Kaffemik: Supporting a distributed JVM on a single address space architecture

Johan Andersson[*], Stefan Weber[*], Emmanuel Cecchet[†], Christian Jensen[*]

[*]Distributed Systems Group
Trinity College Dublin
Dublin, Ireland

Johan.Andersson@cs.tcd.ie
Stefan.Weber@cs.tcd.ie
Christian.Jensen@cs.tcd.ie

[†]SIRAC
INRIA Rhône-Alpes
Grenoble, France

Emmanuel.Cecchet@inrialpes.fr

Abstract

Java is increasingly used to develop large server applications. In order to provide powerful platforms for such applications a number of projects have proposed Java Virtual Machines (JVMs) that are based on network of workstations. These JVMs employ the message-passing paradigm, i.e. all communication between the distributed instances of the virtual machine take place using remote method invocation (RMI) or socket stream communication.

The JVM of Kaffemik is based on multiple instances of a JVM that communicate using the shared memory paradigm. All objects that are created by virtual machines are held in a shared heap that is supported by an underlying shared memory system. By exploiting the single address space abstraction it is possible to avoid overheads that are inherent to message passing.

This paper presents our first experiences with a preliminary implementation of Kaffemik. It gives a detailed discussion of the design and implementation decisions and shows a number of performance measurements that demonstrate the advantage of our shared-memory approach.

1 Introduction

Java is rapidly becoming the object-oriented language of choice in both academia and industry. Moreover, Java is no longer confined to embedded systems and client-side computing (applets), but is increasingly used in server applications, e.g., the World Wide Web Consortiums (W3C) latest web-server, Jigsaw [4], is entirely implemented in Java.

Cheap and scalable computer architectures are increasingly developed using a network of inexpensive workstations [5]. The Java execution model facilitates collaboration among virtual machines executing on this type of cluster through both function shipping (remote method invocations) and data shipping (object mobility). However, these mechanisms are made explicit in the programming model, which complicates the development of server applications on collaborating virtual machines.

A standard implementation of an application server for Java on a cluster of common off the shelf (COTS) machines requires the application to be partitioned among the nodes in the cluster. Objects are made available in the memory of one node in the cluster and can be accessed through remote invocations, e.g. Sun's RMI [6]. Thus, in order to access an object on a remote node, the system has to translate single address space Java references into the multiple address spaces used on the different nodes in the system, this translation is known as swizzling [7]. Furthermore, objects passed as parameters generally have to be converted to an exportable format or serialized before they are transferred to the other node. Serialization is performed automatically by RMI.

Single address space architectures (SASA) [8,9,10, 11] provide the abstraction of a single shared virtual memory, i.e., an object is available on the same virtual address on all nodes in the system. This has the advantage that virtual addresses can be used as a system wide unique object identifiers.

Kaffemik is a scalable distributed JVM implemented on SciOS/SciFS [12]**,** a single address space architecture developed for SCI networks at INRIA Rhône-Alpes. The SCI network is a standard interconnect based on the Scalable Coherent Interface [3]. The single address space allows Kaffemik to offer the abstraction of a single shared object space across all nodes in the system. Among the advantages of a single shared object space are: it simplifies the development of multithreaded applications by eliminating the need for separate communication libraries (RMI, CORBA, etc.), it eliminates the overheads of swizzling and serialization because objects can be accessed directly in the main memory of a remote node and it allows programmers to cache objects in the memory of idle nodes. Also, caching is practical because access to objects in remote memory is faster than loading an object from local disk [5]. Caching objects in the memory of remote nodes means that the size of the application's working set can be as large as the sum of the physical memory of all the nodes in the system.

Kaffemik supports automatic and transparent distribution of objects; objects automatically migrate to the node where they are used. The location of execution is not transparent, i.e., the programmer has to specify the executing node when a thread is created, which gives some indirect control over where objects are located. When a thread invokes a method on an object located in remote memory, SciOS/SciFS migrates the page containing the object into local memory. This involves swapping memory pages if no local pages are free. Objects in remote memory are referenced with local object references and local object references can be passed as parameters in method invocations. Thus, neither swizzling nor serialization is required.

