DISK:
A Distributed Java Virtual Machine
For DSP Architectures

Mihai Surdeanu
Southern Methodist University, Computer Science Department
mihai@seas.smu.edu

Abstract

Java is for sure the most popular programming language today. Its popularity is based on the portability of the code and on the elegant programming framework provided by the language: built in garbage collection and multi-threaded support, easy Internet application development through socket streams, and last but not least familiar syntax. Even if a lot of work has already been done related to Java Virtual Machines (JVMs) and a large number of Java applications are already available, few of these have penetrated the DSP world. Our work aims to bridge this gap between DSP architectures and Java. Such a connection would open the DSP world to a much larger class of consumers, a place now occupied by general-purpose processors.

The two main research directions we see in improving the performance of a JVM for DSP architectures are: (1) writing a Java bytecode Just-In-Time compiler optimized for a specific DSP platform, and (2), taking advantage of the available parallelism present in DSP multi-processor architectures. Our work focuses on the last issue. We propose a distributed JVM designed for a multi-processor DSP architecture. Due to the unavailability of the hardware at the moment, we have implemented our ideas on a network of general-purpose processors, emulating the topology of the quad-processor DSP board constructed by Innovative DSP. We emulate the I/O channels connecting the processors through TCP sockets and interrupts with Unix signals. All the protocols use a constant number of messages, making it possible to estimate the costs of all distributed actions.

In this document we show the design and test results obtained with DISK. While a distributed JVM architecture may not prove to be an exciting project for highly-specialized DSP processors (such as TI's TMS320 family), we believe that this work may be successfully applied to multi-processor architectures based on RISC processors enhanced with signal processing capabilities (such as the new ARM9E processor). Other projects are working on porting Java to this class of processors. On top of this we provide a Java parallel-processing platform that binds the power of Java distributed computing with signal processing.

1 Introduction

Java is slowly starting to penetrate the DSP world. Both JDK [3] and Kaffe [5] have ports available for ARM processors. Nevertheless, the world of DSP multiprocessing remains unexplored by Java environments or applications. There are JVMs for multiprocessor architectures built with general-purpose processors such as Solaris or NT SMPs, but none for DSP multiprocessors.

The reason is not only the fact that Java ports have been available for DSP architectures only recently but also the fact that most (if not all) DSP multiprocessor systems *lack shared memory*. The C6X multiprocessor boards built by BlueWave [1] and Innovative Integration [2] have separate memories for each processor on board. Inter processor communication (if available) is performed through dedicated I/O channels. Hence, even if the processors are tightly connected, we believe that these architectures are better described as multicomputers, where the inter processor communication is performed through message passing not shared memory. Another possible architecture fitting the same classification is a cluster of ARM processors connected by a high-speed network.

The Java Virtual Machine Specification (JVMS) [4] makes the assumption that shared memory is available. Once this assumption is no longer true, as is the case with DSP multiprocessors, one needs a distributed shared memory protocol to provide the illusion of shared memory.

It is possible to show that under the assumption that programs are data-race free, the JVMS is equivalent with Release Consistency [6], a well-known consistency model for distributed shared memories. It is beyond the purpose of this paper to show this equivalence. Instead, we describe a distributed JVM architecture built upon the Release Consistency model, named DISK (DIStributed Kaffe). As the name implies, we built our system on top of the Kaffe Virtual Machine (we used version 1.0.b3). DISK provides a distributed Java environment, where threads are transparently allocated to processors based on a customizable processor allocation algorithm. The object heap is kept consistent using one of the four protocols implementing Release Consistency. DISK is thoroughly implemented at the JVM level. We do not require any changes to the Java language or compiler.

The DISK architecture and protocols are described in section 2. DISK is a distributed-shared-memory system implemented from scratch to take advantage of the Java unique features (object-oriented, just-in-time compiling). Section 3 presents the test results for three benchmark applications. We test the performance of four protocols implementing Release Consistency. Such a study is not yet available for the Java language. Our results indicate that there is no clear winner, each protocol performing best under specific circumstances. This indicates that a run-time adaptive protocol might be the best solution overall. Section 4 concludes the paper.

