THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-02-04

dJVM - A distributed JVM on a Cluster

John N. Zigman and Ramesh Sankaranarayana

September 2002

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-01-02 Jeremy E. Dawson and Rajeev Gore. *Mechanising cut-elimination for display logic*. October 2001.

TR-CS-01-01 Stephen Roberts Peter Christen, Markus Hegland and Irfan Altas. *A scalable parallel fem surface fitting algorithm for data mining*. October 2001.

TR-CS-00-02 Peter Strazdins. *A survey of simulation tools for cap project phase iii*. October 2000.

TR-CS-00-03 Bill Clarke. *Sparc v9 instruction set specification*. October 2000.

TR-CS-00-01 Jens Gustedt, Ole A. Maehle, and Jan Arne Telle. *Java programs do not have bounded treewidth*. February 2000.

TR-CS-99-02 Samuel Taylor. *A distributed visualisation tool for digital terrain models*. July 1999.

dJVM - A distributed JVM on a Cluster

John N Zigman and Ramesh Sankaranarayana

CAP Research Group, Department of Computer Science
Australian National University, Canberra
ACT 0200, Australia

John.Zigman@cs.anu.edu.au, Ramesh.Sankaranarayana@anu.edu.au
http://cap.anu.edu.au/cap/projects/dJVM/

Abstract. dJVM provides a distributed Java Virtual Machine (JVM) on a cluster. It hides the distributed nature of the underlying machine from a Java application by presenting a *single system image* (SSI) to that application. dJVM is based on the Jikes RVM [1] (a JVM written in Java), and is the first distributed implementation of the Jikes RVM. This provides a framework for exploring a range of distributed runtime support algorithms on large clusters. This report describes the dJVM architecture and some of the implementation details.
Key words: cluster, Java, Java Virtual Machine, single system image.

1 Introduction

Java is widely used in the industry for object oriented application development. The main reasons for this are the portability of Java programs, resulting from a clearly defined Java Virtual Machine [8], and the performance speed-ups achieved through improvements in just-in-time (JIT) compilers. A lot of the application development using Java is on the server side.

Server applications are typically multi-threaded, with limited interaction between threads servicing different clients. Scalability and performance are two important issues with such applications. Clusters of commodity hardware can provide a cheap solution to both of the above issues. However, to facilitate the use of such hardware without additional programming complexity, it is necessary to provide an abstraction that efficiently uses the distributed nature of the hardware, while at the same time presenting a unified view of the system (a Single System Image or SSI) to the outside world. This allows a programmer to concentrate on the task of writing programs without having to address issues of distribution.

There are many projects working on solving this problem. The approach taken by them to provide an SSI can be broadly divided into three categories:

1. *Provide an implementation above the JVM.* Typically, by transforming the Java program from the non-distributed form into a form that incorporates the bytecode to implement distribution. These transformations can be done either:

- *Statically*, by transforming the Java classes prior to execution. Examples are [4], [7], [13], and [14].
- *Dynamically*, by transforming the Java classes upon loading using a replacement class loader technique [11].

However, such an implementation is not completely hidden from the program because of Java's introspection facilities.

2. *Build the JVM on top of a cluster enabled infrastructure.* For example, a distributed shared memory ([9], [10] and [16]). While this presents an SSI of the cluster, it is incapable of taking advantage of the semantics of Java to improve efficiency and performance.
3. *Build a cluster aware JVM.* This is the approach we have taken. The JVM presents an SSI to the application, but is itself aware of the cluster. This opens up possibilities for optimization based on the semantics of Java. As far as we know, there is only one other group [2] that has taken a similar approach.

[6] provide extensions to the Java language to support distributed applications. However, the programmer has to make use of these extensions to distribute the objects and hence this does not provide a true SSI.