We have compared the performance of method invocations on remote objects in our single address space architecture with remote method invocations among collaborative virtual machines. RMI is used to swizzle and serialize among the different address spaces. This evaluation shows that remote method invocations are at least an order of a magnitude faster using Kaffemik, than it is amongst two virtual machines using standard RMI.

The rest of this paper is organized in the following way. Section 2 gives an overview of the Kaffemik architecture. Section 3 describes the implementation of the current Kaffemik prototype. In section 4, we present an evaluation that compares method invocations across multiple address spaces, realized using RMI, with Kaffemik. We present a comparison of our approach with related work in Section 5. Finally, our conclusions and directions for further work are presented in Section 6.

2 Design of Kaffemik

The design of Kaffemik has simplicity as its first priority. This simplicity finds its expression in two characteristics of Kaffemik. The first characteristic is Kaffemik's support of Java's standard API. This makes it possible to execute applications that have been developed on stand-alone JDKs on the distributed JVM. The second characteristic is Kaffemik's simple design of a heap that is shared by all nodes of a cluster. The heap and subsequently all objects that are created in the heap are accessible by all nodes.

Java defines the behaviour of memory accesses, synchronization and threads. Their definition is documented in the language specification [13], the virtual machine specification and the API. Kaffemik extends this definition by a set of method calls. These method calls expose distribution characteristics to the developer, primarily in the Thread class. However, all APIs that are defined by Java are supported in order to comply with the Java API definition and be able to run standard Java applications.

Kaffemik is designed as a set of collaborative JVMs, which offer a single object space abstraction to the application. The design of the collaborative JVM is based on the freely available Kaffe Virtual Machine. The virtual machine (VM))is extended into a distributed JVM running on top of a single address space architecture. In principal, the architecture of Kaffemik consists of two parts - a distributed shared memory infrastructure and a distributed Kaffe Virtual Machine, see figure 1.

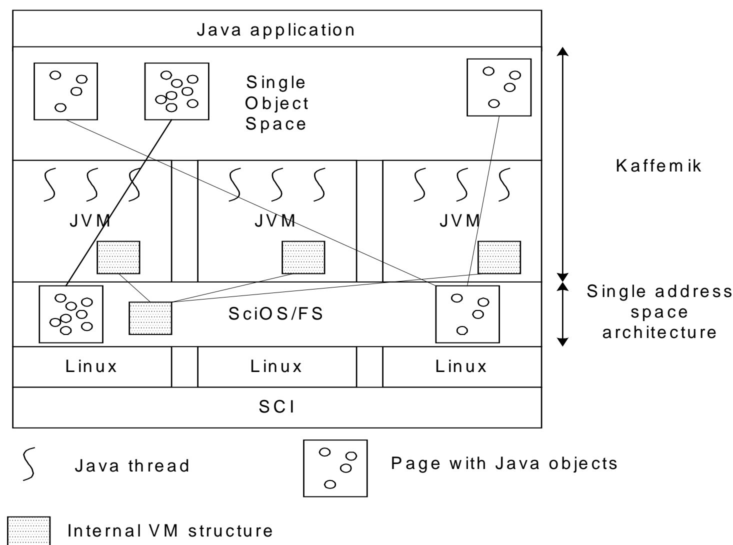

Figure 1. The architecture of Kaffemik

Kaffemik relies on a single address space architecture to provide for the single object space. The single object space forms a monolithic heap over all nodes in the cluster. Kaffemik allocates the monolithic heap from the single address space. All objects created by the Java application will be allocated from the monolithic heap. Because of the single address space architecture, every object is available at the same virtual address on all nodes in the system. Both the VM and Java applications allocate objects from the monolithic heap. Internal VM structures, e.g. internal hash tables, are also shared between the nodes as shown in figure 1, but these structures are allocated directly from the single address space.

We have also redesigned the thread model. Kaffemik's thread model allows the application programmer to start threads at specific nodes within the cluster. Wait and notify are enhanced to support inter-node thread synchronization.