2 Architecture

In this section we describe the DISK architecture and the protocols used to implement the Release Consistency model.

We show the block diagram of a DISK node in figure 2.1. As shown in the figure we use the *Kaffe VM* to execute Java bytecode. The interaction between Kaffe and DISK's distributed environment is done through *software barriers*. Release Consistency is maintained over the cluster of workstations through the protocols implemented in the *RC Protocols* module. The *object, lock, and thread management* module is present only on master processors. This module tracks the location of every global object and lock, and the processor where every thread is allocated. We currently use one master processor per cluster, but the management tasks could be easily distributed to a larger number of processors. The *communication layer* is in charge of the input and output communication channel management. We have currently implemented the

communication layer on top of TCP, but it could easily be ported over any reliable communication protocol. The RC-compliant protocols and the management module communicate in a topology-independent way through the services offered by a *light Remote Procedure Call* (RPC) protocol.

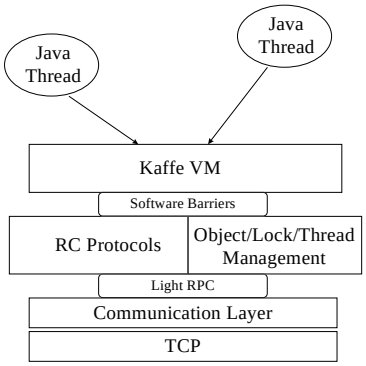

Figure 2.1: Block diagram of a DISK node

DISK is implemented over a bus-connected cluster of workstations. On top of this, the communication layer constructs software point to point channels between all nodes. The topology obtained was highly influenced by the quad-processor C6x board designed by Innovative Integration. This topology is depicted in figure 2.2 for a system with four processors.

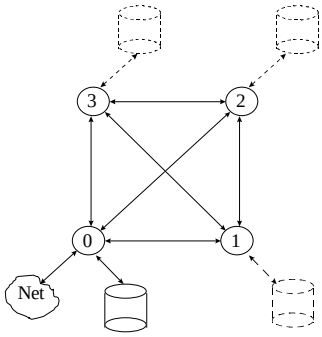

Figure 2.2: Point-to-point-connected system with centralized I/O

Note that only processor 0 (the master processor) has full I/O access. This is because the master processor runs also a file server that provides I/O transparency for the Java threads. To reduce initialization overhead we provide partial I/O capabilities to the slave processors to allow them to independently load Java classes at startup. This design choice yielded an approximate two times faster system initialization compared to the system where all I/O accesses go through the master processor. On a tightly connected DSP multiprocessor board, the Java default classes could be stored inside processor-local ROM for speedup (Kaffe's classes fit in a 500 KB buffer).
The block diagram of the communication layer is presented in figure 2.3.

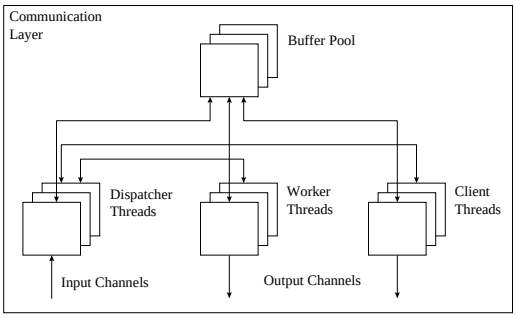

Figure 2.3: DISK communication layer

Each DISK node supports three types of threads:

- *Dispatcher threads* - Each input channel is monitored by a dispatcher thread. The dispatchers are dormant most of the time. They are awaken by the thread system whenever a packet arrives on the monitored channel. The dispatcher reads the packet from the channel into a memory buffer. If the message contained in the packet is a reply, the thread waiting for it is awaken. If the message is a request, a worker thread is designated to serve the request.
- *Worker threads* - Each processor has a pool of worker threads. A worker is created when a request waits to be serviced and the worker pool is empty. When a worker finishes the assigned request it returns to the pool where it waits for the next job.
- *Client threads* - Client threads are created by the Java application. For any program at least one client thread is created to run the main method.

