A Java Environment for High-Performance Computing

Marcelo Lobosco[1], Claúdio Amorim[1], Orlando Loques[2]

[1] Laboratório de Computação Paralela, Programa de Engenharia de Sistemas e Computação, COPPE, UFRJ
Bloco I-2000, Centro de Tecnologia, Cidade Universitária, Rio de Janeiro, Brazil
{lobosco,amorim@cos.ufrj.br}

[2] Instituto de Computação, Universidade Federal Fluminense
Rua Passo da Pátria, 156, Bloco E, 3º Andar, Boa Viagem, Niterói, Brazil
{loques@ic.uff.br}

Abstract—

There has been an increasing research interest in extending the use of Java towards high-performance demanding systems such as scalable web servers, multimedia applications, and large-scale scientific applications. However, given the low performance provided by current Java implementations, these application domains pose new challenges to both the application designer and systems developer. In this paper we present CoJVM (Cooperative Java Virtual Machine), a new Java environment for high-performance computing designed to speed up Java applications when executing on DSM based architectures implemented on clusters of workstations. The shared memory implementation is based on the HLRC DSM protocol and takes advantage of run-time information, extracted from the JVM, to improve application performance.

Keywords— **Java, distributed-shared memory, parallel JVM implementation, high-performance computing, cluster computing**

I. INTRODUCTION

Java [ARN 96] is an object-oriented programming language, developed by Sun Microsystems, which incorporates features such as multithreading and primitives for concurrent programming. One of its main objectives is to allow the portability of programs among different hardware and operating system platforms. This objective is portrayed by the well-known slogan "Write once, run everywhere". The approach taken to reach this goal was the adoption of a standardized supporting platform called the Java Virtual Machine (JVM). The Java compiler generates a platform independent pseudo-code, called *bytecode*, which can then be executed in any computational environment (hardware & operating system) that supports the Java bytecode interpreter included in the standard JVM. The price paid for the portability, achieved through interpretation, as one might expect, is performance.

Several attempts intending to improve Java execution performance have been made, such as the addition of just-in-time compilation support and other optimizations techniques to Java execution environments [SUN 99]. Recent results showed that optimized Java code performed comparably to Fortran for some numerically-intensive regular computations [GUP 00]. However, these improvements were not enough to ensure that Java performed as well as C. Nevertheless, numerous systems for high-performance network computing developed to support Java applications have been proposed in recent years. The applications of these systems tend to be those of a large-scale computational nature, potentially requiring any combination of computers, networks, I/O, and memory, as defined by the Java Grande Forum [JGF]. Examples of such applications include data mining, satellite image processing, scalable web servers, and fundamental physics. At first glance, the choice of Java seems paradoxical, since it is an interpreted language. This single feature, however, has not been enough to dampen the great interest in its use in the development of high-performance computing environments.

Then, why should we use Java for High-Performance Computing? Besides the portability and interoperability achieved by a standard supporting environment, other features of the language such as its object-oriented programming model, simplicity, robustness, multithreading support, and automatic memory management have proved attractive enough for the development of software projects, especially those intended for large and complex systems. Also, the language portability has been decisive for its choice in projects that consider the use of idle computers, connected to the Internet, to solve large computational problems [BAR 98, CAP 97, FOX 96]. In addition, the growing popularity of the language helps to explain its use in the high-performance computing area.

In this paper we present a new Java environment for high-performance computing called **CoJVM** (**Co**operative **J**ava **V**irtual **M**achine). Its main objective is to speed up Java applications executing on a homogeneous cluster of workstations, our target architecture. CoJVM relies on two key features to improve application performance: 1) it uses the HLRC DSM protocol to implement the shared memory abstraction in the cluster [ZHO 96] and 2) it exploits application run-time behavior by introducing a new instrumentation mechanism to the JVM. Mostly important, the syntax and the semantics of the Java language are unaffected, allowing a programmer to write a program in the same way as he/she would write a concurrent program for a single Java Virtual Machine (JVM).

