cJVM: A Cluster JVM Architecture for Single System Image

Yariv Aridor, Michael Factor and Avi Teperman*
[1] *IBM Haifa Research Lab*
Matam, Advanced technology Center, Haifa 31905, ISRAEL
PH: +972-4-8296350, FAX: +972-4-8296114,
(E-mail: yariv|factor|teperman@il.ibm.com)

SUMMARY

cJVM is a Java Virtual Machine (JVM) which provides a single system image of a traditional JVM while executing in a distributed fashion on the nodes of a cluster. cJVM virtualizes the cluster, supporting any pure Java application without requiring that applications be tailored specifically for it. The aim of cJVM is to obtain improved scalability for a class of Java Server Applications by distributing the application's work among the cluster's computing resources. cJVM is based on a novel object model which distinguishes between an application's view of an object (e.g., every object is a unique data structure) and its implementation (e.g., objects may have consistent replications on different nodes). This enables us to exploit knowledge on the usage of individual objects to improve performance (e.g., using object replications to increase locality of access to objects).

Currently, we have already completed a prototype which runs pure Java applications on a cluster of NT workstations connected via a Myrinet fast switch. The prototype provides a single system image to applications, distributing the application's threads and objects over the cluster. We have used cJVM to run without change a *real* Java Server Application containing over 10Kloc and achieve high scalability for it on a cluster. We also achieved linear speedup for another application with a large number of independent threads. This paper discusses cJVM's architecture and implementation. It focuses on achieving a single system image for a traditional JVM on a cluster while describing in short how we aim at obtaining scalability.

KEY WORDS: *cluster machine, cJVM, Java, objects, Java Virtual Machine, single system image*

INTRODUCTION

What if we wanted to take advantage of a cluster to improve the performance of an existing multi-threaded Java application? How would we distribute the work of the application among the nodes of the cluster? How would we enable the application to be unaware of the fact that it is executing on a cluster? How could we execute an existing application which was originally developed for a conventional Java Virtual Machine? We are trying to address these questions.

*Correspondence to: E-mail: teperman@il.ibm.com

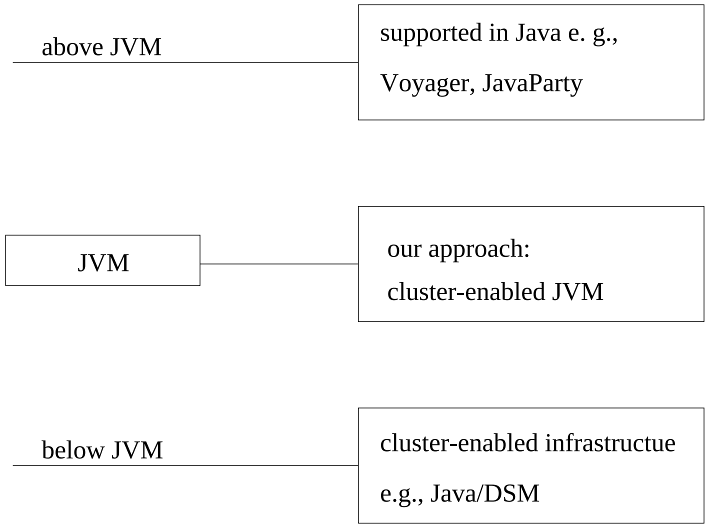

Figure 1. Three Approaches to Clustered JVM

The answers to these questions have two parts. By focusing on existing Java applications we are constrained to solutions that look to the application like a conventional implementation of a Java Virtual Machine (JVM)[4]. Thus, the first part of the answer is that we must provide to a Java application a single system image (SSI) view of a cluster. In other words, the Java application will have the illusion that the cluster is a single computing resource [7], even though the application will execute on the multiple, independent, nodes composing the cluster.

The second part of the answer is that we must intelligently manage the threads and objects created by the application to achieve a performance benefit for a large class of real applications. This paper focuses on the first part of the answer, namely what is required to build a SSI view of a cluster from the perspective of a Java application.

There are three different approaches to enabling a Java application to see a cluster as a single computing resource, as seen in figure 1. First, we could provide an implementation above the JVM in Java, e.g., using third-party Java packages. Several others have taken this approach (e.g., [8, 10, 3, 17]), and in all cases the distributed nature of the implementation is not completely hidden from the program. In other words, the view of a single system image is incomplete. Second, we could build upon a cluster enabled infrastructure below the JVM, e.g., a distributed shared memory, as was done in [13, 20]; such an approach is capable of presenting a single system image, however it is inherently incapable of taking advantage of knowledge of the semantics of Java as described in the next paragraph. Finally, we can provide an implementation of a Java virtual machine which is itself aware of the underlying cluster, but which completely hides that fact from the application. This is our approach. To the best of our knowledge, we are the first to implement and report on this approach.

cJVM provides a single system image of the cluster to a Java application. By working at the level of the virtual machine, we enable exploiting opportunities for optimizations based upon the semantics of Java.[1] Examples of such opportunities include, using

[1] To be precise, we enable taking advantage of semantics at the level of the Java Virtual Machine.

distinct caching and replication policies both at the level of individual objects and individual fields, enabling a Java thread to migrate between nodes to improve locality of access to objects, analyzing the code to prove that certain accesses are always local, etc.

cJVM executes on a cluster, distributing an application's threads and objects across the nodes of the cluster and providing correct semantics for any application written only in pure Java.

The major contributions we report upon in this work are:

1. the architecture of a single system image of a Java Virtual Machine on a cluster
2. a distributed memory model which supports the cluster JVM
3. a novel object model which distinguishes between the application's view of an object's class and the actual implementation and which enables taking advantage of knowledge of the usage of specific objects to improve performance
4. an implementation of threads which transparently supports distributed stacks.

In the next chapter we present relevant background material. This background includes a brief description of a standard Java Virtual Machine (JVM) highlighting the issues and difficulties of implementing a JVM on a cluster, a definition of a cluster and a description of the type of applications for which we wish to achieve performance improvement. The following Chapter (cJVM APPROACH) highlights the essential ingredients of cJVM. The cJVM ARCHITECTURE Chapter describes in detail cJVM, showing the implementation techniques we used to support running Java application in a cluster. The following two Chapters summarize the status of cJVM and describe related work. In the final chapter, we summarize our major contributions and describe our directions of future work.

BACKGROUND

Java Virtual machine

A Java Virtual Machine (JVM) is a platform-specific (operating system and hardware) program that implements a well-defined, platform-independent virtual machine [4]. There are currently implementations of JVMs for a range of platforms from embedded systems up to mainframes.

The JVM is a stack machine whose semantics are given by a set of bytecodes. All code belongs to a method which in turn belongs to a class; Java and the JVM are very flexible allowing classes to be dynamically created by an application, loaded, and then executed in the same application. Java classes are organized into hierarchy supporting single implementation inheritance and multiple interface inheritance. When executed, these bytecodes change the state of the stack and can mutate objects allocated in a heap. The JVM is designed to support multiple concurrent threads of execution; when most traditional implementations are run on a uniprocessor, concurrency comes from time-slicing, while on a multi-processor, true parallelism is possible.

