Implementing Java on Clusters

Yariv Aridor, Michael Factor, and Avi Teperman

IBM Research Laboratory in Haifa
MATAM, Advanced Technology Center, Haifa 31905, Israel
phone: +972-4-8296350, fax: +972-4-8296114
{yariv,factor,teperman}@il.ibm.com

Abstract. We have implemented a virtual machine (VM) for Java which executes on a cluster. Our cluster VM completely hides the cluster from the application, presenting a single system image (SSI) (i.e., the application sees a traditional virtual machine). At the same time it leverages the cluster to achieve improved performance for a range of applications. We show how the flexibility and constraints of the Java Virtual Machine (JVM) Specification [7] impacted the design of our cluster VM. We describe issues related to class loading and distribution-aware implementations of the bytecodes. We also point out the limits on providing a solution for completely transparent distribution of multi-threaded Java applications if one does not modify the VM or the core classes.

1 Introduction

The Java Virtual Machine (JVM) Specification [7] is both abstract (i.e, the implementation is not constrained) and complete (i.e., all externally visible effects are totally specified). We have taken advantage of these two properties in implementing a virtual machine (VM) for Java which executes on a cluster. Our cluster VM hides the cluster from the application, presenting a single system image (SSI) (i.e., the application sees a traditional virtual machine); at the same time it leverages the cluster to improve performance for a range of applications.

By implementing a single system image of a VM on a cluster, we have learned a great deal about the limits of solutions for distribution that do not change the VM. Specifically, a solution that allows *arbitrary* multi-threaded, legacy applications to *transparently* leverage a cluster can be mostly built on top of an existing JVM; however, certain critical changes must be made to the JVM and the core classes to achieve complete SSI and to achieve acceptable performance.

We previously have introduced our cluster virtual machine (VM) for Java[1] and presented detailed performance results [2]. Here, we put particular emphasis on class loading because it is difficult to ensure SSI for class loading on a cluster. In this context we show why SSI cannot be achieved without VM modifications. We also discuss the `get` and `put` bytecodes as an examples of the bytecodes which need cluster-aware implementations. A longer version of the paper discusses the remaining bytecodes [1].

[1] Previously known as "cJVM".

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 722–732, 2001.

This work is interesting not only in its own right but also for what it tells us about the JVM specification. Our observations include: 1) without modifying the VM, one cannot provide SSI for class loading in a distributed VM implementation, 2) the flexibility inherent in opaque object references makes it easy to hide the cluster from an application and 3) symbolic evaluation, which is made possible by the complete specification of the behavior of the bytecodes, is useful for eliminating the overhead of distribution and not only for verification and traditional compiler optimizations.

The next section summarizes the architecture of our cluster VM. We discuss class loading in Section 3 and the byecodes in Section 4. We give performance results and describe the state of our prototype in Section 5. After discussing related work in Section 6, we present our conclusions and discuss future work.

2 Architecture

Our cluster virtual machine runs on a collection of independent computers connected by a fast interconnect. A cluster VM process resides on each computer. Each process contains a subset of the application's threads and objects. When taken as a whole, the set of processes constitutes the VM.

Figure 1 shows the basic architecture of our system. The top half of the figure shows the application's perspective; from this point of view, the cluster is completely hidden and we present a complete single system image (SSI). The lower half of the figure shows the implementation of our virtual machine. The implementation is aware of the cluster.

The implementation distributes objects and threads. To allow access to an object located on another node we use proxies. This applies to all objects including class objects and threads. The *master* is the object's authoritative copy and

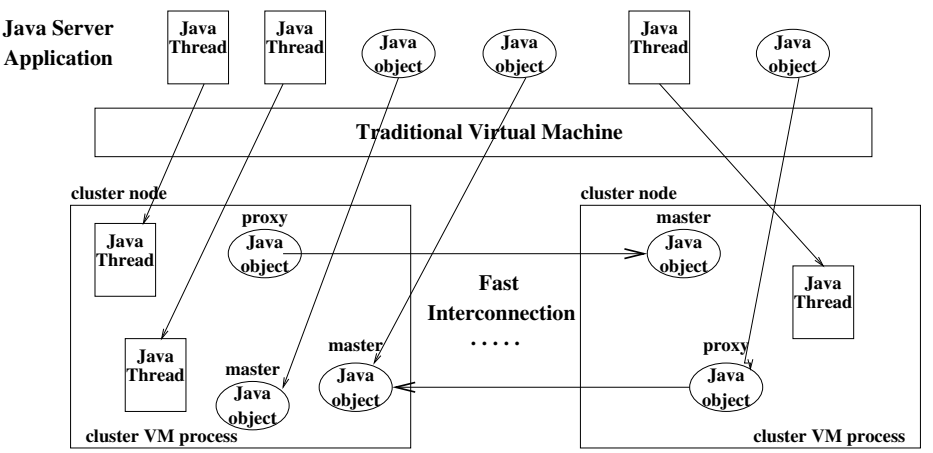

Fig. 1. Architecture of our cluster VM for Java

in general exists where the object was initially created - the *master node*. Note, the location of the master of an instance object is independent of the master node for the instance's class.