3 Kaffemik implementation

The implementation of Kaffemik is based on two components: Kaffe and SciOS/SciFS. Kaffe [14] is an open-source, clean room implementation of Sun's Java environment. The sources of Kaffe are released under the Gnu Public License (GPL). The virtual machine of Kaffe is used as a basis for the implementation of Kaffemik's distributed virtual machine. SciOS/SciFS is an open-source shared memory abstraction developed by INRIA Rhône-Alpes. The following two subsections describe Kaffe and SciOS/SciFS respectively. The concluding subsection will detail our implementation of Kaffemik based on the two packages.

3.1 Kaffe

The Kaffe Virtual Machine environment consists of a virtual machine (VM), a compiler and a set of supporting class packages. The overall structure of the virtual machine is depicted in figure 2. The VM consists of three levels of components. The base level contains components for memory management, synchronization management, thread management and a subsystem for calls to native code. The second level comprises of an execution engine. The top level holds a code verifier and a component for class management.

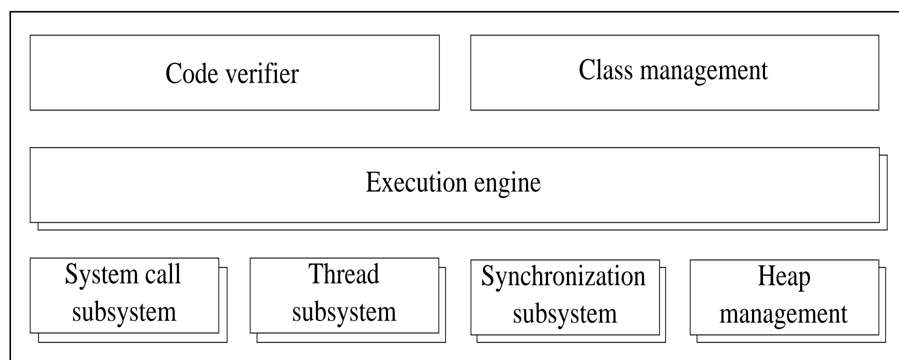

Figure 2: Components of the virtual machine

The components of the virtual machine are loosely coupled with each other. Each component has a well-specified interface. An implementation of a component of the base level maps the functionality that is promised by the interface to the underlying system, which can be of type library, operating system, or another component. This modular

approach allows the exchange of one implementation of a component for another. The flexibility facilitated by the approach is harnessed to provide implementations for different hardware platforms and operating systems. The existing implementation of Kaffe provides a variety of implementations of thread and synchronization components. These implementations map the system to platforms such as Win32, BeOS, Unix-PThreads and Unix System V.

The components of the base level are important for the distribution of the functionality of the virtual machine. The heap management defines functionality for the allocation and release of objects. The thread management allows the creation, administration and termination of threads of execution. The synchronization subsystem implements methods for concurrency control. The native subsystem provides an interface to system calls of the underlying platform.

The memory management component implements heap management and a garbage collector. The heap management is responsible for the allocation of memory resources in order to store objects. All objects that are loaded into the virtual machine and most of the administrative structures of the virtual machine are located in the heap.

The heap is organized in pages. The size of these pages generally matches the size of pages of the underlying (operating) system. The heap management keeps a record with information about the contents and state of each page. These records are kept partly in an array allocated in the data segment of the virtual machine and partly at the beginning of the individual memory pages. The information describes the size and the number of objects stored in a memory page. Each page can contain only objects of the same size. An additional list is kept that holds links to pages that hold objects of the same size, but not are entirely filled. When an object is created, the heap management tries to allocate memory from these partially filled pages. If the object does not fit in any of these pages, a free page is allocated from the heap. If no more pages are available on the heap, the garbage collector is invoked.

The garbage collector of the virtual machine is a non-incremental, non-generational, conservative mark-and-sweep collector with a Boehm-like allocator. If an allocation fails, i.e., there are no free pages left, the garbage collector is invoked. The garbage collector uses heuristics to determine if it is cheaper to run a garbage collection or to allocate more memory for the heap. If the heap has already the maximal defined size a garbage collection is performed, otherwise more memory for the heap is allocated.

The thread and synchronization management components are based on the underlying platform. The execution engine implements a thin layer that provides platform independent thread and synchronization primitives. This layer interfaces with the underlying platform dependent components. Kaffe provides a set of components for different platform including Unix-PThreads, Linux-Threads, Win32, BeOS-Native and Unix-JThreads.