It is important to note that the JVM and Java are not identical; a JVM can support languages other than Java if they are translated to bytecodes. In addition to the JVM proper, Java comes with a rich set of run-time core classes which must be supported by a compliant Java environment. Some of these core classes, as well as application code, may use native methods, methods implemented in a language other than Java. These methods are used, in particular, to interface with the operating system.

The goal of cJVM is to produce a compliant JVM which executes on a cluster, taking advantage of the cluster for scalability, and which provides the illusion that the cluster is a single system.

Memory Model

The basic memory model for the data manipulated by an application running on a Java Virtual Machine consists of stacks and a heap. Each stack consists of a collection of stack frames, one for each invoked method which did not return, where the frame is divided into three areas: parameters, variables, and a conventional push-down stack. The data in each of these three areas consists of primitive types (integral, floating point, and boolean types) or references to objects.

Objects are allocated in a garbage collected heap via explicit program requests to create a new object. The request places a reference to the object on the top of the stack, enabling the object to be further manipulated.

In addition to the heap and the stack, the JVM internally uses system memory for various resources including meta-data related to the program's classes, the program's instructions, and the constant pool[2]. The meta-data associated with a class includes information such as an object representing the class, the class's name, the class's superclasses, information on the class methods (kept in method block structure), etc. Some of this meta-data is represented at run-time as normal Java objects. The program's instructions are the bytecodes (see below) composing its methods.

Bytecodes

We divide the JVM bytecodes into different groups based upon the type of memory that they access. Based upon this division, we gain a better understanding of what is required to ensure the correct semantics of the bytecode in a cluster.

A large set of bytecodes only access the Java stack frame of the current method (e.g., load (store) to (from) a stack frame, control flow, or arithmetic operations on values stored on the stack). It is relatively easy to ensure a single system image for these bytecodes since the code can be replicated and since a stack frame is only accessed by a single thread.

A smaller subset of the bytecodes accesses the constant pool (e.g., load (store) to (from) the stack). Many of these accesses to the constant pool are only on the first invocation, i.e., to resolve the operand of the bytecode. Once resolved, the bytecode is rewritten to point to the entry which now contains a binary encoding of the previously symbolic information. This group is also easy to handle since bytecode resolution is idempotent.

A final group accesses objects in the heap (e.g., `getfield` and `putfield` to access a specific object's fields). It is this group that is interesting for a cluster JVM. If two different nodes access the same object we need to ensure they see the same values, within the constrains of Java's memory consistency [21].

Interpreter Loop

The JVM as a virtual stack machine is powered by an interpreter loop. This is a loop in which on each iteration the next bytecode is executed, the stack is modified

[2]The constant pool is the Java equivalent to symbol table in languages like C. It maintains linkage information about, for example, constants and references to other fields and methods which may not yet be loaded.

as specified by the bytecode, the heap is accessed as appropriate, and the program counter is updated. The interpreter loop can be viewed as a giant switch statement specifying a distinct action for each of the bytecodes.

Usually Java Development Kit (JDK) comes with a Just In Time (JIT) compiler which compiles frequently invoked Java methods to platform native code. This boosts the performance of Java applications significantly. Currently we are focusing on the SSI aspect of cJVM postponing performance issues like the JIT to the next phase of our research.

Threads

Java is a multithreaded language; the language provides the programmer with convenient facilities to define multiple independent threads of execution. In Java a programmer creates a new thread of execution by creating an instance of a `java.lang.Thread` or its subclass; this object is created in the same way as any other object. The behavior of the thread is defined either by implementing a method `run()` in a subclass of `java.lang.Thread` or by passing to the constructor an object that implements the `java.lang.Runnable` interface.

The thread starts executing after the `start()` method is invoked on the thread's Java object. Depending upon the implementation of the JVM, this associates the Java thread with either a system thread or with a thread from a thread library (e.g., "green threads"). On a uni-processor, parallelism between threads is obtained via time-slicing. On a multi-processor true parallelism is possible.

A key point of Java threads is that the program relates to the threads in the same way it relates to any other object. Threads are thus a natural and easy facility to utilize. Thus, Java makes it much easier to write parallel programs than in more traditional languages such as C or C++ where the threading model is an extra-lingual facility.

Clusters

Our focus is on dedicated compute clusters. We consider a cluster to be a collection of homogeneous (same operating system and architecture) machines connected by a fast (i.e., low latency - microseconds and not milliseconds) communications medium. Each node in the cluster is independent, having its own copy of the operating system. We assume that other than the interconnect, there are no physically shared resources between the nodes of the cluster, i.e., there is no physically shared memory. For purposes of our prototype we assumed that there is a logically shared file system, but this is not essential. Examples of such a cluster are a set of PCs connected by a switch or IBM's RS/6000 SP computer. While clusters are used for both scalability and high-availability, in this work we only look at the issue of scalability.

Java Server Applications

We are aiming to provide a solution to scale a particular class of Java applications. We call these applications Java Server Applications (JSAs). These are second tier applications with the basic structure of a concurrent daemon that:

- Accepts a sequence of requests from clients.
- Typically accesses an external "database" in processing the request.
- Has some interactions (i.e., sharing) among requests.

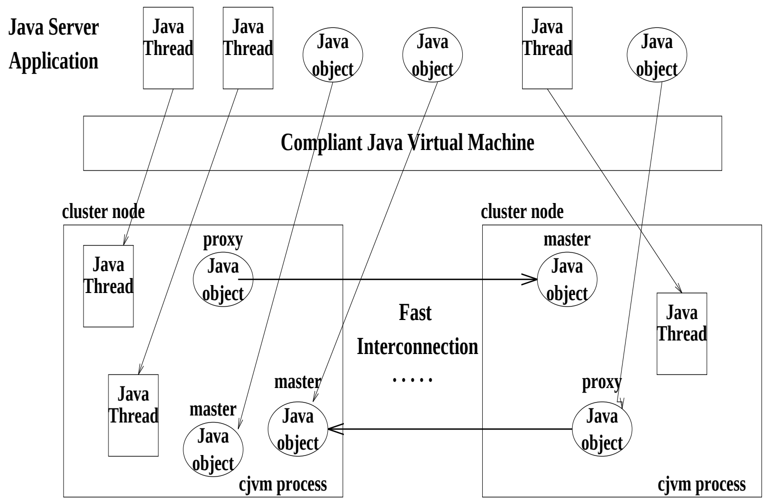

Figure 2. Cluster JVM

In very general terms scalability for a Java Server Application means increasing the number of client requests it can satisfy per unit time. Note that because we are interested in applications which are concurrent daemons, we assume that the application has been explicitly written to use Java threads. We are explicitly not trying to parallelize existing serial code.

cJVM APPROACH

As stated in the introduction, our approach to virtualizing the cluster via a Java Virtual Machine is to cluster-enable an implementation of the JVM. Even though an approach built upon a distributed shared memory infrastructure also provides a single system image view of the cluster we preferred the cluster-enabled JVM approach. The reason being that we can optimize the program within the object model by using information we can obtain from knowledge of Java. One way to obtain this information is through code analysis and rewriting. In our prototype work, we started with Sun's reference implementation for the 1.2 JDK on NT.