To support this organization and to avoid excessive dynamic allocation at runtime (garbage collection is expensive), a *pool of buffers* is created at system initialization and dynamically extended according to the application needs. The dispatcher threads use these buffers when incoming packets are read from the I/O channels or by any other thread that needs to build a packet. The buffer returns to the buffer pool when it is not needed any more.

Having in mind a DSP environment, we made the assumption that virtual memory is not available, hence we do not use the virtual memory management unit to implement the consistency protocol. Instead we modify Kaffe's Just-In-Time (JIT) compiler to call our own consistency functions before any read or write access to object data. We modified the JIT code for the following bytecode instructions:

- PUTFIELD to insert a write barrier before object accesses;
- GETFIELD to insert a read barrier before object accesses;
- IASTORE, LASTORE, FASTORE, DASTORE, AASTORE, BASTORE, CASTORE, SASTORE to insert write barriers before array accesses;
- IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD, SALOAD to insert read barriers before array accesses.

In the current DISK version we do not change the PUTSTATIC and GETSTATIC instructions, hence we do not offer consistency support for class (static) data.
Due to the software barriers the semantics of the above instructions, plus MONITOR_ENTER, MONITOR_EXIT, and thread creation are changed. To keep things familiar for the reader not used with the Java instruction set, we rename the instructions that require read barriers as READ, the instructions that require write barriers as WRITE, MONITOR_ENTER and

MONITOR_EXIT as LOCK and UNLOCK. We also use the name TCREATE to indicate the thread create operation. The semantics of these instructions are presented in figure 2.4.

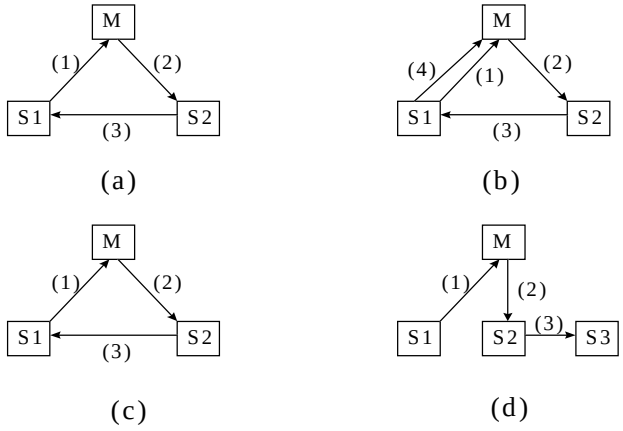

Figure 2.4: Distributed semantics for bytecode instructions.
(a) READ; (b) WRITE; (c) LOCK; (d) TCREATE

In figure 2.4 we denote with M the master processor, with S1 the (slave) processor trying to access a shared resource, and with S2 the processor having the resource. The semantics for READ and LOCK are obvious. During a WRITE operation we have to make sure that the object location recorded by the master processor is protected. Hence, message (1) locks the object record on the master processor, and message (4) unlocks it after the new location of the object is recorded. Message (1) of the TCREATE operation is sent to the master processor in response to a call to *java.lang.Thread.start()*, the Java native method that starts a new thread. The master processor allocates S3 as the host processor of this new thread, based on some processor allocation algorithm (currently we use a round-robin allocation algorithm). Message (2) is sent to the processor S2 currently owning the Java *Thread* object corresponding to the new thread. Message (3) containing a copy of this object is sent to processor S3 where the new thread is created. Note that no reply is expected. Processor S1 resumes execution as soon as message (1) is successfully delivered.
Note that the operations described in figure 2.4 apply only to *shared* objects *not available* locally. DISK is intelligent enough to detect if an object is not shared (i.e. if it is accessed only by threads located on the same processor), in which case the software barrier is greatly simplified.

