distributed architecture exploited by PM^2.

DSM-PM^2 is structured in layers. At the top level, the *DSM protocol policy* layer allows consistency protocols to be built out of common library routines, provided by

the *DSM protocol library* layer. These routines rely on DSM-PM^2's main generic components: the *DSM page manager* and the *DSM communication manager*. The former handles the access rights to the shared pages on each node, using a distributed page table. The latter provides common communication routines, like sending a page request, sending a page, sending an invalidation request, etc. These generic components have been designed in order to facilitate the implementation of new consistency protocols. The main feature of all elements of the architecture is their design for a use in a multithreaded context.

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
init	Called on initializing a protocol

Table 1. Nine routines which define a DSM protocol in DSM-PM2.

DSM-PM^2 currently provides 6 protocols. The application can select one of them via a function call at the initialization of the application:

```
pm2_dsm_set_default_protocol(li_hudak);
```

The user can also create new protocols by defining 9 functions which specify the behavior of the DSM system for some generic DSM events. These functions are listed in Table 1. The 9 functions are grouped into a protocol via a simple function call:

```
int new_prot;
new_prot = dsm_create_protocol
 (read_fault_handler, write_fault_handler,
  read_server, write_server,
  invalidate_server, receive_page_server,
  acquire_handler, release_handler, init);
pm2_dsm_set_default_protocol(new_prot);
```

3 Two *multithreaded* protocols for release consistency

Historically, DSM systems have first used *sequential consistency*. Li and Hudak [7] have proposed several MRSW protocols (*Multiple Readers, Single Writer*) for this

model, based on page replication to serve read accesses and page migration to serve write accesses. One of these protocols, relying on a dynamic distributed page manager [7], has been adapted by Mueller [8] to a multithreaded context. A variant of this protocol is available in DSM-PM^2 under the name li_hudak.

Relaxed consistency models, such as *release* consistency have been introduced in order to allow more efficient DSM implementations, at the price of stricter constraints on the use of the shared data. In these models, consistency actions may be delayed till synchronization operations with locks or barriers. Many implementations have illustrated the advantages of these models with respect to performance, nevertheless most efforts were limited to the *single-threaded* case: a unique, sequential execution flow runs on each cluster node. The main reason is the lack of integration of the DSM libraries with the thread libraries, which came out more recently.

DSM-PM^2 is an original platform which integrates both aspects: *multiple* threads can run concurrently on the each node and keep sharing global data. Within this framework, multithreaded consistency protocols can be designed, implemented and evaluated. Such multithreaded protocols may be derived from classical, single-threaded protocols, but also original protocols can be designed, relying for instance on thread migration or implementing a semantics based on the thread concept (like Java consistency).

We present below two multithreaded protocols for realease consistency, derived from classical, single-threaded protocols.

3.1 A MRSW, invalidation-based protocol: erc_sw

The erc_sw protocol (*Eager Release Consistency, Single Writer*) is a MRSW protocol implementing *eager* release consistency: consistency actions are taken immediately on exiting a critical section (lock *release*), as opposed to *lazy* protocols which may delay such actions until another node enters a critical section (lock *acquire*). In erc_sw, the *acquire* routine takes no consistency action. Pages are manages using a dynamic distributed manager, as in li_hudak. Each page may be modified by a single node at a time (the *owner*). The other nodes may require read-only copies of the page from this node. On modifying the page, the owner sets a flag. At the exit of the critical section, the *release* operation invalidates all copies of the pages whose flags are set. Thus, any node which subsequently reads the page will need to ask for an up-to-date copy. If a node needs to write to the page, it musk ask the page to the owner, together with the page ownership.

The general scheme of this protocol is classical and has been implemented in other DSM systems [3]. Our contribution consists in providing a more complex, multithreaded version, which efficiently handles concurrent accesses on a page on each node. Traditionally, single-threaded systems process page faults sequentially on each node: no page fault is processed until the previous page fault is completed. In a multithreaded environment, the hypothesis of the *atomicity* of page fault handlers is not valid any longer. Threads waiting for a page need to give hand to other threads, in order to allow an efficient overlap of the page transfer by computation. Consequently, multiple threads may produce page faults to the *same* page and the accesses need to be served concurrently in an efficient way, by minimizing the number of page requests. For instance, a single page request may suffice to serve several page accesses. Also, in erc_sw, page requests

may be concurrent with invalidation requests. The implementation guarantees that the consistency constraints are observed, while allowing a high degree of concurrency. Handling these aspects makes the design of multithreaded protocols more difficult than in the single-threaded case.

3.2 A home-based protocol using multiple writers: hbrc_mw

The hbrc_mw protocol (*Home-based Release Consistency, Multiple Writers*) implements a *home-based* approach: each page is statically associated to a node which is in charge of always keeping an up-to-date copy of that page. As in the previous protocol, the *acquire* routine takes no action for consistency. When a thread needs to *read* a shared page, an up-to-date copy is brought from the home node. For a *write* access, a page copy is brought in if necessary, then a *twin* page is created. The thread can then modify the page. On exiting the critical section, each modified page is compared word-by-word with its twin and the modifications detected (called *diffs*) are eagerly sent to the home node, which applies them to the reference copy and then sends invalidation requests to all nodes which keep a copy of the page. In response to these invalidations, the other nodes which have concurrently modified the page compute and send their diffs to the home-node.