Our cluster aware implementation of a Java Virtual Machine is **dJVM**, which stands for *distributed Java Virtual Machine*. It is based on the Jikes RVM [1] and provides an SSI to Java applications. The target machine for the dJVM is a 96 node, 192 processor machine, Bunyip [5] running Linux. It has Fast Ethernet communication hardware using M-VIA [12] and a Linux implementation of the VI Architecture [15] to provide low software overhead on inter-node communication. This will provide a good platform for evaluating the scalability of dJVM and distributed runtime support algorithms.

The Jikes RVM is written entirely in Java and provides an extensible framework for distributed virtual machines. There are two compilers in the Jikes RVM: the Baseline compiler and the Optimizing compiler. The Baseline compiler does not perform any analysis and translates Java bytecodes to a native equivalent. The optimizing compiler performs many aggressive optimizations. It can run on itself, producing competitive performance with production JVMs. This facility is leveraged to improve the performance of any extensions. In addition, it also provides several facilities including those for escape analysis, data dependence analysis and synchronization graphs, all of which are used, with extensions where required, to assist in the analysis of programs for load distribution. The initial design of dJVM targets the Baseline compiler; further development will be on the Optimizing compiler.

As far as we are aware, ours is the first distributed implementation of a JVM written entirely in Java.

The first goal is that of achieving an SSI; this is almost complete. This will be followed by optimizations on the system to improve performance. In order to enable an SSI, some of the important issues that needed to be addressed were the following:

Infrastructure In order to construct a distributed VM, several infra-structural components were altered. These include inter-node communication, the booting process and the use of system libraries.

VM Modifications The VM handles the manipulation of remote data in addition to local data. Class loading, method invocation and object access mechanism were enhanced to achieve this.

Object Allocation and Placement Mechanisms for remote object allocation were put in place, since this is crucial to achieve distribution.

In this report, we will look at all of the above issues. We only provide broad outlines of our solutions. Implementation details are not provided in this report. Section 2 deals with infrastructure, Section 3 with modifications to the Jikes RVM, Section 4 briefly discusses object allocation and placement and Section 5 outlines current status and future directions.

2 Infrastructure

The dJVM employs a master-slave architecture, where one node is the master and the rest are slaves. The boot process starts at the master node. This node is also responsible for the setting up of the communication channels between the slaves, holding global data and the loading of application classes. We now look in some detail at the boot and communication processes.

2.1 Booting

The dJVM boot process starts at the master node, which is made aware of the number of slave nodes that are present in the cluster; the boot process is initiated in the slaves as well. The boot process follows a similar sequence to that of the Jikes RVM boot process and has two main phases. The first completes the initialization of some runtime support data structures including thread, memory manager, compiler and class loader support structures, and initializes any classes that could not be incorporated into the boot image. The second sets up the scheduling to support multithreading, allowing the daemon and main threads to execute.

Once multithreading is enabled, the master activates the communication system (see Section 2.2), which initiates the message systems and threads for handling requests. It then waits for all the slaves to establish communication with it and with each other. The local and remote class identities must be resolved at this point. Once this is done, all globally usable statics initialized at boot time must be coalesced. Global values are held by the master node in this version of the dJVM. Finally, remote class loading is enabled. From this point on, all class loading (see Section 3.1) at a slave node will interact with the master node, since all class loading is first done at the master. All class loading at the master will be done locally.

2.2 Communication

The communication mechanisms employed in dJVM provide a simple abstraction over the underlying interfaces. This abstraction is designed to satisfy an initial set of requirements: independent memories (and the management of those memories) and, initially, flexibility.

Highly targeted hand crafted solutions can provide the best performance at the cost of flexibility. A more flexible system introduces impediments such as copying and allocation. Consequently, the initial communication system design is a compromise between our initial need for flexibility and our long term goal of performance.

To minimize the impact of internal communication design on the rest of the system, a communication manager interface is provided. It hides the underlying communication hardware being used, and the management of the resources associated with synchronous and asynchronous messages. It is responsible for initializing the system, bringing up a substrate (Section 2.2), a message registry (Section 2.2) and a pool of message processing threads (Section 2.2).