DISK provides Release Consistency (RC), a JVMS compliant consistency model, for shared objects. An informal description of RC is that the changes to objects guarded by locks are sent to remote processors only when the lock is released. We refer the reader interested in a formal definition of RC to [6]. As of today there are two main flavors of RC: Eager Release Consistency (ERC) which forces the propagation of changes during the UNLOCK operation, and Lazy Release Consistency (LRC) [7], which delays sending the changes to shared objects until the time of the next LOCK operation. [7] shows that these two models are equivalent for programs where all conflicting accesses to shared objects are protected by synchronization mechanisms (data-race free programs). Just like cache coherence protocols, both ERC and LRC can be invalidate-based (when only invalidation messages are sent) or update-based (when the modified data is appended to consistency messages).
The protocols implementing RC may decide to maintain a single writable copy for each object (similar with cache coherence protocols), or allow multiple writable copies of the same object [8][10]. In the present stage we implement only single-writer protocols but we are planning to

investigate the performance of multiple-writer protocols as well. At this point, DISK supports the following protocols:

- LSI – LRC, single writer, invalidate
- LSU – LRC, single writer, update
- ESI – ERC, single writer, invalidate
- ESU – ERC, single writer, update

The performance of these protocols is analyzed in section 3.

3 Test Results

We have tested the performance of the protocols introduced in section 2 on the following three benchmark applications:

- PMC – This is a producer – multiple consumer problem. The producer repeatedly writes to a shared object and blocks until all consumers have read the resource. The tests presented here have been performed for 8 consumers and 100 iterations.
- JCB – This is an implementation of the Jacobi method for solving partial differential equations. The problem input is a two-dimensional matrix. During each iteration, every element is updated to the average of its nearest neighbors. Each Java thread gets a number of adjacent rows to work on. These tests have been performed on a 240x240 matrix, with 8 threads and 25 iterations.
- MM – This is a parallel matrix multiplication algorithm, where each element in the output matrix is computed by a separate thread. We used 20x20 matrices for this application.

Note that these applications are not written to take advantage of the available parallelism. Instead, the purpose is to stress the system by using a large number of synchronization points, in order to evaluate the relative performance of the proposed protocols. Due to these features, these benchmarks do not scale well when the number of processors is increased. We performed the tests on 2, 3 and 4 processors.
Nevertheless, we believe that these programs are fit to test the relative performance of the protocols introduced in section 2. Even if these applications are simple, they span a large class of real life applications:

- PMC uses a small number of shared objects, but they are repeatedly accessed.
- JCB as well has a small number of shared objects, but their size is much larger than the average Java object size (each row is a separate object).
- MM uses a large number of shared objects (each matrix element is here a separate object), but the shared object size is much smaller than the average object size.

The results obtained are presented in figure 3.1.a to 3.1.e for PMC, 3.2.a to 3.2.e for JCB, and 3.3.a to 3.3.e for MM. For each application we measured the additional memory needed by the protocol, the execution time, the number and size of packets sent throughout the execution, and the number and size of shared objects and locks versus the total number and size of all objects and locks.

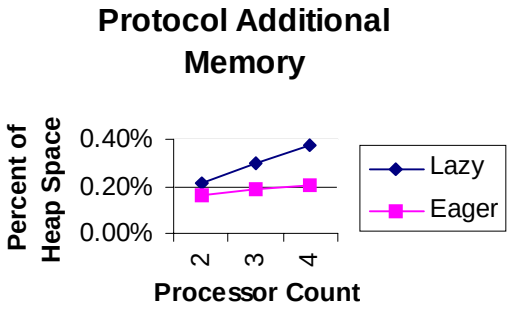

Figure 3.1.a: Protocol additional memory (PMC)

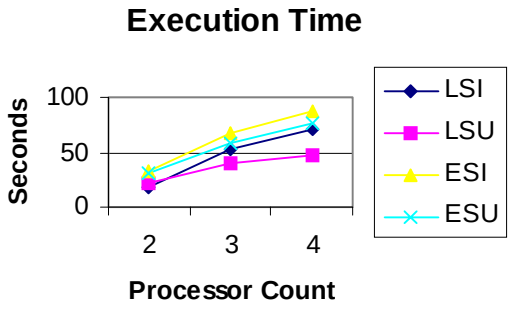

Figure 3.1.b: Execution time (PMC)