Overall, the main contributions of our work are: (a) to combine efficiently a home-base software DSM with a distributed JVM in order to build a high-performance Java environment for clusters; (b) to use the knowledge of application behavior at run-time, extracted from the JVM, to reduce the DSM protocol overheads. More specifically, we show that JVM information can be used in several effective ways to improve application performance, such as to create *diffs* dynamically; to vary the granularity of the coherence unit, and to detect automatically reads and writes to shared memory without paying the high cost of using the virtual-memory protection mechanism. Also, we show that further performance improvements can be achieved using VIA [COM 97] for network communication across the cluster.

The remainder of this paper is organized as follows. Section 2 describes Java support for multithreading and synchronization, and the Java memory model. Section 3 and 4 describes HLRC and the VIA standard, respectively. In section 5, we show that the Java syntax and semantics require no alteration in order to declare and to synchronize objects that are used in a typical DSM environment. In section 6, we describe the architecture and the implementation of CoJVM and show how the JVM instrumentation is used to optimize the HLRC protocol. In section 7, we present related works that implement the notion of shared memory in a Java environment. Finally, in section 8 we draw our conclusions.

II. JAVA

The Java language specification contains all the required concepts that are necessary for software DSM implementation of Java, although compatible Java DSM implementations are unavailable. This section describes such concepts.

A. Multithreading and Synchronization in Java

In Java, threads programming is simpler than in languages such as C and C++. This happens because Java already provides a native parallel programming model that includes support for multithreading. The package *java.lang* offers the *Thread* class that supports methods to initiate, to execute, to stop, and to verify the state of a running thread.

In addition, Java also includes a set of synchronization primitives, which are based on an adaptation of the classic monitor model as proposed by Hoare [HOA 74]. The standard semantics of Java allow the methods of a class to execute concurrently. The *synchronized* reserved word when associated with methods specifies that they cannot execute concurrently. In other words, these methods can only execute in a mutual-exclusion fashion according to the monitor paradigm.

B. Memory Model

The JVM specifies the interaction model between threads and the main memory, by defining an abstract memory system, a set of memory operations, and a set of rules for these operations. The main memory stores all program variables and is shared by the JVM threads (refer to figure 1). Each thread operates strictly on its local memory, so that variables have to be copied first from main memory to the thread's local memory before any computation can be carried out. Similarly, local results become accessible to other threads only after they are copied back to main memory. Variables are referred to as master or working copy depending on whether they are located in main or local memory, respectively. The copying between main and local memory, and vice-versa, adds a specific overhead to thread operation.

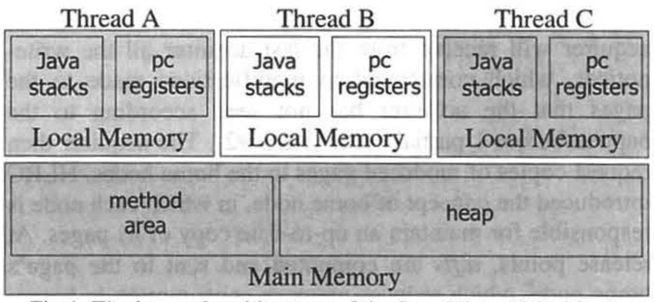

Fig 1. The internal architecture of the Java Virtual Machine's Memory

The replication of variables in local memories introduces a potential memory coherence hazard since different threads can observe different values for the same variable. The JVM offers two synchronization primitives, called *monitorenter* and *monitorexit* to enforce memory consistency. The primitives support blocks of code declared as synchronized. In brief, the model requires that upon a *monitorexit* operation, the running thread updates the master copies with corresponding working copy values that the thread has modified. After executing a *monitorenter* operation a thread should either initialize its work copies or assign the master values to them. The only exceptions are variables declared as *volatile*, to which JVM imposes the sequential consistency model. The memory management model is transparent to the programmer and is implemented by the compiler, which automatically generates the code that transfers data values between main memory and thread local memory.