Figure 2 shows our basic approach. The upper half shows the threads (boxes) and objects (circles) of a Java application as seen by the programmer. They see a traditional Java Virtual Machine. The lower half shows the objects and threads of the Java application distributed transparently (to the Java application) across the nodes of the cluster by the implementation of cJVM. The arrows from the upper half to the lower show how the objects and threads have been distributed. Horizontal arrows show interactions between master and proxy objects.

There is a cJVM process on each cluster node, where the collection of processes as a whole constitutes cJVM. In other words, each of the processes implements the Java interpreter loop while executing part of the application's Java threads and containing portion of the Java objects defined by the application.

In the sequel we describe cJVM approach to the implementation of the following major components: *Object Model, Methods Execution, Thread Model and Memory*

Model. Even though this paper focuses on the architecture of cJVM, in the last section we describe cJVM's approach towards obtaining scalability due to its influence on the architecture.

Object Model

Our cluster JVM implements a distributed heap. In general, when a new object is created, it is created at the cluster node where the request to create the object was executed. Every object has exactly one master copy located at the node that created the object; other nodes that access the object do so via a proxy stored in their local heap. We discuss this memory model in more detail in the next chapter. Two of the key challenges of cJVM are 1) giving the application the illusion that it is using a single monolithic heap and 2) hiding the distinction between master and proxy from the application.

To enable the application to be unaware of whether or not it has a proxy or master copy of an object[3](this is required to preserve the application's illusion that it is executing on a single system) we define a new object model. This model allows multiple implementations for a single method to coexist in a single class and allows selecting on a *per-object-instance* basis the precise code to execute for a given method. We elaborate on this new object model in the next chapter.

Changing the object model such that method invocations are transferred to the master copy of the object is insufficient to support the single system illusion with respect to the heap. As described in the previous chapter, there is a set of field access and monitor access bytecodes which also access the heap. Since we are providing a distributed heap, these bytecodes must be modified to work correctly. In principle (but see below), each of the bytecodes that accesses the heap must be modified to determine if the data it is accessing is located at the node where the bytecode is executed or if it is located at another node. If the data is located at another node, a remote access is required.

Method Execution

When a node accesses a proxy our basic approach is *method shipping*, i.e., the proxy - transparently to the application - redirects the flow of execution to the node where the object's master copy is located. We enhance this basic approach of method shipping with class and object specific caching and replication policies.

cJVM supports pure Java applications; i.e., applications with native methods are not supported since any internal state modified inside them is not exposed to cJVM. However all native and JNI methods which are part of the JDK and which are likely to be used by Java server applications were modified by us to be cluster aware and are used by our prototype. An example of a JDK package with native methods which is not supported is `java.awt` since server side applications do not use GUI.

Thread Model

To enable a scalability gain, different application threads need to execute on different nodes of the cluster. To distribute the threads, we need to change the way threads

[3]Java enables an application at run-time to determine the class of an object instance.

are created; unlike our base approach for objects, when a new thread object is created it is created on the best node as determined by a pluggable load balancing function. Further, our use of method shipping requires changing the view of a thread as an entity that executes its set of instructions on a given node; rather a thread itself becomes distributed. We thus define a new thread model which we detail in the next chapter.

Means to Achieve Scalability

To obtain high scalability on a cluster two main issues have to be addressed: reducing the amount of communication interactions between the nodes of the cluster and reducing the cost of each individual interaction. For the latter, cJVM integrates with very fast networking media. To reduce the amount of communication, cJVM employs a *large* combination of (mostly) simple optimizations addressing caching, locality of execution and object migration. The full range of the optimizations applied by cJVM is described in detail in [2]. In this section we only highlight them.

Caching techniques focus on data which is not mutated during a given program execution. We look at data at the level of classes, objects and even individual object fields. For classes, we cache their static fields which are usually set once and read multiple times. For objects cJVM employs two different approached. 1) Caching all instances belonging to classes whose instances are read-only and 2) caching selected arrays reachable from static final variables, as these arrays tend to not be mutated. Finally, cJVM applies caching of individual fields which are speculatively identified as immutable ones (e.g., private fields which can be modified only by objects of the same class) after the object is initialized.

These caching optimizations provide the biggest performance benefit when used in concert with invocation optimizations. Invocation optimizations address the issue of where a particular method should be executed, going beyond the generic approach of method shipping i.e., always executing a method at the master node[4]. For example, since cJVM caches static fields with class proxies, we gain performance by executing static methods locally upon these proxies even though the master class object is on another node. The same holds for stateless methods which work only on the local thread's stack (e.g., `java/lang/Math.min(a,b)` method accepts two integer parameters and returns the smaller one) or methods which access only immutable fields. Such a classification of methods is done by intra-procedural analysis at runtime, when a class is loaded, which look at the bytecodes each method uses to find the way it accesses the heap.

Placement optimizations attempt to place newly created objects on the node where they will be used. In addition, they migrate objects to enhance their locality with respect to the thread using the object. Since migration can be very expensive (e.g., providing thread-safeness), the use of this optimization is limited.

Our optimizations are almost all speculative. They utilize knowledge of Java semantics (e.g., the heap accesses performed by a method) and data usage patterns (e.g., the typical usage of static data) extracted by analyzing the bytecodes during class loading to determine which optimization to apply on which datum. To handle cases where a heuristics decision was "wrong" the optimizations are augmented with invalidation protocols.

[4]This is made possible by modifying the semantics of the bytecodes to be cluster-aware, as described in the Object Model section.

The big benefit in performance is not from any single isolated optimization but rather from the synergy that comes from using a large set of optimizations at the same time.

Beyond these optimizations, we can also take advantage of the fact that we are modifying the JVM to be cluster-aware to utilize run-time profiling information. For instance, for a given proxy, we can measure the time required for its different implementations and choose the least expensive one. Run-time profiling is the only feature we describe which we have not yet implemented.

Finally, we apply bytecode rewriting using cJVM-specific pseudo-bytecodes;[5] This enables us to change the method's implementation to be directly cluster-enabled. In the next chapter in the section on Thread Model we present an example of this to support distribution of threads towards gaining scalability. In addition, while analyzing the bytecode of every method, we can determine that a particular heap access will always be local, e.g., a master copy of an object can always locally access fields stored in itself (i.e., accesses to `this`), eliminating the overhead of checks on certain bytecodes which can access the heap. Such analysis and bytecode rewriting demonstrate the capabilities, at the level of JVM to apply implicit solutions for efficient cluster-enabled functionality.

Miscellaneous Features

While our base approach for supporting objects in cJVM is a distributed heap with method shipping, for code we use replication. Each node in the cluster contains an independent copy of the code for the classes it is using. On the one hand, we would like to directly load the code on each node using a class as this is a direct way to correctly build the internal data structures supporting the class, e.g., constant pool, method blocks, etc. On the other hand, an application sees a class as an object, and it needs to see only a single object for a given class, even if the class is loaded independently on multiple nodes. Further, the class points to other objects, e.g., its name, its superclass, etc., which also must maintain the illusion of a single system. Finally, when a class is initialized, a chunk of application code, the `<clinit>` or class initialization method, is executed. It is a mistake to execute this code more than once. To address these issues, we define a master copy of a class, similar to the way we define a master for objects. We perform a partial load of the class on any node that uses the class, and for any aspects which must be cluster-enabled, the node contacts the class's master which loads the class in the same way a traditional JVM loads it.