Substrate Using an abstract object to provide an interface to the underlying communications allows different communications mechanisms to be plugged in. Two different implementations (or substrates) have been developed:

- TCP/IP—providing a simple and reliable implementation for our initial system, and
- MVIA—a lower overhead solution for local networks.

The role of both substrates is to provide startup and shutdown of connections, and to handle outgoing and incoming messages.

Startup in a reliable and static set of nodes is simple. A node is designated as the coordinator and all initial connections are made with it. In turn, the coordinator informs all the nodes about all the other nodes. Each establishes the connections required with its peers. In our setup, the master node acts as the coordinator. Once the connections are established, each node may send messages to and receive messages from any other node. Later development will include the case where nodes join and leave the set while it is running.

Transmitting a message requires that message to be encoded into a buffer before being sent. A buffer is obtained from the substrate which maintains a pool of buffers, eliminating allocations requiring *garbage collection* (GC). This pool can be expanded if an inadequate number of buffers are currently available. Transmitted buffers are sent back to the pool to be reused. This buffer mechanism adds one level of copying to the transmission process. However, in the case of MVIA, it allows these buffers to be permanently locked in memory and used directly by the hardware.

Incoming messages are handled by message receiver threads. Each receiver thread blocks on a read from a connection. It processes the incoming packets and links them together to form a message; no additional copying is involved.

The assembled message is then decoded (see Section 2.2) and executed, after which it waits for the next message.

Messaging Model Flexibility plays a significant role in the initial system design. To achieve this, a class tree of message types is developed, which allows for quick and easy extension. Such a system introduces some overhead in the form of additional method calls and translation costs.

Each message type is described by a message class, which extends and implements a common abstract class. Each message class encapsulates the code to *send*, *decode* and *process* a message. Thus, the message functionality is message type dependent and may in part be determined at send time as either:

- *A synchronous message*, requiring a response before processing continues (for obtaining data, locks and invoking methods), or
- *An asynchronous message*, which does not need to block the sender. An asynchronous message:
  - does not require a response (commonly used for GC messages), or
  - requires a response (may be used for some system load monitoring or other non time-critical information).

Send The send method first requests a buffer from the substrate and encodes itself into that buffer. A message requiring a response registers itself with the communication manager before sending it through the substrate to the target node, and waits for the response message to notify it.

Decode Upon receipt, the message type is determined and the appropriate decode method is invoked. The decode method recovers the message from the buffer and does any initial processing where appropriate. A message that will only take a short time to execute may process itself and immediately generate a response (if required), e.g. getting a field of a remotely held object. A method that would potentially take a long time to execute (such as a method invocation) may grab an available message process thread for later processing.

Process This contains the code to perform the actual processing of the message. It may be invoked either by the decode method (for short duration operations) or by a handler thread.

Message Processing Threads Pool A pool of threads waiting to handle incoming requests is managed by the communication manager. A message that only takes a short time to process may be handled immediately. Other messages will be handed over to a handler thread, thus freeing its message receiver thread to process other incoming messages.

A handler thread is removed from the pool, passed the message to be processed and scheduled for processing, returning control to the calling thread. The handler thread will process the message and, on completion, place itself back in the pool.

3 VM Modifications

3.1 Class Loading and Resolution

The Jikes RVM maintains descriptions of types in the form of `VM_Class`, `VM_Field` and `VM_Method` objects. Loading a Java class generates a set of these objects to describe the type information of that class.

In a distributed system, it is necessary to have a commonly agreed to identification of these classes, and in the dJVM we use a centralized class loader to achieve this. Centralized class loading has some advantages and disadvantages. It provides a simple single point of coordination, but does create a bottleneck. However, class loading becomes less common as the program executes, and consequently this is not seen as a performance priority in long running applications.

The class loading strategy employed must account for the dJVM boot process and normal running. Therefore, it has two phases:

- Boot—an initial boot phase, prior to becoming a member of the cluster, in which classes must be loaded locally. Classes loaded locally must have their identity resolved with classes that are also present on the remote machines prior to centralized class loading being activated.
- Running—each class is loaded through a master node, ensuring a commonly agreed to identity for all newly loaded classes.