3.2 SciOS/SciFS

The SciOS/SciFS[1] prototype implements a distributed shared memory system on a SCI cluster of Intel PCs with Linux 2.0 or 2.2 kernels and Dolphin's 32-bit (D310) or 64-bit (D321) PCI-SCI adapters [15]. SciOS and SciFS are both implemented as kernel modules. Figure 3 shows the SciOS/SciFS architecture and how it interfaces with Kaffemik.

---

[1] The SciOS/SciFS prototype sources and documentation are freely available for download from http://sci-serv.inrialpes.fr.

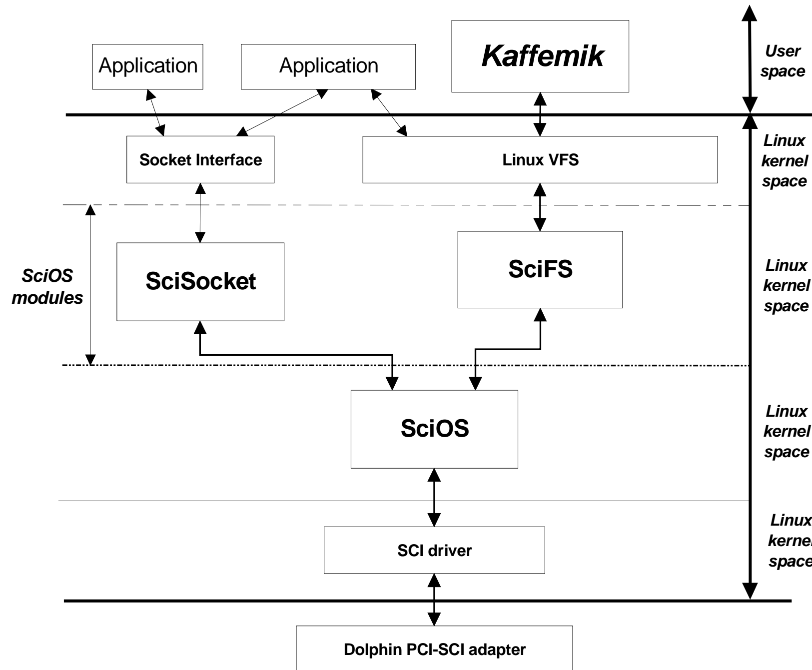

Figure 3: SciOS/SciFS architecture

SciFS implements the actual DSM, providing memory management and coherency protocols. SciFS relies on a lower layer, SciOS, which is also developed at INRIA Rhône-Alpes. The SciOS layer is based on Dolphin's PCI-SCI adapters [15] and offers basic services for SCI clusters, such as messages, remote procedure calls and physical memory management.

SciFS is implemented as a Linux distributed file system and interfaces to the Linux Virtual File System (VFS) facility. Since the file system is interfaced with the file mapping mechanism, SciFS allows for tight integration of SCI with the operating system's virtual memory system. The main abstraction in SciFS is memory mapped files. Shared memory segments are presented to the user as files and are accessed by using file operations such as the *open*, *mmap*, and *close* system calls. A process can open a file, map it in its virtual address space and use normal load and store instructions on the mappings thereby reading and modifying the file's contents. Multiple processes, possibly on different nodes, that open the same file and map it in their address space, share the data contained in the file.

3.3 Kaffemik

Enabling a distributed JVM to share data requires a number of data structures be allocated in the single address space to allow collaboration among the participating nodes in the cluster. In the following subsections we describe how SciOS/SciFS is incorporated with Kaffemik, and how the Kaffe VM is extended to support sharing of data, remote thread creation, and inter-node synchronization. At this point we have not fully incorporated the garbage collector component in Kaffemik.

Kaffemik and SciOS/SciFS

SciFS is used to supply Kaffemik with a single address space architecture. Kaffemik uses the SciFS file system to create, to map (i.e. share), and destroy the shared memory segments. The single object space that comprises Kaffemik's monolithic heap is one of the shared memory segments allocated in the single address space. Besides the monolithic heap, Kaffemik shares cluster meta-data and internal VM structures.

