Implementing Java consistency using a generic multithreaded DSM runtime system

Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark Macbeth, Keith Mcguigan, Raymond Namyst

HAL Id: inria-00563587
https://hal.inria.fr/inria-00563587

Implementing Java consistency using a generic, multithreaded DSM runtime system

Gabriel Antoniu[1], Luc Bougé[1], Philip Hatcher[2],
Mark MacBeth[2]*, Keith McGuigan[2], Raymond Namyst[1]

[1] LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
[2] Dept. Computer Science, Univ. New Hampshire, Durham, NH 03824, USA

Abstract. This paper describes the implementation of Hyperion, an environment for executing Java programs on clusters of computers. To provide high performance, the environment compiles Java bytecode to native code and supports the concurrent execution of Java threads on multiple nodes of a cluster. The implementation uses the PM2 distributed, multithreaded runtime system. PM2 provides lightweight threads and efficient inter-node communication. It also includes a generic, distributed shared memory layer (DSM-PM2) which allows the efficient and flexible implementation of the Java memory consistency model. This paper includes preliminary performance figures for our implementation of Hyperion/PM2 on clusters of Linux machines connected by SCI and Myrinet.

1 Introduction

The Java programming language is an attractive vehicle for constructing parallel programs to execute on clusters of computers. The Java language design reflects two emerging trends in parallel computing: the widespread acceptance of both a thread programming model and the use of a (possibly virtually) shared memory. While many researchers have endeavored to build Java-based tools for parallel programming, we think most people have failed to appreciate the possibilities inherent in Java's use of threads and a "relaxed" memory model.

There are a large number of parallel Java efforts that connect multiple Java virtual machines by using Java's remote-method-invocation facility (e.g., [2, 3, 13]) or by grafting an existing message-passing library (e.g., [4, 5]) onto Java. In contrast, we view a cluster as executing a *single* Java virtual machine. The separate nodes of the cluster are hidden from the programmer and are simply resources for executing Java threads with true concurrency: the Java threads are mapped onto the native threads available at the nodes. The private memories of the nodes are also hidden from the programmer: our implementation supports the illusion of a shared memory within a distributed cluster. This illusion is consistent with respect to the Java memory model, which is "relaxed" in that it does not require sequential consistency.

---

* Current affiliation: Sanders, A Lockheed Martin Company, PTP02-D001, P.O. Box 868, Nashua, NH, USA

The difficulty is that the Java memory model does not exactly match one of the "classical" memory consistency models that are supported by existing distributed shared memory (DSM) systems. Moreover, our approach requires a tight integration of a Java specific DSM system with a native thread management system. Designing such an environment from scratch is a huge task, and only very few such projects have succeeded. In contrast, we have built our system on top of a new, generic, multi-protocol, multithreaded DSM runtime platform, which provides all the primitives necessary for a distributed implementation of the Java consistency model. This allowed us to complete this task within a few weeks. Moreover, the genericity allowed us to explore several alternative implementation strategies with an invaluable flexibility. The preliminary performance figures reported are definitely encouraging.

2 Executing Java programs on distributed clusters

2.1 Concurrent programming in Java

It is relatively easy to write parallel programs in Java. Support for threads is part of the Java API, and provides similar functionality to POSIX threads. Threads in Java are represented as objects. The class java.lang.Thread contains all of the methods for initializing, running, suspending, querying and destroying threads. Critical sections of code can be protected by monitors. Monitors in Java are available through the use of the keyword synchronized. The keyword can be used inline in the code as a statement, or as a modifier for a method. As a statement, synchronized must be provided an object reference and a block of code to protect. A method modified with synchronized uses the instance of the object it is being called on and protects the method body. Every object has exactly one lock associated with it and in both cases a lock is acquired on the referenced object, the protected code is executed, and then the lock is released. Note that the class java.lang.Thread also contains the methods wait(), notify(), and notifyAll(), which provide functionality similar to POSIX condition variables. From monitors and the wait/notify methods, other synchronization constructs, such as barriers and semaphores, can easily be built.