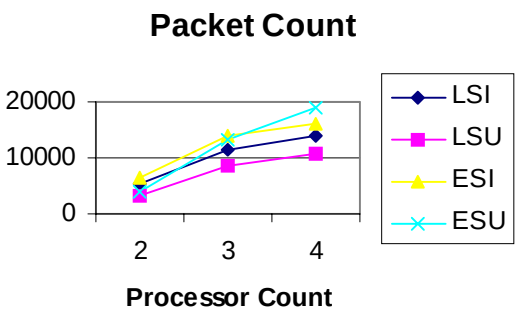

Figure 3.1.c: Packet Count (PMC)

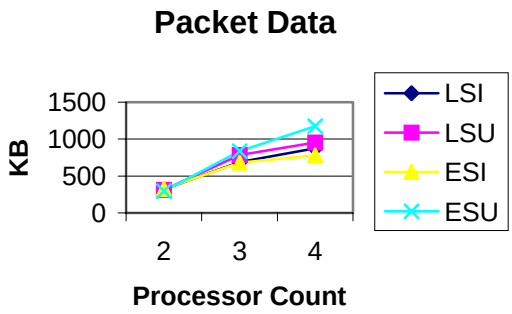

Figure 3.1.d: Packet Data (PMC)

	Total Objects		Total Locks		Shared Objects		Shared Locks	
	Count	Data	Count	Data	Count	Data	Count	Data
2	15923	641136	39	1560	19	1520	3	120
3	16377	667664	55	2200	22	1664	4	160
4	16839	694544	74	2960	25	1808	6	240

Figure 3.1.e: Shared versus total objects/locks (PMC)

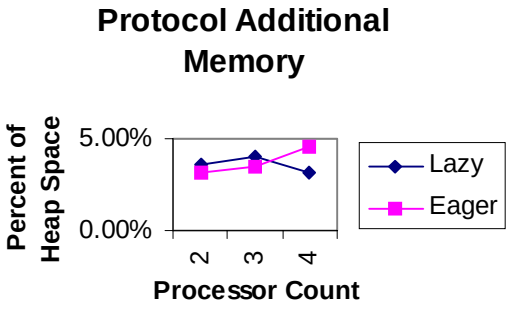

Figure 3.2.a: Protocol additional memory (JCB)

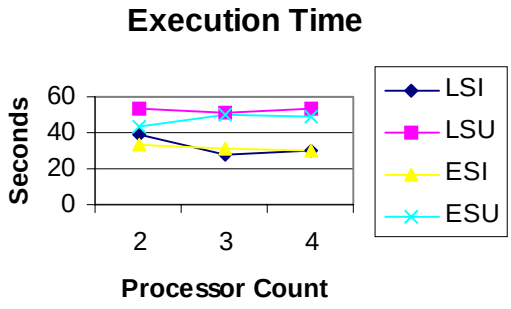

Figure 3.2.b: Execution time (JCB)

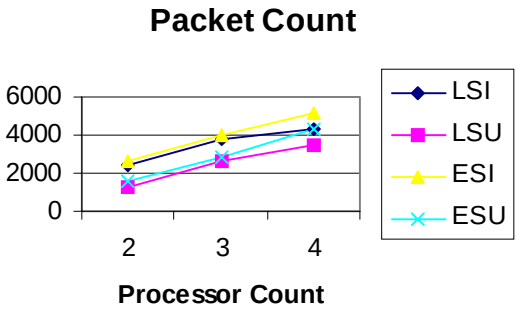

Figure 3.2.c: Packet count (JCB)

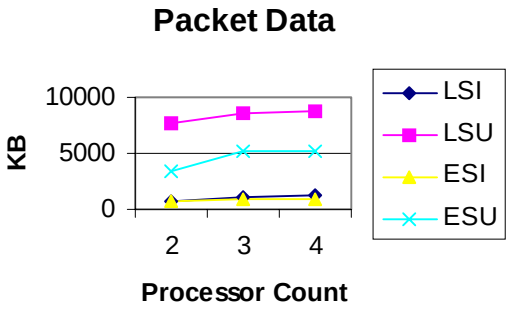

Figure 3.2.d: Packet data (JCB)