To support remote accesses, as well as to support method shipping and some additional functions, each cJVM process contains a set of server threads. These threads, which we manage as a pool with high and low watermarks, execute an infinite loop in which they wait for requests, service the request, and send a response. While a remote DMA approach, as is supported by VIA [11], might be useful in some cases, e.g., accessing a primitive field of a remote object, it cannot handle other cases, e.g., locking a remote object. For simplicity of implementation at this stage of our effort, we chose to use a single implementation instead of the optimal implementation for each specific type of remote access.

[5]Pseudo-bytecodes are bytecodes that are specific to the cJVM implementation. They never appear in a class file and are never seen outside of cJVM, thus they do not make cJVM non-standard.

As stated in the BACKGROUND chapter some of Java core classes use native methods. We are explicitly not focused on native methods for the following reasons. Most of the native methods[6] are related to GUI which is irrelevant to our target application; i.e., Java Server Applications. Native methods in, e.g., Math, Zip and Jar packages do not need to change since they can be executed locally. This leaves us with relatively small number of JNI methods which need to become cluster aware. There is no silver bullet to solve this problem, i.e., to automatically make each JNI method cluster aware. We have done that manually for each JNI method in our code base.

There is a large number of additional changes we need to make to a traditional JVM to turn it into a cluster JVM. These changes include modifications to the initialization of the JVM such that only one node executes the applications main method, changes to JVM termination, changes to numerous native methods, etc. We do not elaborate on any of these items.

cJVM ARCHITECTURE

This section focuses on the three of the more novel aspects of cJVM's architecture: the object model, the thread model and the memory model.

Object Model

The object model of cJVM is composed of master objects and proxies. A master object is the object, as defined by the programmer. A proxy is a surrogate for a remote object through which that remote object can be accessed. While a proxy is a fundamental concept used in systems supporting location-transparent access to remote objects [9, 10], we push the idea one step further. *Smart proxies* are a novel mechanism which allows multiple proxy implementations for a given class while using the most efficient implementation on a per object basis.

To motivate smart proxies, consider three different vector objects, all of which are accessed by multiple threads of an application:

- vector A: this vector is relatively small, each access is a bursty one and at any point in time localized to a thread, and the accesses involve a mix of read and writes
- vector B: this vector is relatively large, the accesses are sparse and not localized to a single thread at any point in time, and the accesses involve a mixture of read and write operations
- vector C: this vector is relatively large, after a period of initialization all of the accesses are read-only, and the accesses are continuous and not localized to a single thread

It is clear that different proxy implementations for each of the three cases can improve performance. For vector A, it would be beneficial to use a caching proxy which allows only exclusive caching. Vector B, requires a simple proxy which ships all accesses to the master copy, while for vector C, it would be beneficial to use proxies which support multiple readers, single writer (e.g., allowing, at any point in time during the program execution, at most one caching proxy to update the vector and propagate the changes to all other proxies, while applying all read operations locally).

[6] 719 out of 1156 in the JDK 1.2 code base we use

There are two challenges applying smart proxies in the context of single system image:

- preserving the application's illusion that is it executing on a single system, being unaware whether it is using a proxy or a master object.
- designing efficient proxy implementations which do not violate the Java semantics.

The first challenge is met by (1) implementing proxy objects with the same internal representation (e.g. object header, method tables) as their master objects and (2) having all the proxy implementations coexist within a single class object.

Figure 3 describes our implementation of the new object model based on Javasoft's JVM implementation for Windows NT 4.0 where various arrows represent pointers to data structures. Specifically, the virtual method table of a class is logically extended into an array of virtual method tables, as seen in figure 3. In addition to the original table of method code, each of the other tables refers to the code for a particular proxy implementation. All the virtual tables and the code for the proxy implementations are created by cJVM on the fly during class loading. Thus, every class has an array of at least two virtual tables: one for the original method code and one for the code of the most efficient proxy implementation. The code generated for the methods of simple proxy is straight forward. cJVM introduced a new pseudo-bytecode, `execute_remote`, whose implementation does method shipping to the master object. Thus the code of each simple proxy method is as follows:

```
execute_remote
<x>return
```

where `<x>` represents the type of object returned by the original method. In addition to simple proxy, classes can, as well, maintain other proxy implementations such as one that collects run-time statistics, if the program is running in a profiling mode, or multiple implementations that are changed dynamically based on run-time information (i.e. object locality). An example is given later in this section.

Upon creation of a master object or a proxy, it points to the correct virtual table of its implementation which distinguishes it from other proxies as well as from its master object; this distinction is only visible from within the implementation of cJVM - the application cannot distinguish between the master and the proxies. It should be noted that it is possible to change proxy implementations during run-time. A particular set of implementations may constrain changing to other implementations to occur only if there are no active methods executing on the instance (e.g., during garbage collection). However, at the level of a mechanism, cJVM is architected without any such constraints.

To help address the second challenge of designing efficient proxy implementations which maintain the Java semantics, we analyze every class during class loading, to classify methods based upon the way they accesses object fields. We use this information to help choose the most efficient proxy implementation for every method. Examples of proxy implementations we have already implemented are:

simple proxy: This is the default implementation which always transfers all operations to the master.

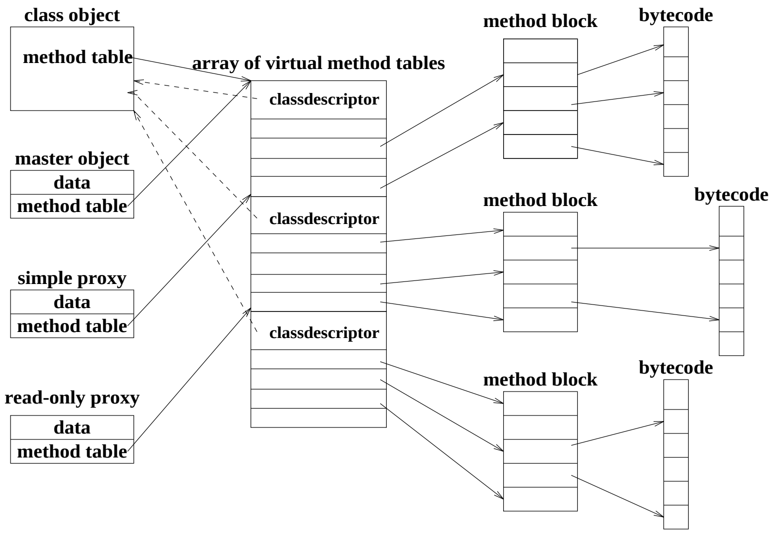

Figure 3. cJVM Object Model

read-only proxy: This implementation applies the operation locally, based on the fact that it is guaranteed to access only fields which are never changed (e.g., fields that are only written in the constructor) so the proxy maintains replicas of these fields.

proxy with locally invoked stateless methods: We consider a method as stateless if it does not access fields of its object. While, it is semantically correct to apply any kind of method directly on a proxy,[7] there is a clear performance gain in doing so for stateless methods, while in the general case, a method would have to access object fields remotely, overwhelming any performance gained by invoking this method locally.