As opposed to the previous protocol, hbrc_mw allows a page to be concurrently modified on multiple nodes. This avoids the ping-pong effects produced by erc_sw protocol when two nodes concurrently write to disjoint addresses on the same page (*false sharing*). Also, the twinning technique reduces the communication overhead, since only the modified data is sent to the home-node, instead of whole pages. On the other hand, more processing time and more memory is required (for twin creation and diff computation). The tradeoff between these aspects depends on the access patterns to shared data, which vary from one application to another.

Here again, our contribution consists in designing a *multithreaded* protocol based on the scheme presented above. The protocol routines are not atomic, as is usually the case in single-threaded systems. As in the case of the erc_sw protocol, accesses to the *same* page, on the *same* node, are processed *concurrently* in an efficient way. Concurrency also needs to be carefully handled for *release* operations: the *release* function may be called concurrently by multiple threads, but diffs related to the same page are sent to the home node only once. Also, a release operation can run concurrently with an invalidation request sent by the home node (as a result of receiving diffs from some other node). The concurrent release and invalidation may require that the local diffs for the *same* page be sent to the home node, but the two operations need to synchronize so that the diffs be sent only once. These race situations are illustrated below.

3.3 Implementation in DSM-PM^2

The two protocols have been implementated in the C language and use the programming interface of DSM-PM^2's basic components (the DSM page manager and the DSM communication manager). Figure 2 illustrates the *release* operation of the hbrc_mw protocol. The code has been slightly simplified, for the sake of clarity. It illustrates a typical

situation of concurrency, specific to a multithreaded context. This function is activated by DSM-PM^2 on exiting a critical section.

The main loop of this function traverses the list of the pages modified since the last entry to a critical section. For each page, if the local node is the home node of the page, invalidations are sent to all nodes having a copy of the page. Otherwise, the function computes the diffs between the modified page and its saved twin and sends these diffs to the home node. Once the home node acknowledges their application, the function goes on to the next page.

The multithreaded implementation of this protocol needs to guarantee that concurrent calls to this function by different threads on the same node can proceed correctly. The dsm_pending_release function detects such a race. If a thread calls the release function while another thread is executing it and has already sent the diffs, then the second thread does not send the diffs again; it only waits for the diffs sent by the first thread to be acknowledged (wait_for_release_done).

Another situation of concurrency occurs when this release function is called by two threads on two *different* nodes $N1$ and $N2$. If the threads have modified the same page, both need to send the diffs to the home node. On receiving the first diffs, say from node $N1$, the home node will send an invalidation to node $N2$. The node $N2$ then has to respond by sending its own diffs. Independently of this, the *release* operation on node $N2$ also requires that the diffs for the same page be sent to the home node. The implementation guarantees that the diffs are sent only once, while allowing both operations on node $N2$ (release and invalidation) to proceed concurrently. This is achieved thanks to the the test diffs_to_send.

4 Experimental results

We have studied the behavior of our two protocols for release consistency using a 1D FFT kernel extracted from the SPLASH-2 [13] benchmark suite, which we adapted for a multithreaded execution. The program uses two shared matrices $\sqrt{n} \times \sqrt{n}$ distributed on processors by blocks of lines. Communication takes place during 3 transpose operations and involves all-to-all communication. Transpose operations are blocked in a pipeline fashion: processor i first transposes a submatrix from processor $i + 1$, then a submatrix from processor $i + 2$, etc. Our experiments have been carried out on two different platforms: 1) a cluster of 200 MHz Pentium Pro PCs under Linux 2.2.13, interconnected by Fast Ethernet under TCP; 2) a cluster of 450 MHz Pentium II PCs interconnected by a SCI network handled by the SISCI interface. The costs of the basic operations are given in Table 2.

In Figures 3 and 4 we compare the two protocols for release consistency to the li_hudak protocol provided by DSM-PM^2, which implements sequential consistency. The comparison has been done on our two clusters, using 4 nodes, with one application thread per node. The execution on the SCI cluster is about 20 times faster than on the Fast Ethernet cluster, because of the relative performance of the two networks in terms of bandwidth and latency. In both cases, the erc_sw and hbrc_mw protocols produce much more efficient execution times than the li_hudak protoco. This illustrates the benefits of release consistency compared to sequential consistency in the presence of

```
void dsmlib_hbrc_release () {
  if(!dsm_test_and_set_pending_release_or_wait()) {
    int index = remove_first_from_list (&page_list);
    while (index != NULL) {
      dsm_lock_page(index);
      if (get_owner (page) == dsm_self ())
        invalidate_copyset (page);
      else {
        if diffs_to_send(page) {
           send_diffs (page, get_owner (page));
           set_access (page, NO_ACCESS);
           free_twin (page);
           send_invalidate_req
                (get_owner (page), page);
           wait_ack (page);
        }
      }
      dsm_unlock_page(index);
      page = remove_first_from_list (&page_list);
    }
    clear_pending_release_and_broadcast_done();
  }
}
```

Fig. 2. The lock_release functionof the hbrc_mw protocol.

read/write false sharing, which occurs here (i.e., one node writes a page while another one reads it). The figures obtained on TCP illustrates the superiority of the hbrc_mw protocol compared to erc_sw, thanks to its multiple writers, which results in a reduced number of page transfers. The less performant the network, compared to the processors, the more important this advantage.