Java has a well-defined memory model [6]. All threads share the same central memory, so all objects, static values, and class objects are accessible to every thread. Thus, reads and writes of such values should be protected by monitors when appropriate to prevent race conditions. 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.

2.2 The Hyperion System

Hyperion is an environment for the high-performance execution of Java programs developed at the University of New Hampshire. Hyperion supports high performance by utilizing a Java-bytecode-to-C translator and by supporting parallel

execution via the distribution of Java threads across the multiple processors of a cluster of workstations.

We are interested in computationally intensive programs that can exploit parallel hardware. Therefore we expect that the added cost of compiling to native machine code will be recovered many times over in the course of executing a program. (This focus on compilation distinguishes us from projects investigating the implementation of Java interpreters on top of a distributed shared memory [1, 15].)

To produce an executable for a user program, Java bytecode is first generated from the Java source using a standard Java compiler. (We currently use Sun's javac.) The bytecode in the generated class files is compiled using Hyperion's java2c. The resulting C code is compiled using a native C compiler for the cluster and then linked with the Hyperion runtime library and any necessary external libraries.

Hyperion's Java-bytecode-to-C compiler, java2c, uses a very simple approach to code generation. Each virtual machine instruction is translated directly into a separate C statement or macro invocation, similar to the approaches taken in the Harissa [10] or Toba [14] compilers. Currently, java2c supports all non-wide format instructions as well as exception handling.

The Hyperion run-time system is structured as a collection of subsystems in order to support both easy porting to new target architectures and experimentation with different implementation techniques for individual components.

- The threads subsystem provides support for lightweight threads, on top of which Java threads can be implemented.
- The communication subsystem supports the transmission of messages between the nodes of a cluster.
- The memory subsystem is responsible for the allocation, management (including synchronization mechanisms), and garbage collection of Java objects. Table 1 provides the key primitives for implementing memory consistency. On a distributed implementation these primitives need to be supported by the underlying DSM layer.

loadIntoCache	Load an object into the cache
invalidateCache	Invalidate all entries in the cache
updateMainMemory	Update memory with modifications made to objects in the cache
get	Retrieve a field from an object previously loaded into the cache
put	Modify a field in an object previously loaded into the cache

Table 1. The Hyperion DSM subsystem API

The compiler generates explicit calls to put and get to access shared data. Objects are loaded from the main memory to the local cache using loadIntoCache. The primitives invalidateCache and updateMainMemory are called on entering/exiting monitors to ensure consistency, as described in Section 3.2.

To implement the above primitives, we utilize DSM-PM2, a generic, multi-protocol DSM layer built on top of the PM2 multithreaded runtime system. It provides an easy-to-use API for specifying consistency protocols.

3 Implementing Java consistency

3.1 DSM-PM2: a generic, multi-protocol DSM layer

PM2 (Parallel Multithreaded Machine) [11] is a multithreaded environment for distributed architectures. Its programming interface is based on *Remote Procedure Calls* (RPCs). 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. Threads running on the same node can freely share data. Threads running on distant nodes can only interact through RPCs.

PM2 provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such functionality is typically useful to implement dynamic load balancing policies. The interactions between thread migration and data sharing are handled through a *distributed shared memory* facility: the *DSM-PM2* layer.

DSM-PM2 provides a programming interface to manage static and dynamic data to be shared by all the threads running within a PM2 session, whatever their location. To declare *static* shared data, one simply brackets the corresponding C declarations by specific DSM-PM2 keywords. *Dynamic* shared data are allocated as needed by calling a specific allocation routine instead of the ordinary malloc primitive.

Since the DSM-PM2 API 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. Applications can thus be programmed as if a true physical shared memory was available. Nevertheless, an application can choose to bypass the fault detection mechanism by controlling the accesses with explicit calls to get/put primitives. In some cases, the resulting cost may be smaller than the overhead of the underlying fault handling subsystem. DSM-PM2 copes with this approach as well, as illustrated by the implementation of the Java runtime discussed in Section 3.2.

Since existing DSM applications require different consistency models, DSM-PM2 has been designed to be generic so as to support multiple consistency models. Sequential consistency and Java consistency are currently available. Moreover, new consistency models can be easily implemented using the existing generic DSM library routines.