A set of objects are used to describe the type of a class, i.e. fields, methods and interfaces. A class type remains constant during the lifetime of the JVM. As such, each object describing this high level type information can be copied. Its identity is maintained by mapping each local copy to the same global identifier (UID).

In addition to replicating type information, for performance purposes, it is necessary to replicate literal values and static finals. This requires the class loading process to obtain these values and place them in the local VM's table of contents (JTOC). Furthermore, the local dictionaries used to maintain indexes to this data must also be updated.

Once loaded, a class can be instantiated. Instantiation compiles all static and virtual methods needed. Compilation can be done locally, generating code objects that are only visible within a node. The code generated is placed in arrays of type `INSTRUCTION` which can be directly executed. Each object has a TIB [1] (Type Information Block) as part of its header that describes some low level type specific information which includes a method table. In a homogeneous system, it is possible to replicate the TIB objects and the method code objects (this will be explored later).

The final phase is class initialization. This executes the static initialization code `<clinit>` for a class. In a JVM, class initialization happens only once. However, in the dJVM some of the runtime support structures are local to each node and must be initialized on each node where it is used. Thus, class initialization:

[1] A TIB is an object used to describe object type information.

– for a runtime support class (specified by implementing `DVM_LocalOnlyStatic`), occurs once on each node that uses it, or
– for a globally used class, occurs once on the master node.

Recall that the runtime support classes are for the internal management purposes of the dJVM and not for use by the application.

3.2 Method Invocation

When a method is invoked on an object, there are two main issues that need to be addressed before the method can be executed. The first is to determine where the object containing the method resides. The second is to decide where the method should be executed. We deal with the first issue in Section 3.4. We now look at the issue of method execution.

Once we locate the home node of the object, we need to decide where to execute the method. We can migrate the object to the node that invoked the method and execute it there. However, effective object placement locates objects at nodes based on execution patterns. Hence, we will not pursue this approach and will use the following options based on context.

– Where the method is a static method, or is a method that does not access fields of its object, or accesses only *immutable* (that is, *read only*) fields of its object, the method is executed locally, since the method code is replicated and available locally. The immutable fields of the object are cached locally to reduce remote accesses.
– In all other cases, the method is executed on the node where the object is located, through the remote method invocation mechanism described in Section 2.2.

If an object is known to be immutable, then that object is replicated and cached locally. The LID to UID mapping of that object is changed to indicate that it is locally cached.

When we execute a method on a remote node, the execution context of the corresponding thread changes, along with the physical identity of the thread. However, the global identity of the thread must not change. In order to ensure this, each logical thread has a unique global identifier and the mapping of that identifier to a local physical thread (`VM_Thread`) is changed at each node where it executes. This requires special handling and caching of application level threads, i.e. threads that extend `java.lang.Thread`.

This raises the question of the reuse of local thread structures. Figure 1 depicts a thread, *Thread A*, instigated on *Node 1* that remotely calls a method `b.x()` on *Node 2*. The local thread structure, *Thread B*, is given the same global identity as *Thread A*. Method `b.x()` performs a remote call to method `c.y()` back on *Node 1*. Local *Thread A* is reused, continuing the processing of threads call chain. This is more efficient than allocating a new thread at *Node 1* that logically should have the same global identity as *Thread A*. Our design differs in

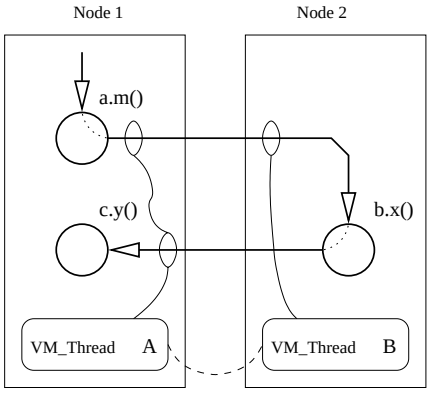

Fig. 1. Local VM_Thread Reuse.

this respect from that of [2], where they create a new thread at *Node 1* in such a scenario.