III. SOFTWARE DSM

A software DSM system provides a shared memory abstraction on a cluster of computers. This illusion is often achieved through the use of the virtual memory protection mechanism, as proposed by Li [LI 89]. Two main shortcomings of such an approach are (a) the occurrence of false sharing and fragmentation phenomena due to the use of the large virtual page as the unit of coherence, which lead to unnecessary communication traffic; and (b) the high OS costs of treating page faults and crossing memory protection boundaries.

Several relaxed memory models, such as LRC [KEL 92], have been proposed to alleviate false sharing. In LRC, shared pages are write-protected so that when a processor attempts to write to a shared page an interrupt will occur and a clean copy of the page, called the twin, is built and the page is released to write. In this way, modifications to the page, called *diffs*, can be obtained at any time by comparing current copy with its twin. LRC imposes to the programmer the use of two explicit synchronization primitives: acquire and release. In LRC, coherence messages are delayed until an acquire is performed by a processor. When an acquire operation is executed the acquirer will receive from the last acquirer all the write-notices, which correspond to modifications made to the pages that the acquirer has not seen according to the happen-before-1 partial order [KEL 92]. The acquirer then request copies of modified pages to the home nodes. HLRC introduced the concept of home node, in which each node is responsible for maintain an up-to-date copy of its pages. At release points, *diffs* are computed and sent to the page's home node, which reduces memory consumption in home-based DSM protocols and contributes to the scalability of the HLRC protocol [ZHO 96].

IV. VIRTUAL INTERFACE ARCHITECTURE

The Virtual Interface Architecture (VIA) is a user-level memory-mapped communication architecture developed by the industry that aims to achieve low latency and high bandwidth communication. The main idea is to remove the critical path of communication from the operating system kernel. The operating system is called just to establish a communication channel, after which it is up to the user to manage all the communication.

A virtual interface (VI) is the interface between a NIC (Network Interface Card) and a process that allows the VI direct access to the process' memory. The VI represents a communication endpoint and pairs of VIs are connected to form a bi-directional point-to-point communication channel. VIA does not provide any multicast or broadcast support. A VI consists of two working queues: send queue and receive queue. To each queue there is an associated work notification mechanism called the "doorbell" that notifies the NIC for incoming request. The VI consumer is a software process that communicates through a VI such as an application program or a standard operating system communication facility. The VI consumer posts requests to the queues, in the form of descriptors, to send or receive data. The VI provider handles asynchronously requests, which receive proper status when are finished. The VI Consumer looks up the status of descriptors to verify that messages are correctly sent or received. The VI Consumer can then remove the descriptor or reuse it to send or receive another message. A completion queue allows VI consumers to combine many completion events of multiple VIs into a single queue, which helps the task of event management.

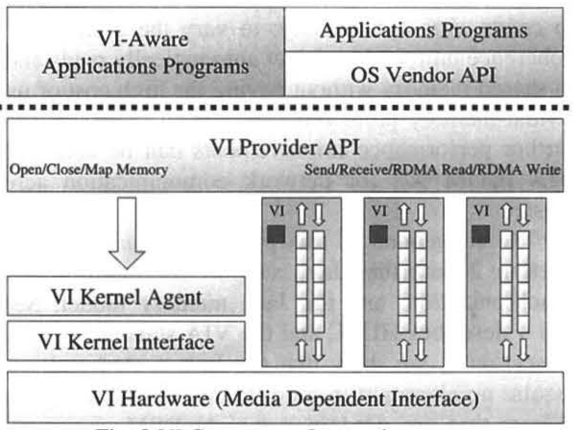

Fig. 2 VI Component Interaction

To directly transfer data between the VI Consumer and NIC without copying data to temporary buffers, the memory must be kept pinned to the same physical memory location until the VI Consumer deregisters the memory. This can become a problem if a program needs more memory than the physical memory available.