	Total Objects		Total Locks		Shared Objects		Shared Locks	
	Count	Data	Count	Data	Count	Data	Count	Data
2	1851	785216	47	1880	499	483520	16	640
3	2551	10530056	67	2680	744	724792	20	800
4	3249	1320672	85	3400	987	965840	22	880

Figure 3.2.e: Shared versus total objects/locks (JCB)

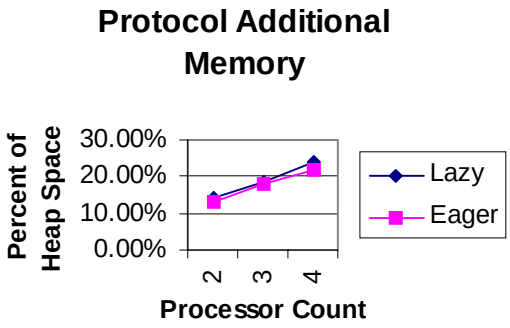

Figure 3.3.a: Protocol additional memory (MM)

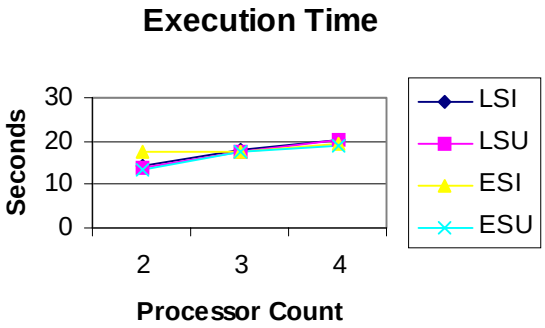

Figure 3.3.b: Execution time (MM)

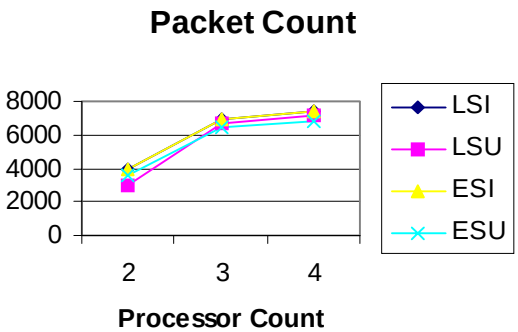

Figure 3.3.c: Packet count (MM)

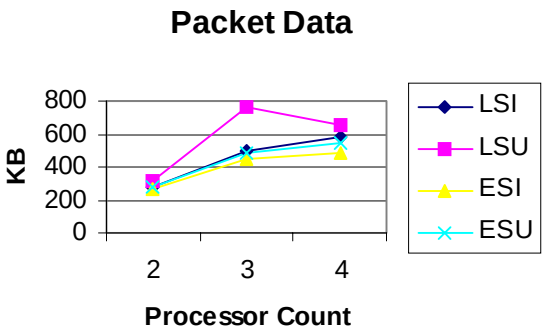

Figure 3.3.d: Packet data (MM)

Total Objects		Total Locks		Shared Objects		Shared Locks	
Count	Data	Count	Data	Count	Data	Count	Data

2	16406	1095432	634	25360	3131	139072	604	24160
3	18172	1164632	716	28640	4463	182472	673	26920
4	19904	1230024	764	30560	5761	222064	708	28320

Figure 3.3.e: Shared versus total objects/locks (MM)

PMC uses few shared objects hence the additional memory required by all protocols is negligible (see figure 3.1.a). The interesting result for this benchmark are the execution times presented in figure 3.1.b. The update protocols (both LSU and ESU) perform much better than their invalidate equivalent protocols. The explanation is based on the size of the shared objects (see figure 3.1.e). For PMC, the average shared object size ranges between 70 and 80 bytes. Even though this is about two times larger than the average Java object size reported in [12], it is still small. Hence, as figure 3.1.b and 3.1.c show, appending object data to consistency messages reduces the execution time by reducing the number of consistency messages sent. Obviously, the amount of data sent by the update protocols is larger than the invalidate protocols, but the difference is small, as figure 3.1.d shows.