Every object has one master and possibly multiple proxies. We have developed *smart proxies* [2] which enable us to hide the master-proxy distinction from the application. Smart proxies also support caching and other behaviors beyond simply remoting an access to the master. Smart proxies work by allowing multiple implementations of a method to be associated with a single class.

Object placement is determined at run time on a per-instance basis and is driven by actual run-time usage. In addition to placing individual objects, we also replicate both objects and individual fields (see [2]).

We distribute computation using two mechanisms. First, when a thread is created, we decide, based upon a pluggable load balancing function, where the thread should be created. Second, our default means of accessing a remote object is *method shipping*, whereby we bring the thread to the master of the object it is accessing. Note, as we describe in [2], we do not always use method shipping.

3 Class Loading

Class loading[2] is complicated in a cluster-aware implementation of a virtual machine. On the one hand, in a shared-nothing distributed implementation, such as ours, for performance reasons the code for a given class needs to be locally available on all nodes where it will be executed; remotely accessing the code would add too great overhead. Replicating the code requires replicating the internal data structures of the class such as method blocks, run-time constant pool, etc. On the other hand, as we elaborate below, it violates SSI, and thus is incorrect, to allow each node to independently load each class it needs to execute. Our focus for class loading is on functional correctness, i.e., SSI, and not performance; we assume that the cost will be amortized over a significant period of usage.

Class loading consists of the steps shown in the leftmost column of Table 1. While some of the these steps are idempotent, most are not. For those steps which are not idempotent, executing the step on each node independently, i.e., more than once, violates SSI. The third column explains how each step impacts the ability to achieve SSI in an implementation of a cluster VM.

The fact that application code can be involved in class loading, via an application class loader, is one of the prime reasons that SSI for class loading is impossible to achieve without VM modifications. For instance, the application can be aware of how frequently it is called, and thus, can detect if it is called a different number of times than on a traditional implementation.

The problems that can occur when each node loads classes independently show the limitations in providing complete SSI using a framework implemented in Java. Because all frameworks for distributed programming in Java [11,12,14]

[2] We use the phrase "class loading" generically to refer to the three steps of loading, linking and initializing.

Table 1. Phases of class loading and their impact on SSI

Step	Impact	Explanation
find external representation (loading)	not idempotent	Since the external representation may change over time, loading it independently on multiple nodes can lead to different representations being loaded which would violate SSI. In addition, since this function is performed by the method `loadClass` which can be overridden by the application, the application can be aware of how many times the external representation is requested.
create internal representation (loading)	idempotent	The internal representation depends only upon the external representation. While this function is performed by `ClassLoader.defineClass` which an application class loader can override, the internal representation is created by the (native) implementation provided by the VM as part of `java.lang.ClassLoader`; thus, the application need not be aware how many times an internal representation is created.
create `Class` object (loading)	not idempotent	There must be a single master copy of the class object on one node, with other nodes having proxies. The VM-provided implementation of `ClassLoader.resolveClass` performs this function.
verify (linking)	idempotent	Verification depends only upon the internal representation. The VM-provided implementation of `ClassLoader.resolveClass` performs this function.
prepare (linking)	not idempotent	Executing this step more than once could allow an application to see the default value on one node after the application initialized the class and wrote a value on another node. The VM-provided implementation of `ClassLoader.resolveClass` provides this function.
resolve (linking)	idempotent	Given the same internal representations two nodes will reach the same conclusion in resolving a symbolic constant pool entry.
initialize	not idempotent	Executing `<clinit>` more than once violates SSI. It can break memory coherence (e.g., an application seeing the result of the class initialization on one node after it wrote a value on another node), and the application can count how many times the class initializer executed.

independently load a class on every node where it is used, they are inherently limited in how close they can come to achieving a single system image.