Figure 4 shows how every Kaffemik on each node maps and share all the information contained in the SciFS DSM.

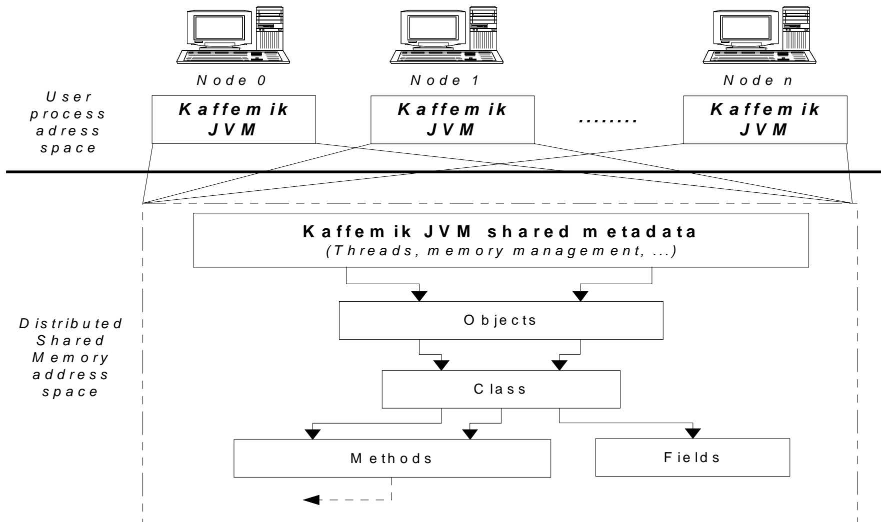

Figure 4: Kaffemik's mapping of DSM space

To ensure data coherency, Kaffemik uses the locking mechanisms provided by SciFS. When a JVM needs to access data, it performs a regular memory reference and lets the DSM migrate, replicate or access remotely the information in the right physical memory. The data distribution and placement is transparent to Kaffemik.

Cluster meta-data

Meta-data describing the nodes in the cluster must be shared. This meta-data contains information about how many nodes there are in the cluster and the node id for each node. Since SciOS/SciFS abstracts shared memory segments as files, information about the file descriptors and the offsets into the processes' virtual address space where the files must be mapped, have to be shared amongst the distributed JVMs.

Heap component

To support a distributed JVM with a monolithic heap requires sharing of the virtual machine's heap component structures. Kaffemik's heap management component currently maintains the same structures as Kaffe does. The meta-data Kaffe uses to describe the heap, and the freelists used to manage partially allocated pages, are shared in separate memory

segments. These segments are not allocated from the monolithic heap. Sharing the meta-data and the freelists enables all Kaffemik instances in the cluster to allocate objects from the monolithic heap.

Internal VM structures

Kaffe allocates a number of internal structures, such as hash tables for UTF-8 constants and strings to support management of classes loaded by the VM.

Kaffe allocates two hash tables to manage references to String constants and UTF-8 constants. These hash tables, like the UTF-8 and String objects are allocated on the heap. References to the objects are then inserted into the hash tables. Kaffemik separates the hash tables for the strings and UTF-8 constants from the heap and shares them in separate memory segments. This is necessary, because otherwise they are not guaranteed to be allocated at the same addresses when a new Kaffemik is started in the cluster, and also because Kaffe uses UTF-8 constants to lookup references to classes and methods. Even though these hash tables are allocated in separate memory segments, the UTF8 and string objects are still allocated on the monolithic heap with the references inserted into the shared structures.

Kaffe's internal types (e.g. char, int, and float) are also allocated on the heap. Kaffemik shares the internal types in a separate memory segments.

A class pool manages references to classes that have been loaded by Kaffe. This class pool is represented as a hash table, but it is allocated outside the heap. Kaffe allocates classes on the heap, and puts the references in the class pool. Kaffemik allocates the class pool in a shared segment, and maintains the references to loaded classes exactly as Kaffe.