These are representatives of a large set of possible implementations whose logic can range from actions that are always beneficial to performance (e.g., replicating read-only fields and stateless methods) to actions whose worthiness depends upon run-time conditions (e.g., caching an object at the node where it is being used).

As an example of one of such analysis consider the identification of methods which do not mutate the heap but work *only* on the stack. Such *stateless* methods can easily be detected during class loading by scanning the bytecodes of the method and determining that there are no field modification bytecodes. If such method is detected it can be marked as always executed locally since it does not affect the heap. See [18] for more details on proxy types and statistics.

Proxy implementations based on analysis of the code are not always sufficient for gaining the best scalability for applications. For example, we are currently designing a proxy implementation which determines whether to apply methods locally or remotely based upon run-time conditions. For example, in case a method accesses only static data, its implementation will determine, at run time, if the master object is co-located

[7] Since all bytecodes have been cluster enabled, it is semantically correct to execute the method containing them locally; the operation is shipped to the master object.

with the class's master object. If so, it will invoke the method remotely on the master object so the static data will be accessed locally. Otherwise, it will safely invoke this method locally, saving the overhead of a remote method call.

Thus we can construct proxies whose logic depends both upon profiling information and upon code analysis. This is exactly the advantage of working at the level of JVM, we can uniquely exploit both static and dynamic knowledge about specific objects, classes and about the Java language itself.

cJVM's object model (smart proxies mechanism) is significantly different from sub-classing in the following aspects:

- Sub-classing cannot be used to change the behavior of an existing object
- Sub-classing requires explicit programmer action to determine the subclasses and such actions violate cJVM's property of single system image (e.g. letting Java applications to run on a cluster runs without any source modifications)
- All proxies of the same subclass still behave the same (e.g. cache the same data) while with smart proxies, it really enables using a proxy on a per instance object basis.

Thus, sub-classing is a static mechanism (at the level of programmers) while smart proxies is at the level of run-time (VM) systems and is transparent to the application.

Thread Model

To gain scalability on a cluster, objects and threads need to be distributed among the cluster nodes to (1) utilize less loaded nodes while dynamically balancing load and (2) improve locality to other objects they access. cJVM, as we stated, uses a distributed heap; the thread model needs to allow accessing an object whose master copy is on another node. Naturally, this is not supported by a traditional JVM so we need to extend the thread model in the context of cJVM.

There are three main alternatives for allowing a thread to access a remote object. In thread migration, when a thread accesses a remote object, the execution of a thread is halted, its execution environment (e.g., program counter, stack) is serialized and moved to the host containing the master copy of the object, and the thread is then re-instantiated and its execution resumed. In object migration [14], threads never migrate. Rather, whenever a thread accesses a remote object (or a copy of the object) the object is brought to the node where the thread is executing. A third approach is method shipping [15] in which neither threads nor objects migrate. Instead, operations on proxies are redirected to the nodes where their master object resides and handled locally by special service threads. In cJVM, we chose method shipping as our base approach since it is easier to extend method shipping with object-shipping-like function (e.g., caching) than the opposite. Method shipping avoids all this. Method shipping is implemented internally as an RPC from one node to another and is used for remote operations like: remote method invocation, remote monitor operations and remote field access. It drastically differs from the standard RMI([9]) since it is transparently invoked at the level of the JVM.

In cJVM, we use method shipping for the following reasons:

1. In object migration, it is necessary to coordinate between multiple threads on different nodes using the same object. This can be expensive and difficult to

ensure in a language such as Java where multiple threads can read and write fields of the same object without explicit synchronization. (see [20] for a Java implementation that does this.) In addition, we believe it is harder to extend an object migration mechanism with caching and replication (as described in the previous section on Object Model) than it is to extend a method migration mechanism.

2. Thread migration is an appropriate mechanism to use when remote accesses are infrequent and coarse-grained. For such uses, the overhead of migration does not overwhelm the speedup gained by distributing the application's threads. However, the Java Server Applications which we aim at scaling do not have the characteristic of relatively infrequent and coarse-grained remote accesses. Rather, each invocation of a method on a shared object (we assume some degree of sharing) is a remote access, and in Java method invocations are often fine-grained computations.

The method shipping approach changes the application's view of a thread as being an active object executing its set of instructions on a single node. To maintain a single system image we need to provide implicit ways to enable remote thread creation. Otherwise threads will never be distributed over the cluster nodes. Further, given that we distribute threads and use method shipping, we need to maintain a uniform thread identification and uniform access to its stack even though the stack may be distributed.

Thread Creation

It is very important for us to distribute the application's threads since this is the only way cJVM can obtain scalability improvements. We thus need to consider how threads are created in Java. As describe in the BACKGROUND chapter, there are two ways in the Java language to specify the code to be executed by a Java thread. It is either specified by a subclass of the `java.lang.Thread` class or by a class implementing `java.lang.Runnable` (see figure 4), an instance of which is passed to the constructor of a `Thread`. In both cases, we need to be able to create the instances of these classes remotely to allow distribution of the application's threads. In the first case, the thread itself will be distributed via a load balancing function while in the later case, the thread will be co-located (see below) with the instance of the class specifying its code, distributed via a load balancing function.

Figure 5 shows the original bytecodes generated from the source code in figure 4. Objects are created by the `new` bytecode, whose single operand is a reference to the class for which an instance should be created. The set of bytecodes immediately after the `new` opcode, pushes the parameters (if any) and invokes the constructor of this new object (via the `invokespecial` bytecode).

To support remote thread creation, we modified the semantics of the `new` opcode. Specifically, when each class is loaded it is analyzed and marked as `java.lang.Runnable` if it implements the `Runnable` interface[8]. For every `new` opcode, we first check the class which is the parameter of the opcode. If the class is not flagged as `java.lang.Runnable` then the opcode `new` is rewritten to `new_quick`, following the

[8]The class `java.lang.Thread` implements the `java.lang.Runnable` interface

```
class Target implements java.lang.Runnable {
    public Target () {  }
    public void run() {  }
}
class Foo  {
     public Foo () { }
}
public class Test {
    public static void main (String[] argv) {
    new Foo();
    Runnable r = new Target();
    new Thread();
    new Thread(r);
    }
}
```

Figure 4. Bytecode Rewriting for Remote Thread Creation: source code

```
Method void main(java.lang.String[])
   0 new #1 <Class Foo>
   3 invokespecial #6 <Method Foo()>
   6 new #4 <Class Target>
   9 dup
  10 invokespecial #9 <Method Target()>
  13 astore_1
  14 new #3 <Class java.lang.Thread>
  17 invokespecial #8 <Method java.lang.Thread()>
  20 new #3 <Class java.lang.Thread>
  23 aload_1
  24 invokespecial #10 <Method java.lang.Thread(java.lang.Runnable)>
  27 return
```

Figure 5. Bytecode Rewriting for Remote Thread Creation: original bytecode

standard behavior of the JVM as described in [4]. This is the case for creation of the object of type `Foo` as seen in line 0 of figure 6.