These primitives may also be used to provide alternative protocols for a given consistency model. For instance, the usual action on a access fault is to bring

the data to the accessing thread. Alternatively, one may choose to preemptively migrate the thread to the data: this may be much more efficient in certain cases! One may even build hybrid protocols, which are able to dynamically switch from one policy to another depending of some external load information.

The overall structure of the DSM-PM2 layer is presented in Figure 1. 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. The *DSM communication module* is responsible for providing elementary communication mechanisms such as delivering requests for page copies, sending pages, and invalidating pages. It implements a convenient high-level communication API based on RPCs. On top of these two components, the *DSM protocol library* provides elementary routines that are used as building blocks to implement consistency protocols. For instance, it includes routines to bring a copy of a page to a thread, to migrate a thread to a page, to invalidate all the copies of a page, etc. Finally, the *DSM protocol policy* layer is responsible for implementing consistency models out of a subset of the available library routines and for associating each application data with its own consistency model.

Fig. 1. Overview of the DSM-PM2 software architecture
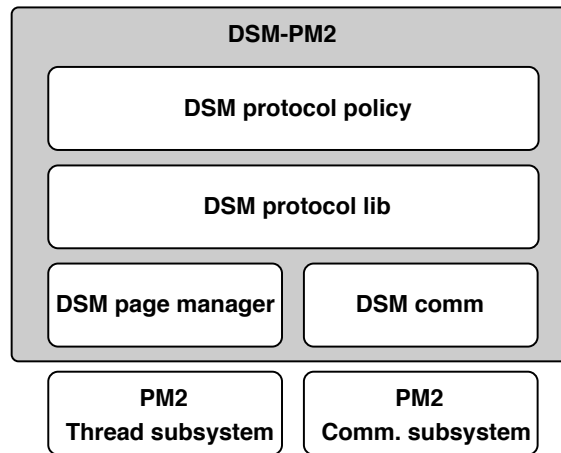
3.2 Using DSM-PM2 to build a Java consistency protocol

Java consistency requires a MRMW (*Multiple Reader Multiple Writer*) protocol: an object can be replicated and copies may be concurrently modified on different nodes. To guarantee consistency, the accesses to shared data have to be protected by monitors (corresponding to the keyword synchronized at the language level).

On entering a monitor, the primitive **invalidateCache** is called. Objects are loaded from the main memory to the local cache using **loadIntoCache.** Finally, on exiting a monitor, the modifications carried out in the local cache are sent to the main memory via the **updateMainMemory** primitive. We have implemented these protocol primitives using the programming interface of the lower-level DSM-PM2 components: *DSM page manager* and *DSM communication module.*

Main memory and caches. To implement the concept of *main memory* specified by the Java model, the runtime system associates a *home node* to each object. It is in charge of managing the reference copy. Initially, the objects are stored on their home nodes. They can be 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.

Access detection. Hyperion uses specific access primitives to shared data (**get** and **put**), which allows us to use explicit checks to detect if an object is present (i.e., has a copy) on the local node. If the object is locally cached, it is directly accessed, else the **loadIntoCache** primitive is invoked. The default mechanism for access detection provided by DSM-PM2 is thus bypassed and the cost of page fault handling is saved. An alternative we have planned to investigate would be to allow direct access and call **loadIntoCache** within the page fault handler.

Access rights. Java objects cannot be read-only, so that a node can either have full *read-write* access to an object, or have no access at all. This feature simplifies the protocol implementation, since only these two cases have to be considered.

Sending modifications to the main memory. Since shared data are modified through a specific access primitive (**put**), the modifications can be recorded at the moment when they are carried out. For this purpose, a bitmap is created on a node when a copy of the page is received. The **put** primitive records all writes to the page. The modifications are sent to the home node of the page by the **updateMainMemory** primitive.

Pages and objects Java objects are implemented on top of pages. Consequently, loading an object into the local cache may generate prefetching, since all objects on the corresponding page are actually brought to the current node. On the other hand, **updateMainMemory** may generate non-required updates, since the modifications of all objects located on the current page are sent to the home node. These side effects do not affect the validity of our protocol with respect to Java consistency.