If one is willing to work at the VM level, there are several ways to support SSI for class loading. The approach we took is to use the mechanisms as implemented in a JVM for a traditional platform, modifying the way these mechanisms are invoked to address the issues from Table 1. One alternative is to completely load the class on one node and then transfer the internal representation to the other nodes. While we experimented with this alternative, it has two difficulties. The first, purely technical issue, is that due to the complicated graph structures

and inter-class references, it is hard to cleanly determine which data needs to be replicated and how to copy this data. The second is that given the lazy nature of some of the steps of class loading it is unclear when the class should be replicated.

The master node for a class C is the node which is the master for the class loader L_C which is to be used to load C; for system classes we use an arbitrary fixed node as the master. In our approach, the master executes those aspects of class loading which are not idempotent. All nodes that use the class execute those aspects which are idempotent. Other than the master, a node loads a class when it first needs the class, e.g., to create an instance (either master or proxy); the master loads the class the first time it is needed by any node in the cluster. The master is also the authoritative source for values of static fields; although, we do make extensive use of caching (see [2]).

When the master node loads a class, it follows the same flow as a traditional JVM implementation. The interesting scenario is when a node n_p, which will have a proxy of class C needs to load C. In this case, n_p sends a message to node n_m, the master node for L_C. n_m will be the master for C. The message requests that, unless the class is already loaded, n_m call `loadClass` on L_C to load the class. This eventually leads to a call to `defineClass` on `java.lang.ClassLoader`; `defineClass` is passed the binary image of the external representation for the class. Node n_m completes the processing of the message by returning the binary image to n_p.[3]. Node n_p then calls the implementation of the `defineClass` from `java.lang.ClassLoader` to create the internal representation on n_p.

We also modify `defineClass` to analyze the method's bytecodes; this analysis is used to construct the smart proxy implementations. We perform this analysis on each node in our implementation; although depending upon the tradeoffs between computation and communication, we could perform this analysis only on the master and send the results to the proxies.

Once the internal representation is created, the class needs to be linked, i.e., verified, prepared and possibly resolved. The class's master executes the normal flow to verify the class. On a node containing a proxy, we have two options depending upon relative costs. We can either verify the class (which is correct but expensive) or send a message to the master requesting that it link (and verify) the class. We are currently experimenting with local verification.

To support caching, we allocate memory in proxies of `Class` objects as well as in the master.[4] Thus, we prepare the static fields on all nodes. Preparation, however, is not idempotent; we address this problem by being careful in how we read and write static fields. In particular, a proxy node prior to reading a local copy of static field will pull the value of the field from the master, after either verifying that the master has been initialized or initializing the master if necessary. Any writes to the static field are always performed at the master and invalidate any cached copies of that field.

[3] To allow returning the external representation even if the class was already loaded, we persistently associate the external representation with the class object.

[4] To be precise, to support caching we allocate memory in all proxies.

We ensure that classes are initialized at most once by only running class initializers on the class's master node. We mark a proxy as initialized only after its master has been initialized.

Resolution in our implementation occurs on-demand and not during the linkage phase. Resolution is idempotent, since all nodes see a consistent view of the internal representation of classes and since classes are initialized at most once.

Since all classes loaded by a given class loader have the same master node, we have a potential bottleneck which can have two embodiments: accesses to static fields and static method invocations. By default we ship accesses to a proxy's fields and invocations of its methods to the master; however, we avoid these bottlenecks by providing heavy caching of static fields and executing static methods directly on the proxy.

4 Bytecodes

In building a cluster virtual machine – a virtual machine which looks to the application like a traditional JVM – a critical aspect is determining which bytecodes need to have a cluster-aware implementation. For lack of space this section looks only at `put/get` bytecodes showing how they are made cluster-aware and how they can be optimized for performance. The interested reader is refered to [1] for full discussion on other bytecodes such as object creation, method invocation, exception throwing and handling and synchronization.

4.1 Accessibility

Our master-proxy model provides accessibility to objects independent of their physical location. In more detail, the implementation of all `get`s and `put`s contains a *barrier* which determines if the operation is being performed on a master or a proxy as shown:

```
ObjectRef o = popStack();
if (cJVM_isMaster(o))     // barrier
   <do normal flow>
else
   <perform Remote Bytecode (RBC) >
```

As shown, if the operation is being performed on the master, the operation proceeds as in a traditional implementation of the JVM. To handle operations performed on a proxy, we have implemented a remote bytecode (RBC) mechanism to send a message to the master node for the target object; a server thread, which is part of our cluster VM implementation, handles this message.