We have also studied the influence of multithreading on the efficiency of the protocols, by varying the number of threads per node for a given problem size (256 kB) and for a fixed number of nodes (4). The execution times are presented in Tables 3 and 4. First, let us note that the overhead due to multithreading is not significant. On the other hand, when the communication-to-computation ratio is high (i.e., for a low-speed network and for consistency protocols which involve a lot of communications), using more

Operation	TCP/Fast Ethernet (PPro 200 MHz)	SISCI/SCI (PII 450 MHz)
Page fault	23	11
Transmit request	370	38
Page request	3900	119
Page protection	22	12

Table 2. Cost of elementary operations (μs).

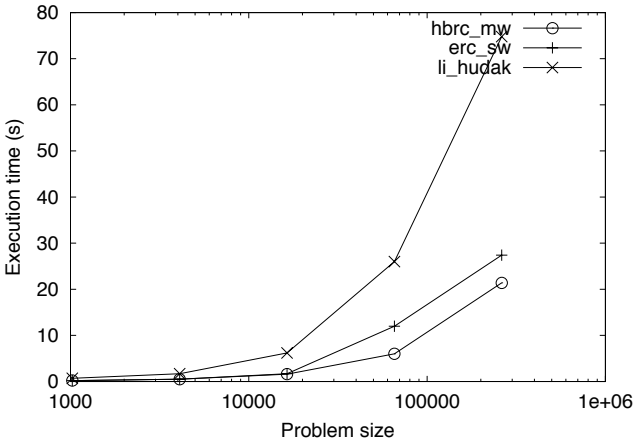

Fig. 3. FFT on TCP/Fast Ethernet (4 nodes).

Threads/node	1	2	4	8
li_hudak	74.8	54.8	54.4	57.5
erc_sw	27.2	21.0	19.6	24.1
hbrc_mw	21.4	20.9	20.9	21.1

Table 3. FFT on TCP/Fast Ethernet (seconds).

than one application thread per node may improve performance, thanks to the overlap of communication by computation. A finer analysis of these phenomena is currently in progress.

5 Conclusion

We have illustrated how *multithreaded* protocols can be designed, implemented and evaluated within the unified framework provided by DSM-PM^2. This platform appears as an interesting tool for developers of distributed, multithreaded applications using a shared memory model, since it facilitates DSM protocol implementations by providing a lot of common, basic components. In this paper, we focused on two protocols imple-

Threads/node	1	2	4	8
li_hudak	79	57	44	36
erc_sw	1.3	1.3	1.4	1.4
hbrc_mw	1.2	1.2	1.2	1.2

Table 4. FFT on SISCI/SCI (seconds).

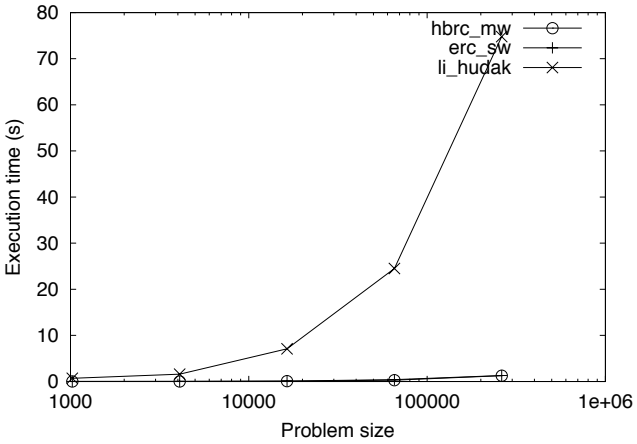

Fig. 4. FFT on SISCI/SCI (4 nodes).

menting the *release consistency* model through alternative mechanisms, and we highlighted some issues related to the higher degree of concurrency due to a multithreaded context. The approach presented can be generalized to other consistency models (such as Java consistency, or scope consistency).

Acknowledgments

We thank Vincent Bernardi for his help with the implementation of the two protocols for release consistency within DSM-PM^2.

References

1. G. Antoniu and L. Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001. Held in conjunction with IPDPS 2001. IEEE TCPP., Springer-Verlag.
2. 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.
3. 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.
4. 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.

5. IEEE. *Standard for Scalable Coherent Interface (SCI)*, August 1993. Standard no. 1596.
6. L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, 87(3), March 1999.
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*. PhD 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. 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.
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.