An analysis of figures 3.1.c and 3.1.d shows that update protocols reduce the number of messages with the expense of sending slightly more data. This indicates that update protocols are better fit for systems where the message transfer overhead is large (i.e. TCP), hence one would like to reduce the message count as much as possible. On the other hand, invalidate protocols are fit for systems where the main concern is the amount of data to transfer, where the transfer initialization overhead is negligible (i.e. DSP processors tightly-connected on the same board).

Nevertheless, update protocols must be used with care. As seen in figure 3.2.d, for JCB where the average shared object size is large (roughly 1 KB), update protocols end up sending at least five times more data, even though the message count is less than the message count for invalidate protocols. As seen in figure 3.2.b, this yields execution times almost two times longer for the update protocols. This suggests that probably the best performance would be achieved by an adaptive protocol, a protocol capable to decide between update or invalidate based on the object size. We are planning to investigate such a protocol in the near future.

Figure 3.3.b shows a relatively close performance of all four protocols for the MM benchmark. The troubling result for this application is the large protocol overhead (almost 25% of the heap space). This is caused by the large number of very small size shared object (each element in the output matrix is a word-sized shared object). Having a large number of small size objects is the worst scenario for DISK, since every shared object is managed separately. Every shared object is allocated a separate header to store needed information such as the object state, the system-wide id, and queue of threads waiting for a valid copy of this object. This adds a significant overhead if the object size is small. While we are working to reduce the shared object header, these results are also an indication for the programmer. A wise programmer should reduce the number of shared objects (i.e. objects *referred* from a Thread object) to a minimum.

Another interesting observation is the fact that the ERC protocols, generally discarded as having a worse performance than LRC, are successfully competing with LRC protocols. In two out of three applications, ERC protocols outperform their equivalent LRC protocols. This can be explained by the relatively uniform distribution of shared objects accesses in all studied applications. Hence, broadcasting changes to all processors when locks are released, the way ERC protocols work, proves to be a better solution.

4 Conclusions

We have introduced in this paper a distributed Java virtual machine architecture we named DISK. Currently DISK is implemented on a cluster of general purpose processors, but it is immediately

portable to DSP architectures. All the hardware and operating system requirements needed by DISK are already presented on DSP platforms: we use the Kaffe virtual machine (already ported on ARM processors) to execute Java bytecode. We do not use the virtual memory management unit to implement the consistency protocol. Instead we modified Kaffe's just-in-time compiler to insert calls to our consistency functions before any object access. All these features make DISK portable to virtually any architecture.

DISK implements the Release Consistency model, a memory consistency model equivalent with the Java virtual machine specification. DISK currently supports four protocols implementing Release Consistency. The analysis of this protocols shows that, unlike page-based distributed shared memories, for DISK there is no clear winner. Different factors, like the speed of the interconnection network, the average shared object size, or object distribution, favor one protocol over the other. This suggests that an adaptive system capable to dynamically switch between protocols may be the best solution.

References

1. Blue Wave Systems. http://www.bluews.com
2. Innovative Integration. http://www.innovative-dsp.com
3. ARM. http://www.arm.com
4. The Java Virtual Machine Specification. http://java.sun.com
5. The Kaffe Project. http://www.kaffe.org
6. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. In Proceedings of the Seventeenth International Symposium on Computer Architecture, May 1990
7. P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. Ph.D. Thesis, Rice University, 1994
8. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamany, W. Yu, and W. Zwaenopoel. *TreadMarks: Shared Memory Computing on Networks of Workstations*. In IEEE Computer, vol. 29, No. 2, February 1996
9. C. Amza, A. Cox, S. Dwarkadas, L.J. Jin, R. Rajamany, and W. Zwaenopoel. *Adaptive Protocols for Software Distributed Shared Memory*. In Proceedings of IEEE, special issue on distributed shared memory, March 1999
10. J.B. Carter. *Design of the Munin Distributed Shared Memory System*. In Journal of Parallel and Distributed Computing, special issue on distributed shared memory, 1995
11. A.S. Tanenbaum. Distributed Operating Systems. Prentice-Hall, 1995
12. S. Dieckemann and U. Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. UCSB Technical Report TRC98-33, December 1998