If the parameter of the `new` implements `java.lang.Runnable`, then the `new` opcode is rewritten into `remote_new` pseudo bytecode (figure 6, lines 6, 14 and 20). `remote_new` is a pseudo-bytecode that is private to the implementation of cJVM. When executed, it determines the *best* node in the cluster on which to create the new `java.lang.Runnable`; this determination is based upon a pluggable load-balancing routine. It then sends this node a request to create the instance of the class at that node. A proxy for that instance is created at the node executing the opcode,[9] and its reference is pushed on the stack, allowing the application to behave as if the original new opcode was executed. Upon subsequent executions, each of the rewritten bytecodes will directly and correctly apply either local or remote object creation.

[9]Unless the node which is selected happens to be this node.

```
Method void main(java.lang.String[])
   0 new_quick #1 <Class Foo>
   3 invokespecial #6 <Method Foo()>
   6 remote_new  #4 <Class runnable>
   9 dup
  10 invokespecial #9 <Method runnable()>
  13 astore_1   <pop the target host to a temporary variable>
  14 remote_new #3 <Class java.lang.Thread>
  17 invokespecial #8 <Method java.lang.Thread()>
  20 remote_new  #3 <Class java.lang.Thread>
  23 aload_1 <push the target host onto the stack>
  24 invokespecial #10 <Method java.lang.Thread(java.lang.Runnable)>
  27 return
```

Figure 6. Bytecode Rewriting for Remote Thread Creation: rewritten bytecode

One case of thread creation which we cannot support with this localized bytecode rewriting is co-locating a thread with a given target Runnable (see last source line in figure 4). We are currently investigating ways to do such co-location based on method analysis which provides some information about static object connections.

Such analysis and bytecode rewriting demonstrate the capabilities, at the level of JVM to apply implicit solutions for cluster-enabled functionality.

Distributed Stacks

Applying method shipping causes the stack of a Java thread to be distributed across the multiple system threads (handling the remote operations) at different hosts. This is shown in figure 7 where the upward arrow in each node describe stack growth direction. Thus, the thread model of a traditional JVM needs to be modified to maintain the application's illusion of a thread running all its code on one host and its stack although distributed, can be accessed as single entity. Specifically, that new model should guarantee that:

- The Java thread's stack can be traversed even though it is distributed[10]
- A consistent and correct value is returned by `Thread.currentThread()`, regardless of on which host it was invoked[11].
- The Java thread is identified as the owner of monitors obtained by the frames of the distributed stack, regardless of which system thread was the one to enter these monitors.
- A change in the state (i.e., running or suspended) of a system thread involved in method shipping (i.e. due to methods like `stop()`), must apply to all of the system threads involved.

To enable traversing a stack and changing the state of system threads along the chain, all the local fractions of that stack are concatenated, as seen in figure 7. In

[10] A stack is accessed by the methods `printStackTrace()`, `countStackFrames()`, `fillInStackTrace()` and `dumpStack()`

[11] This Java method always returns a reference to the thread object which invokes it

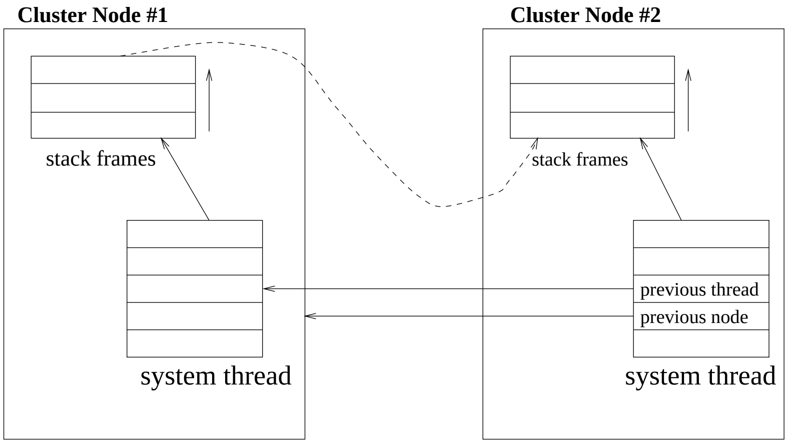

Figure 7. A Distributed Stack

addition, the implementation of those methods which changes a thread's state and access stack frames are modified.

To help maintain thread identification, every Java thread is assigned a logical identifier; this is a reference to the Java thread object which initiated the method shipping. We thus pass with every remote operation the global address of the Java thread initiating the remote operation. Due to our memory model (see section on Memory Model below) logical identifiers are translated in every host into local references to proxies to the master thread object or to that object itself (see example below).