VIA supports two types of data transfers: Send-Receive, that is similar to traditional message-passing model, and Remote Direct Memory Access (RDMA), where the source and destination buffers are specified by the sender, and no receiver is required. VIA defines two RDMA operations, RDMA Write and RDMA Read. Figure 2 shows VIs functioning between the application program and the NIC.

We choose VIA to perform protocol communication because: (a) the direct access to the NIC enables low latency communication, which has been shown to improve DSM performance [RAN 00]; (b) data can be directly transferred between the buffers of a VI Consumer and the network without copying any data to or from intermediate buffers, which also helps to improve performance; and (c) the RDMA support can be used to automatically fetch and update regions of memory from/to the page home node.

V. Declaration and Synchronization of Shared Objects

In CoJVM, the declaration and synchronization of objects in the distributed shared memory follows the Java model, since the language specification already includes the concepts required to support the DSM abstraction. This can be exemplified by the concurrent programming model of Java, which assumes the existence of a main memory shared among all threads running on the Java Virtual Machine [LIN 99]. Besides, the language already provides synchronization primitives. The synchronized reserved word permits the definition of blocks or even methods that must be executed in mutual exclusion. The wait method forces an object to wait until a state is reached. The notify and notifyAll methods notify one or all objects, respectively, that some condition was changed. The programmer with the use of these primitives can easily construct a barrier or other synchronization constructs.

Therefore in CoJVM all declared objects are implicitly and automatically allocated into the shared memory. This is achieved transparently through the declaration of the entire Java heap as a shared memory space, in a way similar to Java/DSM [YU 97]. MultiJav [CHE 98] uses a run-time system analysis to automatically detect shared objects. Aleph [HER 99] forces the programmer to use a library to specify what object will be shared.

No extra synchronization primitive is added in our environment. Other environments, such as Java// [CAR 98], centralize synchronization in a special method for each class, the method live. The method forbid (method, condition) is used in Java// for an implicit synchronization, working as a guard, impeding the access to the method method, when the condition condition is true. Titanium [YEL 98] introduces its own barrier instruction.

VI. Architecture and Implementation of DSM

There are two basic alternatives to implement the DSM abstraction on the JVM. The first one is to implement DSM as a library. A software layer is interposed between the JVM and the applications, with the purpose of providing the DSM support, working, therefore, as a middleware. The great advantage of this architecture is its portability: this middleware can be used to add to any standard JVM implementation the DSM abstraction. However, the performance of this architecture could turn its use prohibitive. Aleph and Charlotte [KAR 98] adopt this alternative.

The second alternative is to implement DSM at the JVM level. When compared with the previous alternative, this has the advantage of not adding another software layer to provide the desired functionality. In general, it is expected that the second alternative provides better performance than the first one. The possibility of access to the JVM internal state is also an attractive advantage of the second architecture. However, this architecture is not portable, i.e., each host system (hardware + OS) must have a modified JVM. We choose the second alternative to implement software DSM in the JVM. MultiJav, cJVM [ARI 99a, ARI 99b], and Java/DSM also adopt this basic alternative.

To provide the shared memory abstraction, we decided to use the HLRC (Home-based, Lazy Release Consistency) protocol for two main reasons. First, it scales better than homeless LRC implementations [ZHO 96]. Second, the HLRC implementation already supports the VI Architecture [RAN 00]. However, our implementation of HLRC differs significantly from the base HLRC protocol since our HLRC version does not make use of the virtual memory protection mechanism to detect reads and writes to the shared memory areas in a controllable way. During the *bytecode* interpretation, CoJVM uses the memory access instructions such as putfield and getfield to detect access to shared memory areas that require coherence actions. The advantage of this approach is that we cut off the high costs of using the virtual-memory protection mechanism. In addition, CoJVM can afford HLRC to enforce coherence at small granularity units than the page unit, contributing to reduce or even eliminate false sharing and fragmentation.