Remote threads

Kaffemik supports thread creation on remote nodes. A Java application can spawn off threads on a remote Kaffemik instance, and then the remote Kaffemik instance will start the thread. To support remote thread creation, Kaffemik extends the Java API with a ThreadQueue thread. During the initialization of the VM, each Kaffemik instance starts a ThreadQueue thread. This thread is added to a globally shared thread queue array, which contains one ThreadQueue thread for each instance of Kaffemik in the cluster. When the ThreadQueue thread is started, it waits for threads to be added to its local queue. As soon as a thread is added to the local queue, the ThreadQueue removes the thread from the queue and invokes the thread's start() method.

To support the application programmer with the mechanism of starting threads at remote nodes Kaffemik extends the Thread class with a new method called `Thread.startAt(nodeid)`. The application programmer starts a thread on a remote node by calling `startAt` with the node id for the node on which the thread is going to be started. This adds the thread to the ThreadQueue on the corresponding Kaffemik instance.

Wait and notify

Kaffemik provides a wait and notify mechanism between threads running at different nodes. When one node's thread releases a lock, it notifies a waiting thread, possibly waiting at another node. To avoid the problem of resuming this thread at the same node from which

the notify originates, due to Kaffe's thread model [14], we create a remote thread space shared amongst all nodes in the cluster. The remote thread space is an array with entries for every node in the cluster. Kaffe's original threads are extended with a node id, which is set to the id of the node where the thread starts. When the thread holding the lock is about to resume the next thread waiting for the lock, it inspects the node id of the thread and if this thread originates at another node, the thread is put in the corresponding entry of the remote thread space. The node waiting for the lock polls its entry in the remote thread space and resumes the thread polled from the remote thread space.

4 Kaffemik evaluation

This evaluation uses Intel P-III 800MHz PCs equipped with 256MB of RAM, Linux 2.2.14, interconnected with 64-bit PCI-SCI adapters (D321). The PCI-SCI adapters are manufactured by Dolphin Interconnect Solutions [14].

The evaluation measures the costs associated when a JVM has to cross multiple address spaces in order to access and modify data, and compare this with a single address space approach. The multiple address space approach is realised using Kaffe v1.0.6 with RMI and the single address space approach is realised using Kaffemik.

4.1 Multiple address spaces vs. Single address space Experiment

A matrix is used to measure the costs of crossing address space boundaries. The size of the matrix is 320x320. It is filled with 32-bit integers and occupies exactly 100 pages of memory as Intel based Kaffemik uses a page size of 4096 bytes. A matrix of this size can for example represent a database object in an object-oriented database, a CAD drawing or a web page with graphics in a web server application. These scenarios correspond to the areas where Kaffemik can be applied.

Kaffe with RMI (local)

The matrix is created on the client and passed as an argument in a remote method invocation to the server. At the server side the matrix is modified. Modifying the matrix is analogous to updating the data of a database object or assembling a web page. When the server is finished with the modification it sends the matrix back to the client. The test measures the time it takes to modify the matrix. Table 1 shows the result of this test.

Touch (rows, columns)	% of matrix	Processing time (µs)
(0,0)	0	4
(80,320)	25	112
(160,320)	50	430
(240,320)	75	1046
(320,320)	100	1934

Table 1: Kaffe with RMI

The touch column indicates how many rows/columns of the matrix are modified. It can be thought of as the time it takes to update a certain percentage of for example a database object. The processing time is the time Kaffe spent on modifying the matrix. Both the

server and the client run locally, because we do not have a RMI implementation for SCI. This avoids penalising Kaffe RMI with the overheads of a slower local area network.

Kaffemik (two nodes)

In this case a client thread running at one of the two nodes creates the matrix. The matrix is the passed to a server thread running at the other node using a shared buffer. The server modifies the matrix, and puts it back into the buffer, and notifies the client thread, which then gains control over the matrix. The processing time is the time it takes for the server to modify the matrix. Table 2 shows the results from running Kaffemik on two nodes.