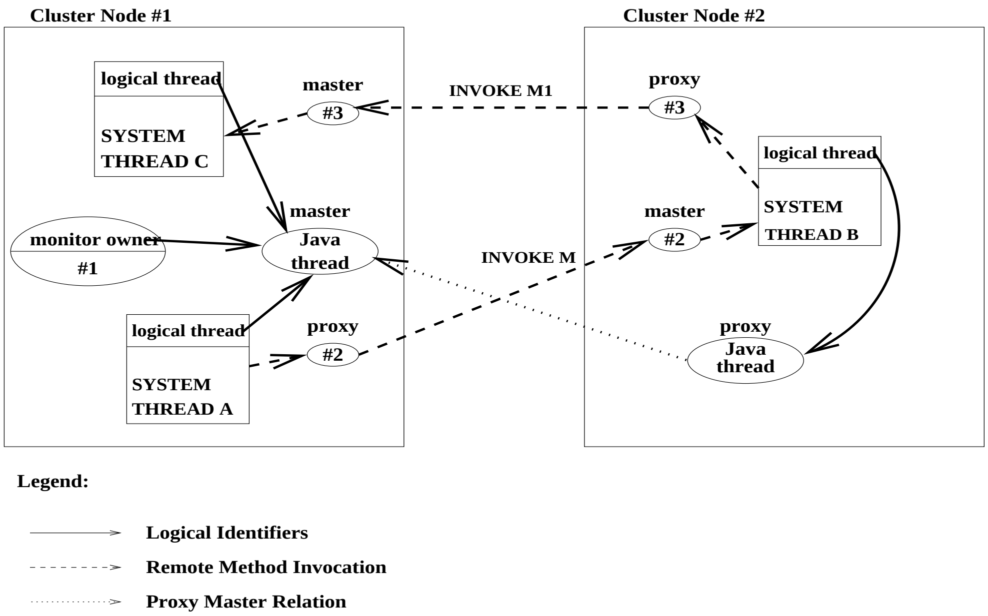

Figure 8. using Logical Identifiers in cJVM

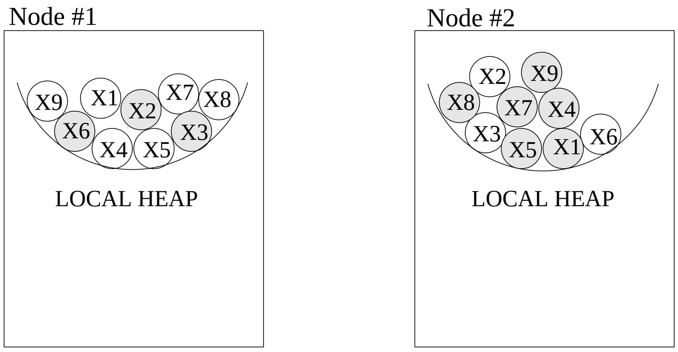

Figure 9. cJVM Distributed Heap

cJVM uses these logical identifiers to identify the owner of an object's monitor and to refer to current threads. Figure 8 demonstrate how logical identifiers maintain the thread identity over multiple hosts. In this figure dashed lines denote method invocations, dotted lines denote proxy-master relationship and bold lines denote logical identifiers. The figure shows a scenario in which a system thread A, bound to a Java application thread, invokes a method (M) upon proxy #2. Consequently, the method is shipped to node 2 and invoked upon the master object #2 by system thread B. During execution of the method, method (M1) is invoked upon proxy #3. Again, the method is shipped; it is applied by system thread C on node #1 where the master object #3 resides. With logical identifiers,

1. if `Thread.currentThread()` is invoked by thread B it will return a reference to the proxy of the Java thread implemented by system thread A, through which the master Java thread is referred.
2. although two threads are involved, system thread C is able to enter the monitor of object #1 which is already being locked by system thread A, since their logical identifiers are equal[12].

Memory Model

While the cJVM object model provides uniform access to proxies and master objects, a new memory model is required to (1) enable all proxies to locate and refer to the master object and (2) provide an application with an illusion of using a single monolithic heap.

cJVM maintains a distributed heap built on top of the local heap implementations found in a traditional JVM. Thus, new objects (both master objects and proxy objects) are allocated in the local heaps, the same way it is done in traditional JVMs. Each node independently manages its portion of the distributed heap. Figure 9 shows two cluster nodes with their local heaps. Objects are noted by empty circles while proxies are grayed out.

However, when a local object is passed, for the first time, as an argument to a remote node, it becomes global. A remote node may now invoke methods on this object and

[12]Recall that Java allows the same thread to acquired the same lock (monitor) more then once, while blocks other Java threads from acquiring the same monitor until it is freed.

```
resolve (GAO, GAC) returns <object reference>:
{
    local_object_ref = translate (GAO);
    if  (local_object_ref == null)
    { // no such master object or proxy is found.
         class_local_object_ref = translate(GAC);
         if (class_local_ref==NULL)
         { // class object is not found
            class_local_ref=<create a proxy to the master class object>;
         }
         // create a proxy for the object with a GAO global address.
         return new_proxy(class_local_ref,GAO);
    }
     return local_ref;
}
```

Figure 10. Resolution of a Global Address

access the data of this object; the remote node will create a proxy for the object. The local address of the master object and the local address of the proxy have no necessary relationship to one another. In order for the nodes to communicate about objects, objects which are used by more than one node are given *global addresses*.

A global address is an identifier of a global object which is unique over the cluster. It is generated the first time that object is passed as an argument to a remote operation by combining the node id (a number) and a node unique counter which is incremented each time its value is used. Every time an object is passed between two nodes in the cluster, we pass a two-tuple containing the global address of the object (GAO) and the global address of its class (GAC). At the target node, these two global addresses are resolved as shown in figure 10. In this figure, translate(GA) returns a reference to a local object (either a master or a proxy) with the same global address GA, if found; this translation is supported by a hash table.

As seen in figure 10, using a two-tuple of the form (GAO, GAC) we can construct a new proxy without additional messages to determine the class of the proxy, as would be expected if we sent only the global address of the object. However, when the same remote object is passed as argument more than once to the same node, the two-tuple form incurs a slightly higher overhead due to locating and packaging the global address of the class object, which is not needed. We still don't have enough experience to evaluate that tradeoff in real applications.

As stated above each node independently manages its portion of the distributed heap using its local garbage collection (GC). Distributed Garbage Collection is a well known difficult problem which we are not addressing in this research. We did however enhance the GC to free proxies while avoiding collecting master objects with global address (i.e., they may have a master or proxy on another node) even if no local objects refers to them since objects on other nodes may.

STATUS

Currently we have a prototype which runs pure Java applications on a cluster of IBM Intellistation stations running Windows NT, connected via a Myrinet fast switch. Our communication is via MPI [5] which we use for portability, implemented over HPVM [1]. The prototype provides a single system image of a Java Virtual Machine to applications, distributing the application's threads and objects over the cluster. The prototype supports all of the core classes with pure Java implementation and a portion of the core classes with native implementations. The prototype is based upon the Java 1.2 implementation from Sun for Windows NT.

On an embarrassingly parallel application, one with limited interactions between the applications threads, cJVM achieves super linear speedups on a small cluster. The super-linear speedup is not surprising since by using the cluster we are running fewer threads on a node and thus reducing underlying operating system contention. From this experiment, we learn that cJVM does not add significant overhead in those cases where interactions between the threads is not required.

We used cJVM to run an unmodified version of the Portable Business Object Benchmark (pBOB). pBOB is a kernel of business logic, inspired by the TPC-C benchmark specification. In accordance with the TPC's fair use policy, we note that pBOB deviates from the TPC-C specification and is not comparable to any official TPC result. pBOB creates warehouses representing customers, stocks orders etc. upon which multiple threads apply transaction. The idea is to increase the throughout of their threads (representing clients) by running these threads in parallel on multiple nodes.

We chose pBOB as it is a large (10Kloc), self-contained, pure Java application which only depends on the Java core APIs. Running pBOB on four nodes cJVM cluster we achieved 3.2 speedup as compared to running pBOB on one node, using the interpreter (not JIT) in both runs. See [18] for more details.

One item we have intentionally not addressed is supporting a Just-In-Time (JIT) compiler. The reason we have implemented cJVM without JIT support is due to limited resources; We do not believe there are any inherent reasons why cJVM could not be integrated with a JIT compiler, and we would like to add JIT support after our code stabilizes.

RELATED WORK

Java on distributed machines has been extensively studied since Java was announced. From a programmer's point of view, tools and infrastructure supporting Java applications on a cluster ranges from completely explicit solutions to implicit solutions similar to cJVM.

Explicit approaches [10, 9] assume an architecture of multiple JVMs while handling remote objects and threads at the level of the Java language and external frameworks. Most of these frameworks have very little relevance to cJVM. Above all, unlike cJVM, they do not hide from the application the complexity of distributing the Java application. From a technical perspective, they don't support transparent remote class objects. In addition, for example, proxy classes in Voyager or JAVA/RMI are created manually by the programmer. In addition, Voyager programs are always mapped to specific network configurations defined in advance (e.g., every node runs a daemon with a specific port). With DO! [3], special framework classes are used to restructure the Java program, to cluster objects to improve object locality.