By using a bit vector in a way that each bit is associated with a 32-bit of a primitive Java type that is located on the heap, another optimization can be made: every time an object enters a monitor and updates a variable, the bit corresponding to that memory region is set. In this situation, the bit vector reflects exactly the modifications made to the page, so both twin generation and *diff* creation become unnecessary. As a result, the *diff* is automatically generated and corresponds to the positions marked in the bit vector. Note that the memory overhead required for such an approach is negligible and equal to 3% (1/32bits) since 1 bit is spent to mark each 32-bit of a primitive Java type.

TABLE I
EXECUTION TIME OVERHEAD

Application	JDK 1.2.2 (s)	CoJVM (s)	% Difference
Ray-Tracer	95,89	94,41	-1,54
Euller - Size A	642,03	747,15	16,37
FFT - Size A	186,18	188,24	1,10
Alpha-Beta Search - Size A	281,14	294,03	4,58

Table I illustrates the execution time overhead to mark the bit vector for four applications: ray-tracer, provided by Manta [MAN], and Euller, FFT, and Alpha-Beta Search, from JavaGrande Benchmark [JGF]. For two applications, Ray-Tracer and FFT, the difference between JDK (Sun's implementation of Java) and CoJVM is negligible. For Alpha-Beta Search, the difference is under 5%. The bigger difference occurs in Euller: 16%. We believe that this difference can drop, since the code that sets the bit vector is not optimized yet: it is written in C, while the JVM interpreter is written in assembler.

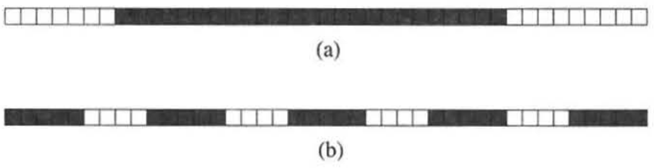

Fig 3. RDMA Write. Shade squares denote modified memory regions. To transfer the modified region in (a) using RDMA, we will need just one message. To transfer the modified region in (b) using RDMA, we will need 5 distinct messages, one per modified memory block, because there are gaps among dirty blocks.

Page transfers are done with the send-receive approach. D*iffs* transfers are done using either send-receive or the RDMA Write provided by VIA. We have to choose between send-receive and RDMA Write because VIA does not provide a Scatter-and-Gather mechanism[1], which would be useful to the *diff* transfer. A latency penalty is imposed if exists more than a dirty contiguous segment per memory block [RAN 00], since it will be necessary to send one message for each memory segment. Figure 3 illustrates the situation. Nevertheless, we can send a memory block that contains all the dirty blocks, minimizing the latency penalty of multiple messages, as figure 4 shows.

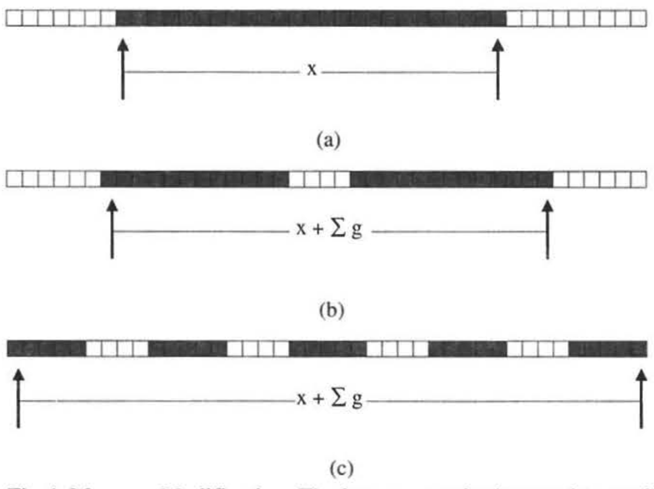

Fig 4. Memory Modification. The latency penalty imposed to send many message with diffs using RDMA could be overcame by sending one message that contains all the modifications (illustrated in b and c). Nevertheless, sending *diffs* with RDMA write is only suitable when the time to send both data and gap is smaller than the time to send diff with send-receive and apply it in the home node.