Method calls are synchronous. Thus, a call made to another will block the instigating thread until that call is satisfied. However, if the same global thread calls a method on that node, then the blocking thread is interrupted and informed of the new incoming operation it is required to perform. Once that operation has been completed, it returns to waiting for the remote method call to return.

To effect a remote invocation, a message encapsulating the identity of the thread and the method to be executed, along with the appropriate parameters, is generated. This message is passed to the target node, where a local thread resource is assigned (if one already hasn't), the parameters placed on the stack and the appropriate method invoked. This mechanism can be implemented either by:

- *Generic*—Code is constructed at compile time to retrieve the parameters from the stack (converting each reference from a local to a global representation) packing them into a message, which is sent as a request to the target node. Upon receipt of an invocation request, the parameters must be unpacked onto the stack and the method invoked.
- *Proxy methods*—Each method that is compiled has two additional methods generated, *proxy* and *stub*. The first packs and sends a request and the second unpacks an invokes the method. This requires determining whether an object is local or remote. In [3], the authors state that "We cannot make this determination by using different classes for the master and proxy, or by adding a field to the application, as the introspection APIs would make this visible to the application, violating SSI". However, this can be circumvented by modifying the introspection mechanism in the Jikes RVM runtime libraries, so that the mutations in the class definitions are hidden from the application.

We use the second solution, as it is cleaner and more flexible.

Exceptions and interrupts must also be accounted for. A thread that handles an incoming request must catch all possible exceptions from the application code. Once an exception is caught, it must package that exception and return it to the node that initiated the call, where it will be re-raised. By contrast, an interrupt must be propagated along the call chain to the node currently executing the global thread, where it is finally dispatched to the underlying `VM_Thread`.

3.3 Data Access

In broad terms, there are two areas that need to be considered: globally referenced data (i.e. static variables) and instance data (i.e. objects and arrays).

Globally Referenced Data Each node must manage its local resources. The local resource management information in a non-distributed JVM is global data. However, in a distributed JVM, most of that information is global only within the context of that node. Hence, the set of static variables in a distributed JVM can be divided into two mutually disjoint sets, namely:

1. the set of static variables that are global to all the nodes, and
2. the set of static variables that are global within a specific node.

One way to implement the above is to encapsulate the node specific information in an object that is visible only within the specific node. Another is to change the code that accesses static data according to the data that is being accessed. We take the second approach.

A runtime test is necessary to determine if a static variable is held locally or remotely. Furthermore, as the Jikes RVM is written in Java, including all its runtime structures, this raises two issues that must be dealt with:

1. Whether the static variable is only a local runtime support variable or not.
2. If the static variable may be locally or remotely held, then we must test whether it is locally or remotely held using a local runtime structure.

If care is not taken, then the code generated to test whether a static variable is local or remote held may itself require a similar test, resulting in an infinite loop.

In the Jikes RVM the static field are held in a Java Table of Contents (JTOC). Associated with each field is a descriptor, that descriptor identifies the category and type of information held, e.g. literal `int`, static field `long`. The set of descriptors is an array of bytes, held as a static array in `VM_Statics`, and is therefore referenced from the JTOC. Each descriptor has two unused bits, and we use these to indicate whether it is a read only field and/or a remotely held field.

In general, the runtime support classes in the Jikes RVM only contain static variables that are used locally within a node. This is communicated to the compiler through a simple annotation method commonly used in Java – an empty

interface `DVM_LocalOnlyStatic` is implemented by any class that contains static data that is always accessed locally. The code generated to access a static field of such a class is unchanged from that of the original compiler. For other classes, the test described above is performed. Clearly, this does not introduce any overhead for locally used static variables. However, it does introduce some overhead for other static variables. The overhead will be reduced in later implementations by combining the descriptor and JTOC information into the one array.