Touch (rows, columns)	% of matrix	Processing time (μs)
(0,0)	0	6175
(80,320)	25	30270
(160,320)	50	53800
(240,320)	75	79100
(320,320)	100	102195

Table 2: Kaffemik (two nodes)

The processing time is considerably higher compared to Kaffe RMI. This is because of the overhead induced by page migration between the nodes. When optimised, SciOS/SciFS should be able to migrate one page in 125μs [12]. However, we currently use SciOS/SciFS with strict consistency, and with a number of flags activated for debugging information, timers, and internal sanity checks within SciOS/SciFS, which all slow down page migration. Moving to lazy release consistency, optimising the internal structures and deactivating the flags should lower processing time, but this assumption has not been tested.

Total execution time

A separate measurement looks at the total time taken from the point where the client initiates the request until it regains control over the matrix. In the RMI case, this time is measured from the point where the client invokes a remote method on the server passing the matrix as an argument. In Kaffemik, this time is measured from the point when the client puts the matrix in the shared buffer until the server notifies the client and the matrix is removed from the buffer. The results are displayed in the table 3. Efficiency gain is the result of dividing the total time of Kaffe with RMI with Kaffemik's total time.

Touch (rows, columns)	Kaffe RMI (μs) (local)	Kaffemik (μs) (2 nodes)	Factor
(0,0)	64252713	968181	66.31
(80,320)	64215334	968366	66.31
(160,320)	64231220	968267	66.34
(240,320)	64179276	964340	66.55
(320,320)	64162860	967653	66.31

Table 3: Total execution time

Using a single address space approach offers superior performance. Kaffemik is around 66 times faster that Kaffe with RMI. Kaffe with RMI offers very poor performance. This poor performance is due partly on the serialisation of the matrix and partly on an inefficient implementation of RMI[2]. However, Kaffemik should offer even better performance. The main reason why better performance is not obtained with Kaffemik stems from the inter-node wait/notify implementation. Acquire and release of locks consumes too much time. This is major concern for us since it hampers the scalability of Kaffemik, and is subject for redesign.

4.2 Discussion

Crossing address space boundaries is far more inefficient than using a single address space approach. In terms of overall performance, Kaffemik offers an extensive performance gain compared to Kaffe with RMI. However, there are problems related to SciOS/FS. The time spent on page migration cannot be neglected. This however can be addressed by optimising SciOS/SciFS' page migration schema. For example, by using lazy release consistency and optimising the size of SciOS/SciFS' internal page tables and file tables, we expect to achieve better performance. The main concerns derive from the implementation of the inter-node wait and notify mechanism, which must be redesigned to achieve better scalability.

5 Related work

Yu and Cox propose in Java/DSM [17] a virtual machine implementation on a software DSM system. Java/DSM employs TreadMarks [18] as underlying DSM system. TreadMarks implements a shared memory abstraction as a user level library using System V system calls. The motivation of this project - similar to the one presented in this paper – was to hide the distributed execution completely from the developer and exploit at the same time the ease of use of the shared memory paradigm.

Java/DSM, in contrast to Kaffemik, assumes that communication between nodes is expensive. This assumption leads to a design that avoids communication. Every node employs its own garbage collector and a great number of structures are replicated on every node of the system. This design leads to a higher consumption of resources and to a higher complexity of the algorithms used inside the virtual machine. This can be avoided by sharing structures of the virtual machine and exploiting an underlying shared address space abstraction.

KaffeOS [19] by Back et al proposes an integration of operating system characteristics into the JVM. The main characteristic that is considered in this implementation is the abstraction of processes. Every process (or application) requires its individual address space to provide a secure separation from other processes. In Java, the address space is represented by the heap, in which all objects of an application are accommodated. The virtual machine of KaffeOS maintains a kernel heap, individual heaps per process and shared heaps for sets of processes. The kernel heap holds objects that are needed by the virtual machine; individual heaps hold objects that are specific for an application and shared heaps hold objects that are accessible by one or more applications.

The implementation of KaffeOS is a stand-alone implementation. Applications that are executed as processes on the same JVM can communicate through shared heaps without the

[2] We have compared Sun's RMI with Kaffe's RMI and Sun's RMI is substantially faster.

need of serialization. The communication of applications on separated JVMs relies on a message passing mechanisms such as RMI or socket communication. This communication still requires serialization.