[1] It is possible to gather data but it is not possible to scatter data in one RDMA write descriptor.

Let x be the total amount of data to be transferred; g the gaps between two distinct blocks; $t_{RDMA}(y)$ the time to transfer y bytes using RDMA write; $t_{SR}(y)$ the time to transfer y bytes using send-receive and $t_A(y)$ the total time to apply y bytes in the memory. It is better to use RDMA Write to automatically update a home node if:

$$t_{RDMA}(x + \Sigma g) \leq t_{SR}(x) + t_A(x)$$

The second part of the equation indicates the time: (a) to transfer the diff from a remote node to the home node; and (b) to apply the diff at the home node. This operation is similar to a scatter-and-gather, so no gap among dirty block areas is transferred.

The bit vector can also be used to choose either *diffs* or RDMA Write to update the home node memory, since its analysis permits us to know exactly the size of the gaps among dirty memory areas.

It is worth to note that CoJVM does not modify the standard semantics of the JVM. In fact, it provides transparency at the programming level, allowing a programmer to practice concurrent programming in the same way as it is done in a single JVM. Thus, no additional effort is required from the programmer to use the proposed environment.

VII. Related Work

In this section we describe some systems that implement software distributed shared memory (SDSM) on Java. A detailed survey on Java for high performance computing, including systems that adopt message-passing approaches, can be found in [LOB 01].

A. MultiJav

One of the main objectives of MultiJav is to maintain the portability of the Java language, allowing its use in heterogeneous hardware platforms. CoJVM does not address this issue, assuming a homogeneous cluster environment. The MultiJav's approach implements the distributed shared-memory (DSM) model into Java by modifying the JVM, but using standard Java concurrency constructs, thus avoiding changing the language definition. The approach is similar to ours. However, sharing is object-based in MultiJav, while we share the primitives Java data-types. MultiJav runtime system, through an analysis of the load/store instructions of the *bytecode* being executed, can automatically detect which objects should be shared, in order to guarantee consistency. This technique seems to be the main contribution of the work. Different threads are permitted to access variables of a same object. Thus a significant amount of false sharing may occur. MultiJav uses a multiple-read / multiple-write protocol to alleviate potential false sharing situations.

B. Java/DSM

Java/DSM under development at Rice University was the first proposal to support a shared-memory abstraction on top of a heterogeneous network of workstations. The main idea behind Java/DSM is to execute an instance of JVM in each machine that participates in the computation by using a system that combines Java portability with TreadMarks [KEL 94], a software DSM library, which enables the JVM to be extended across the network. TreadMarks uses a homeless protocol, while CoJVM adopts a home-based DSM protocol.

Java/DSM is similar to the system presented in the last subsection, except for the changes to Java's semantics. In contrast to that system, the heap is allocated in the shared memory area, which is created with the use of TreadMarks, and classes read by the JVM also are allocated automatically into the shared memory. In this regard, CoJVM adopts a similar to approach.

Java/DSM extends the Boehm and Weiser collector [BOE 88], which is a distinguishing contribution of the work. The garbage collector of each machine maintains two lists; one containing remote references for objects created locally (export list), and other keeping references to remote objects (import list). The lists contain an estimate of the actual cross-machine reference set, which are used only for garbage collection purposes. For most of the time, each machine independently executes the garbage collection, although some synchronization operations are required once a while in order to take care of cyclic structures. CoJVM uses a similar technique, but it is not necessary to create any new structure. Again, the knowledge extracted at run-time from both the JVM and the shared-memory protocol is used to maintain the information necessary to perform garbage collection. The home node must maintain the list of remote nodes that have a copy of its memories blocks to perform the coherence action. This information is equivalent to the export list. The import list is the difference between the total memory and the memory blocks that the node is home for. The Java garbage collection algorithm is adapted to incorporate the notion of local and remote memory blocks.