4 Preliminary performance evaluation

We first evaluate the performance of our protocol primitives on three different platforms. The first column corresponds to measurements carried out on a cluster of Pentium II, 450 MHz nodes running Linux 2.2.10 interconnected by a SCI

network using the SISCI protocol. The figures in the next two columns have been obtained on a cluster of Pentium Pro, 200MHz nodes running Linux 2.2.13 interconnected by a Myrinet network using the BIP and TCP protocols respectively. The cost of the **loadIntoCache** primitive for a 4 kB object can be broken down as follows (time is given in μs):

Operation/Protocols	SISCI/SCI	BIP/Myrinet	TCP/Myrinet
Preparing a page request	1	2	2
Transmitting the request	17	30	190
Processing the request	1	2	2
Sending back a 4 kB page	85	134	412
Installing the page	12	24	24
Total	**116**	**192**	**630**

The processing overhead of DSM-PM2 with respect to the raw transmission time is 10–15%. The overhead related to page installation includes a call to the **mprotect** primitive to enable writing and a call to **malloc** to allocate the page bitmap necessary for recording local modifications. This latter cost could be further improved using a custom **malloc**-like primitive.

As a preliminary test, we ran a program that estimates π by calculating a Riemann sum of 50 million values. Parallelism can be utilized by assigning sections of the Riemann sum to different threads and then doing one final sum reduction to complete the calculation. Compared to an optimized sequential C program, the Java/Hyperion program achieves the following speedups on the Myrinet/BIP cluster described above (time is given in seconds):

# Nodes	Time	Speedup	Efficiency
C code	9.4	1.0	100%
1	9.6	.98	98%
2	4.9	1.9	95%
4	2.5	3.8	95%
6	1.7	5.5	92%
8	1.3	7.2	90%

Even if this pi program exhibits good performance, it is admittedly rather simple! For a more complex evaluation of our approach, we are currently working on a minimal-cost graph-coloring application.

5   Conclusion

We propose utilizing a cluster to execute a single Java Virtual machine. This allows us to run threads completely transparently in a distributed environment. Java threads are mapped to native threads available on the nodes and run with true concurrency. Our implementation supports a globally shared address space via the DSM-PM2 runtime system that we configured to guarantee Java consistency.

We plan to make further evaluation tests using more complex applications. Thanks to the genericity of DSM-PM2, we will be able to study alternative protocols for Java consistency. We also intend to perform comparisons between different access detection techniques (segmentation faults vs. explicit locality checks).

References

1. Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.
2. F. Breg, S. Diwan, J. Villacis, et al. Java RMI performance and object model interoperability: Experiments with Java/HPC++. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 91–100, February 1998.
3. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and meta-computing in Java. *Concurrency: Practice and Experience*, 10:1125–1242, 1998.
4. A. Ferrari. JPVM: Network parallel computing in Java. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 245–249, 1998.
5. V. Getov, S. Flynn-Hummell, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 45–54, February 1998.
6. J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.
7. P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *High-Performance Computing and Networking (HPCN '98)*, volume 1401 of *Lect. Notes in Comp. Science*, pages 628–637. Springer-Verlag, 1998.
8. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Computer Systems*, 7(4):321–359, November 1989.
9. F. Mueller. Distributed shared-memory threads: DSM-Threads. In *Proc. of the Workshop on Run-Time Systems for Parallel Programming (RTSPP '97)*, pages 31–40, Geneva, Switzerland, April 1997. Held in conjonction with IPPS '97.
10. G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Third Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 1–20, Portland, June 1997.
11. Raymond Namyst and Jean-François Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.
12. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE computer*, 24(8):52–60, September 1991.
13. M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.
14. T. Proebsting, G. Townsend, P. Bridges, et al. Toba: Java for applications — a way ahead of time (WAT) compiler. In *Third Conference on Object-Oriented Technologies and Systems (COOTS '97)*, Portland, June 1997.
15. W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of the Workshop on Java for High-Performance Scientific and Engineering Computing*, Las Vegas, Nevada, June 1997.