Instance Data The approach taken to implementing the reference faulting mechanism, outlined in Section 3.4, dictates the changes necessary to the compilers for handling instance data. In particular, there are four different types of code to consider: object and array access, code execution, lock operations and type checks.

Field access (`getfield` and `putfield`) and array element access (`aaload`, `aastore` etc.), dereference an object[2] to determine the memory location of the data. The software detection of remote references mentioned in Section 3.4 necessitates a test of the reference itself. A local reference is accessed in the normal manner, whereas a remote access is initiated by calling a static method, which generates a message that contains a description of the remote operation (see Section 2.2).

Type Operations The remaining operations are type checking operations. Explicit type checking operations (`checkcast` and `instanceof`) can interrogate the types through remote calls. We cache this information locally, since an object's type remains unchanged during its life time. Any interrogation of an object to obtain its TIB is intercepted to obtain the TIB from the local cache.

3.4 Object Location

We use a reference faulting mechanism to determine whether an object has a locally held instance, or is only available remotely. This is achieved by using an appropriate global and local addressing scheme for objects. Each object has an associated *universal identifier* or *UID* that uniquely identifies the object in the cluster. The UID needs to be resolved into an object address at a specific node. One of the ways in which the UID can be allocated is *centrally*, where the allocation of UIDs is done by a master node in the cluster. However, this could lead to a bottleneck at that node. We have chosen to use a *decentralized* approach, where each node in the cluster allocates a UID for an object that it owns, from a range of UIDs under its control. A UID is generated when an object reference is exported for the first time. The node that owns an object is called the *home node* of that object. While this eliminates the above mentioned bottleneck problem, it does lead to more complicated updates resulting from object movement from one node to another.

[2] An array element access is considered to be a variant of field access.

At any given node in the cluster, an object reference either points to a local object or to a remote object. In our implementation, a local object has an associated *object identifier* or *OID*. This is identical to the address of the object at that node and thus avoids any overheads incurred through indirection tables or indexes. A remote object has an associated *local logical identifier* or *LID* at that node. This LID needs to be mapped to the UID of the object to determine its exact location in the cluster.

In the Jikes RVM, all object and array addresses are 4 byte aligned. We use this property to make the reference faulting mechanism work. All the LIDs are misaligned, while the OIDs, being actual object addresses, are not. This can be implemented by either:

- *Software*—Misaligned addresses can be detected by examining the LSBs (least significant bits) of an address and branching if not zero. This introduces a couple of instructions into the instruction pipeline. Importantly, no indirection or additional loads are required.
- *Hardware*—In the Jikes RVM, checking array bounds and object types are 4 byte aligned operations, and their interrogation via a misaligned address will cause a hardware trap. Most accesses will be to local objects, so the added expense of a hardware trap for remote objects will be outweighed by zero overhead for local access.

We have currently implemented this using software, but intend to implement the hardware faulting mechanism.

3.5 Locking

Locking operations are directed to the home node of the object, and in the case of locks on classes they are directed to the home node of the class (initially, the master node). The thread identifier sent with the lock is the global identifier of the thread, for obvious reasons. A thread that already has a lock can acquire additional locks on the same object. The number of locks and unlocks must be equal. For efficiency, additional requests need not be sent to the home node. A local count can be kept, and an unlock request send to the home node once the count reaches zero.

The explicit lock operations `monitorenter` and `monitorexit` are handled by the Jikes RVM runtime system and do not need compiler modifications. Implicit locks on object instances (`synchronized` methods) are similarly handled. However, implicit locks on statics must be directed to the home node of the class. In the case of classes that implement `DVM_LocalOnlyStatic`, this is done locally. In all other cases, it is done by the master node.

4 Object Allocation and Placement

In order to enable the distribution of objects across the nodes in the cluster, there should be a way of remotely allocating an object on a specified node. In

the Jikes RVM, the `VM_Allocator` class does the work of object allocation. In dJVM, this is replaced with an allocator that directs requests to a standard local allocator or to an allocator on another node. On initialization, each node will have a instance of an allocator that directs allocation requests to the local node. Each allocator instance acts as a placeholder, enabling the remote invocation mechanism to be used to effect remote allocation requests to specific nodes. The UID of this placeholder object is known to all the other nodes. A remote allocation request is passed on to the placeholder object of the node at which the allocation is to be made, which then allocates the object locally at that node.