C. cJVM

cJVM has been developed at IBM Haifa Research Laboratory at Israel. cJVM supports the idea of single system image (SSI) in which a collection of processes can execute in a distributed fashion with each process running on a different node. To implement the SSI abstraction, cJVM uses the proxy design pattern [GAM 95], in contrast to our approach that adopts a release consistency protocol.

In cJVM a new object is always created in the node where the request was executed first. Every object has one master copy that is located in the node where the object is created; objects from the other nodes that access this object use a proxy.

Aiming at performance optimization, during class loading, the associate methods are classified according to the way they access the object fields. Thereafter, the classification helps to choose the most efficient proxy implementation for each method. Three proxy types are supported: (a) standard proxy that transfers all the operations to the master copy; (b) read-only proxy that applies the operations locally, based on the fact that it is guaranteed to access only fields that never change, so the proxy can replicate and maintain these fields; and (c) proxy that locally invokes methods without state, since these are methods that do not access object fields. Although this classification is done dynamically, at class loading time, the information it uses is static. So another difference between CoJVM and cJVM is that we use data available at run-time in the JVM.

cJVM modified the semantics of the new *opcode*, allowing the creation of threads in remote nodes. CoJVM does not modify the semantics nor the syntax of any Java *opcode*. If the parameter for the new *opcode* is a class that implements Runnable, then the new *bytecode* is rewritten, as the pseudo *bytecode* remote new. This pseudo *bytecode*, when executed, determines the best node to create a new Runnable object. A pluggable load balancing function makes the choice of the best node. We do not treat load balancing for two reasons: (a) scientific applications usually executes in a exclusive fashion, in a way that the CPU is not shared with other applications; so load balancing is not necessary; and (b) for commercial applications, we think that a good designer or initial configuration is more effective in improving performance than dynamic load balancing.

VIII. CONCLUSION

In this work we introduce CoJVM, a cooperative JVM that addresses in a novel way several performance aspects related to the implementation of distributed shared memory in Java. We showed that CoJVM complies with the Java language specification while supporting the shared memory abstraction as implemented by our customized version of HLRC, a home-based software DSM protocol. The main difference between CoJVM and current DSM-based Java implementations is that it takes advantage of the run-time application behavior, extracted from the JVM, to reduce the overheads of the coherence protocol. More specifically, a specialized run-time JVM machinery (data and support mechanisms) is used to create *diffs* dynamically, to allow the use of smaler data coherence units, and to detect automatically reads and writes to the shared memory, without using the time-expensive virtual-memory protection mechanism. Moreover, CoJVM uses VIA as its communication protocol aiming to improve Java application performance even further.

REFERENCES

[ARI 99a] ARIDOR, Y.; FACTOR, M.; TEPERMAN, A. *cJVM: a Single System Image of a JVM on a Cluster.* International Conference on Parallel Processing 99 (ICPP 99), September 1999.

[ARI 99b] ARIDOR, Y.; FACTOR, M.; TEPERMAN, A. *cJVM: a Cluster Aware JVM.* Proceedings of the First Annual Workshop on Java for High-Performance Computing in conjunction with the 1999 ACM International Conference on Supercomputing (ICS), Rhodes, Greece, June 1999.

[ARN 96] ARNOLD, K.; GOSLING, J. *The Java programming language*, First Edition. Addison-Wesley, 1996.

[BAR 98] BARATLOO, A.; KARAUL, M.; KARL, H; KEDEM, Z. *An Infrastructure for Network Computing with Java Applets.* In Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing; Palo Alto, California, USA, February 1998.