Another work in this category (of multiple JVMs) with some relevance to cJVM is JavaParty [8]. It focuses on transparent support of remote objects in Java. The only extension to pure Java is the use of an explicit remote attribute to distinguish classes of remote objects. In JavaParty remote objects are created on nodes selected implicitly based on load, bandwidth or other criteria. JavaParty also allows the programmer to provide a placement strategy himself. In addition, it supports monitoring of objects' interactions during program execution and when appropriate, schedules object migration to enhance locality. To summarize, JavaParty does not completely hide the underlying network from the programmer; in other words the application has to be aware of the cluster, which is not the case in cJVM.

A final work in this category is ProActive [17] where a programmer can create an object on a remote node by invoking a utility method on a run time support class, providing the type of the object, the parameters to its constructor, and the node on which the object should be created. After creating the remote object, the programmer can treat it as a normal object; however, if the programmer uses Java's run-time introspection s/he will see the type of the proxy. As with the other approaches based upon a run-time library, ProActive does not support a single system image.

In contrast with the aforementioned frameworks, Java/DSM [13] is an implicit approach, at the level of infrastructure. It is a modified JVM whose heap is implemented in a distributed shared memory. Objects are allocated from the shared (DSM) region. Conceptually, Java/DSM supports SSI; its implementation, however, is incomplete. It supports most of the Java API but it does not support thread migration and a thread's location is not transparent. In addition it does not support `wait()` and `notify()` between threads on different processors although it does support synchronized methods. Compared to cJVM, using DSM technology (shipping pages of memory between nodes), it does not take advantage of Java semantics to gain better performance; on the contrary, it can potentially cause false sharing which degrades performance.

Another project which builds upon a cluster-enabled infrastructure is Hyperion [20]; this is probably the work most related to cJVM. This can be categorized as an implementation of a JVM on top of an object-based distributed shared memory; in some respects Hyperion uses a hybrid design, containing both elements of a cluster-enabled JVM and of a JVM on top of a cluster-enabled infrastructure. Hyperion takes a JIT-like approach, compiling Java bytecodes into C and then compiling the C code into machine code prior to execution.[13] Hyperion has several features in common with cJVM. The heap is distributed and there is a master copy of an object on the node where the object is created.[14] Objects are created on the node where `new` is executed, except for Java threads which are created on a node chosen by the system based upon a load balancing function. But there are several significant differences between cJVM and Hyperion. In Hyperion, each node is statically assigned a portion of the address space which it can use to create new objects, unlike in cJVM where each node is free to manage its heap independently. More importantly, Hyperion uses an object shipping model, in which a copy of a remote object is brought to the accessing node. The accessing node uses this local cached copy which is written back to the server at

[13] It is unclear if the compilation is done on-the-fly or must be done as an explicit step in building the program.
[14] In Hyperion they use the term *local* instead of *master*.

Java synchronization points as required by Java's memory semantics (see Chapter 17 of [19]). This homogeneous approach should be contrasted with cJVM's uses of smart proxies which allows e.g., different kinds of proxies for objects of the same class.

Compared with these works, the work on cJVM has its own uniqueness in terms of (1) full Java compliant implementation, that is single system image and (2) a novel object model supporting replications and caching of objects towards high performance.

CONCLUSIONS and FUTURE WORK

This paper presents an approach and implementation of a single system image for a JVM on cluster machines - cJVM. In its essence, there is an object model which distinguishes between an application view of an object (e.g. every object is a unique data structure) and its implementation (e.g. objects may have consistent replications), enabling to exploit knowledge on the usage of individual objects to improve performance (e.g. using object replications to increase locality of access to objects).

It should be noted that the mechanisms discussed in this paper, although tailored for Java semantics, can be utilized and are also valid in a broader context of distributed systems.

Having completed a prototype of cJVM that supports SSI, our next step is addressing the issues of defining and handling multiple kinds of implementations for objects towards gaining high performance of real Java server applications. To that end we plan to address several specific issues including: analyze application classes to detect behavior patterns (e.g., read only fields) where performance can be improved by using new types of proxies, and incorporate them in cJVM. Furthermore we would like to use run-time profiling to detect behavior change in objects and switch proxies during run-time for better performance. Another item we would like to improve is the performance of the communication layer to reduce latency times.

ACKNOWLEDGEMENTS

We would like to thank Oded Cohn, Tamar Eilam, Zvi Harel, Hillel Kollodner, Assaf Schuster and Yoram Talmor for their input to cJVM. Special thanks to Alain Azagury who initiated this research activity.

REFERENCES

1. *High Performance Virtual Machine User Documentation* August, 1997
2. Y. Aridor, M. Factor, A. Teperman, T. Eilam and A. Schuster *Transparently Obtaining Scalability for Java Applications on a Cluster* To appear in *JPDC Special Issue - Java on Clusters*, July 2000
3. P. Launay and J. Pazat *A Framework for parallel Programming in Java* IRISA, December 1997, 1154
4. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification* Addison-Wesley, 1997
5. http://www.mcs.anl.gov/mpi
6. http://www.microsoft.com
7. G. Pfister *In Search of Clusters: The Coming Battle in Lowly Parallel Computing* Prentice-Hall, 1995

8. M. Philippsen and M. Zenger JavaParty: Transparent remote Objects in Java Concurrency: Practice and Experience 1997, vol. 11, No. 9, 1125—1242
9. http://web2.java.sun.com/products//jdk/1.1/docs/guide/rmi/index.html
10. http://www.objectspace.com/voyager/
11. http://www.viarch.org/
12. http://www.myri.com/
13. A. Yu and W. Cox. Java/DSM A Platform for Heterogeneous Computing In *ACM 1997 Workshop on Java for Science and Engineering Computation* June,1997
14. E. Jul and H. Levy and N. Hutchinson Fine-grained mobility in the Emerald system In *ACM Transactions on Computer Systems* 1988, Vol. 6, No. 1, 109—133
15. A. Birell and B. Nelson Implementing Remote Procedure Calls In *ACM Transactions on Computers Systems* 1984, Vol. 2, No. 1, 39—59
16. A. S. Grimshaw Easy to Use Object-Oriented Parallel Programming with MENTAT In *IEEE Computer* 1993, Vol. 26, No. 5. 39—51
17. 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* 1998, INRIA Sophia Antipolis, France
18. http://www.haifa.il.ibm.com/projects/systech/cjvm.html The home page URL
http://www.haifa.il.ibm.com/projects/systech/cjvm_papers.html Papers on cJVM
19. J. Gosling, B. Joy and G. Steele *The Java Language Specification* Addison-Wesley, 1996 Ch. 17
20. M. MacBeth and K. McGuigan and P. Hatcher *Executing Java Threads in Parallel in a Distributed-Memory Environment* IBM Center for Advanced Studies Conference, Canada, November–December 1998
21. A. Gontmakher and A. Schuster *Characterizations for Java Memory Behavior* Intl. Par. Proc. Symp. (1st Joint IPPS/SPDP), March 1998, 682—686,
22. William Pugh Fixing the Java Memory Model In *ACM Java Grande Conference* June 1999, 89—98