Introducing a local or remote allocation decision process at runtime can be expensive. Compile time analysis can eliminate some of these decisions by generating local allocation code where it is clearly sensible to do so. Object placement is important for load balancing and performance improvements. Ideally, a new object should be placed on the node where it is most required. In [2, 3] a number of techniques and patterns were examined to improve efficiency and these shall be incorporated into the dJVM. Additionally, we will examine further techniques using escape analysis, call chain, and static and dynamic profiling information to enhance object placement.

5 Conclusions and Future Work

In this paper, we described the dJVM and presented some of the design issues that we came across in developing our system. We also outlined some solutions to these issues. Our initial aim of getting a prototype of a dJVM that supports SSI up and running is nearing completion. Once this is done, we will look at the following issues:

- *Optimizing Compiler*—The optimizing compiler implements a range of analysis and optimization techniques. These optimization techniques can be applied to the Jikes RVM (and hence our extensions) as well as the application code. Consequently, we intend to use facilities, such as escape analysis and profiling, to feed into the object placement and migration decision making processes.
- *Communication*—Flattening the communication hierarchy and removing all but essential object creation and data copying.

Concurrently with the development of dJVM using the optimizing compiler, we will investigate some techniques to improve performance. We intend to use techniques such as code analysis, and static and dynamic profiling, for object placement and migration, object caching and thread migration. We also intend to implement efficient distributed garbage collection algorithms.

6 Acknowledgment

The dJVM project is funded under the ANU-Fujitsu CAP Research Program.

References

1. Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño—a Compiler-supported Java Virtual Machine for Servers. In *Workshop on Compiler Support for Software System (WCSSS99)*, Atlanta, May 1999.
2. Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Cluster Aware JVM. In *First Annual Workshop on Java for High-Performance Computing (with ICS99)*, Rhodes Greece, June 1999.
3. Yariv Aridor, Michael Factor, and Avi Teperman. Implementing Java on Clusters. In *Euro-Par 2001, LCNS 2150*, pages 722–732, Rhodes Greece, 2001. Springer-Verlag.
4. D. Caromel and J. Vayssiere. A Java framework for seamless sequential, multi-threaded, and distributed programming. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, INRIA Sophia Antipolis, Greece, 1998.
5. DCS. Bunyip (Beowulf) Project. http://tux.anu.edu.au/Projects/Bunyip/.
6. Michael Hicks, Suresh Jagannathan, Richard Kelsey, Jonathon T. Moore, and Cristian Ungureanu. Transparent communication for distributed objects in java. In *ACM 1999 Conference on Java Grande*, pages 160–170, San Francisco, California, USA, 1999. ACM Press.
7. P. Launay and J. Pazat. A framework for parallel programming in Java. EUT Report 1154, IRISA, December 1997.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. 2nd edition, 1999.
9. M.J.M. Ma, F.C.M. Lau C.L. Wang, and Z. Xu. JESSICA: Java-Enabled Single System Image Computing Architecture. In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99)*, Las Vegas, July 1999.
10. M. MacBeth, K. McGuigan, and P. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *IBM Centre for Advanced Studies Conference*, Canada, NovemberDecember 1998.
11. Alonso Marquez, John N Zigman, and Stephen M Blackburn. Fast portable orthogonally persistent java. *Software: Practice and Experience*, 30(4):449–479, April 2000.
12. NERSC. M-VIA. http://www.nersc.gov/research/FTG/via/.
13. Objectspace. Voyager. http://www.objectspace.com/products/voyager/.
14. Michael Philippsen and Matthias Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
15. VIArch. VIArch. http://www.viarch.org/.
16. Weimin Yu and Alan L. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.