[BOE 88] BOEHM, H.; WEISER, M. *Garbage Collection in an Uncooperative Environment.* Software: Practice and Experience 1988; 18(9):807-820.

[CAP 97] CAPPELLO P.; CHRISTIANSEN B.; IONESCU M.; NEARY M.; SCHAUSER K.; WU, D. *Javelin: Internet-Based Parallel Computing Using Java.* Concurrency: Practice and Experience 1997; 9(11):1139-1160.

[CAR 98] CAROMEL D.; KLAUSER W.; VAYSSIRE J. *Towards Seamless Computing and Metacomputing in Java.* Concurrency: Practice and Experience 1998; 10(11-13):1043-106.

[CHE 98] CHEN, X.; ALLAN, V. *MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines.* The 1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98), Las Vegas, Nevada, USA, July 1998.

[COM 97] Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0.* http://www.viarch.org. Accessed on December, 25.

[FOX 96] FOX, G.; FURMANSKI, W. *Towards Web/Java Based High Performance Distributed Computing - an Evolving Virtual Machine.* In Proceedings of the 5 th . IEEE Symposium on High Performance Distributed Computing; 1996.

[JGF] Java Grande Forum. http://www.javagrande.org. Accessed on December, 25.

[GAM 95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.; BOOCH, G. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GUP 00] GUPTA, M.; MIDKI, S.; MOREIRA, J. *High Performance Numerical Computing in Java: Compiler, Language and Application Solutions.* Tutorial at the ACM / IEEE Supercomputing; Dallas, USA, November 2000.

[HER 99] HERLIHY, M.; WARRES, M. *A Tale of Two Directories: Implementing Distributed Shared Objects in Java.* In ACM 1999 Java Grande Conference, Palo Alto, California, USA, June 1999.

[HOA 74] HOARE, C. *Monitors: An Operating System Structuring Concept*, Communications of the ACM, 12(10), October 1974.

[KAR 98] KARL, H. *Bridging the Gap Between Distributed Shared Memory and Message Passing.* In ACM 1998 Workshop on Java for High Performance Network Computing. Palo Alto, California, USA, February 1998.

[KEL 92] KELEHER, P.; COX, A.; ZWAENEPOEL, W. *Lazy release consistency for software distributed shared memory.* Proceedings of the Nineteenth International Symposium on Computer Architecture, pages 13-21, May 1992.

[KEL 94] KELEHER, P.; DWARKADAS, A.; COX, A.; ZWAENEPOEL, W. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* Proceedings of the 1994 Winter Usenix Conference, pp.115-131, January 1994.

[LI 89] LI, K.; HUDAK, P. *Memory coherence in shared virtual memory systems.* ACM Transactions on Computer Systems, 7(4):321--359, November 1989.

[LIN 99] LINDHOLM, T.; Yellin, F. *The Java Virtual Machine Specification.* Second edition. Addison-Wesley, 1999.

[LOB 01] LOBOSCO, M.; AMORIM, C.; LOQUES, O. *Java for High-Performance Computing.* Technical Report RT-01/01. Available at http://www.caa.uff.br/reltec.html.

[MAN] Manta: Fast Parallel Java. http://www.cs.vu.nl/manta/.

[RAN 00] RANGARAJAN M.; IFTODE L. *Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance.* In Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10-14, 2000, Atlanta, Georgia, USA.

[SUN 99] Sun Microsystems. *The Java Hotspot TM Performance Engine Architecture.* http://java.sun.com/products/hotspot/whitepaper.html. Accessed on December, 25.

[ZHO 96] ZHOU, Y.; IFTODE, L.; LI, K. *Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems.* Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, October 1996.

[YEL 98] YELICK, K. et al. *Titanium: A High-Performance Java Dialect.* In ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California, February 1998.

[YU 97] YU, W.; COX, A. *Java/DSM: a Platform for Heterogeneous Computing.* ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.