Sirer et al propose with Kimera [20] a distributed virtual machine. The distribution in this project concentrates on the distribution of security services such as byte-code and parameter verification. Services are separated into static and dynamic services. Static services that can be performed before the execution of an application are located at a central server. Dynamic services such as parameter verification have to be performed at execution time and hence are located at the clients. The fixation of static services on a central server removes a number of tasks from virtual machines on clients and makes these lighter than common virtual machines. This approach suffers from the separation of address spaces on the clients. Applications that run on the clients have to rely on communication over RMI and have to serialize data that is transferred between the clients.

6 Conclusions and future work

In this paper we presented our ongoing work on a distributed Java virtual machine. The issue that we addressed is the communication among virtual machines in a cluster. The proposed solution evolves around a set of JVMs that share a heap. The sharing is facilitated by a component that provides a single-address space abstraction. We evaluated our design by comparing the costs that arise when the communication of clustered JVMs has to cross multiple address space boundaries with the cost for communication in our single address space approach. Our experiments showed that the exploitation of the single-address space abstraction gives the virtual machines extensive performance advantages in comparison to systems that rely on message passing.

References

[1] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. In Proceeding of *Euro-Par 2000: Parallel Processing*, August 2000.

[2] Y. Aridor, M. Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the 1999 IEEE Internation Conference on Parallel Processing (ICPP-99)*, September 1999.

[3] IEEE Std 1596-1992. IEEE Standard for Scalable Coherent Interface (SCI) *The Institute of Electrical and Electronics Engineers, Inc.*, 1993.

[4] World Wide Web Consortium. W3C's Java Server. http://w3c.org/jigsaw , 2000.

[5] M. M. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principals (SOSP'95)*, December 1995.

[6] Sun Microsystems. Java™ Remote Method Invocation (RMI). http://www.java.sun.com/products/jdk/rmi/ , 2000.

[7] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge

addresses on standard hardware.
In 1992 International Workshop on Object Orientation in Operating Systems, Dourdan, France, 1992. IEEE, IEEE Computer Society Press.

[8] J. Chase, H. M.Levy, E. Lazowska, and M. Baker-Harvey.
Opal: A Single Address Space System for 64-Bit Architectures.
In *Proceedings of IEEE Workshop on Workstation Operating Systems*, April 1992

[9] K. Murray, T. Wilkinson, P. Osmon, A. Saulsbury,T. Stiermerling, and P. Kelly.
Angel: Resource Unification in a 64-bit Micro Kernel.
In *Proceedings of th 27th Hawaii International Conference on System Science*, September 1993.

[10] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo.
Mungi: A Distributed Single Address-Space Operating System.
In *Proceedings of the 17th Australasian Computer Science Conference*, January 1994.

[11] P. Déchamboux, D. Hagimont, J. Mossière, and X. Rousset de Pina.
The Arias Distributed Shared Memory: An Overview.
23rd Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, Nov. 1996.

[12] Povl T. Koch, E. Cecchet, and X. Rousset de Pina.
Global management of coherent shared memory on an SCI cluster.
In *Proceedings of SCI Europe'98*, September 1998.

[13] Tim Lindholm and Frank Yellin.
The Java Virtual Machine Specification.
Addison Wesley, Reading Massachsetts, 1996.

[14] T. Wilkinson.
Kaffe: A java virtual machine.
http://www.kaffe.org, 1996.

[15] Dolphin Interconnect Solutions.
PCI-SCI cluster adapter specification. May 1996.

[16] Gabriel Antoniu, Luc Bougé, and Raymond Namyst.
Generic distributed shared memory: the DSM-PM2 approach.
Research Report RR2000-19, LIP, ENS Lyon, Lyon, France, May 2000.

[17] W. M. Yu and A. L. Cox.
Java/DSM: A platform for heterogeneous computing.
In *Proc. of Java for Computational Science and Engineering-Simulation and Modeling Conf.*, June 1997.

[18] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel.
Treadmarks: Shared memory computing on networks of workstations.
IEEE Computer, February 1996.

[19] Godmar Back, Wilson C. Hsieh, and Jay Lepreau.
Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.
In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[20] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad.
Design and implementation of a distributed virtual machine for networked computers.
In *Proceedings of the 17th ACM Symposium on Operating System Principals (SOSP'99)*, December 1999.