4.2 Performance

The reason we changed the VM to support `get` and `put` operations on a cluster was performance. As we described above, to support remote accesses, we need

to determine if an instance is a master or a proxy. We cannot make this determination by using different classes for the master and proxy, or by adding a field to flag the object as proxy, as the introspection APIs would make this visible to the application, violating SSI. In our VM, we made this determination by modifying the implementation of the handle, the mechanism used by the JVM to reference the object. If we modify neither the VM nor the core classes, we will need to access an auxiliary data structure to determine if the object is a master or a proxy. It is fairly obvious that such overhead would be completely unacceptable.

Furthermore, simply modifying the VM is insufficient to achieve good performance. The algorithms described above have three performance aspects that require further consideration. First, we need to ensure that execution of a remote bytecode is rare. Second, we need to ensure that we do not pay too great overhead for the barriers.[5] Third, we need to reduce the cost of a RBC.

Given our use of method shipping, we only access a field of a proxy if an application makes an unencapsulated access. Method shipping ensures that encapsulated accesses are executed against the master. Thus, well-structured code should have only a small number of remote `get`s and `put`s.

To reduce the number of remote accesses further, we employ several caching mechanisms. First, we cache all final fields. We also cache all static fields based upon the empirical observation that these fields are rarely modified [2]. Finally, we replicate fields which we heuristically determine to be effectively read-only, i.e., we believe they will not be written in the given run of the program.

The use of barriers can be greatly reduced, at least for `getfield` and `putfield`, given the observation that most accesses are encapsulated; [5] made a similar observation. The target object of an encapsulated access is the `this` reference which is specified to be stored in the zeroth local variable. Therefore, for code executed against masters, if we prove that the access is targeted against the zeroth local variable, we can eliminate the barrier.[6]

Proving that the target of getfield is `this` is fairly easy. Assuming no prior `astore_0`, the target of the `getfield` is `this` in the following code snippets:

```
aload_0
getfield
```

or

```
aload_0
dup
getfield
```

These and other very simple pattern-based, static analyses account for anywhere from 64% to 80% of the occurrences of `getfield` as measured for our

[5] As was observed in the field of garbage collection [6], read barriers can be prohibitively expensive.

[6] In implementation terms, one way to eliminate barriers is in a JIT which generates different code depending upon whether a barrier is needed.

Table 2. Description of Benchmarks

Program	Description
pBOB	A business logic kernel. Creates m threads which generate transactions concurrently. $m = 4$ is used for measurements.
N-Body	Simulates the motion of n particles due to gravitational forces, over ts number of simulation time steps, using m threads. $n = 640$, $m = 4$ and $ts = 10$ are used for measurements
TSP	Find, in parallel, the shortest route among n cities using m threads. $n = 14$ and $m = 4$ are used for measurements

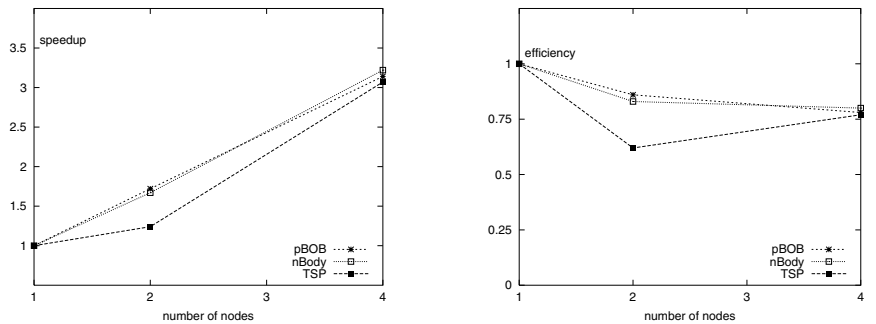

Fig. 2. Speedup and Efficiency

benchmark applications (set Table 2) and around 75% of the `getfields` in the Spec98JVM benchmarks [13]. Based upon dynamic data collected from an instrumented JVM, roughly 85% of the uses of getfield are for the `this` pointer.

5 Status

Our cluster VM for Java prototype runs on a cluster of IntelliStations running Windows NT and connected via a Myrinet fast switch [10]. In our prototype we have modified the interpreter loop of the Sun reference implementation of JDK 1.2; we have implemented all of the features we described except for some aspects of class loading and `this` elimination. Even though our prototype runs on Windows NT, our code is not NT specific and can easily be ported to other operating systems.

We have run several benchmarks as shown in Table 2.[7] Figure 2 shows the speedup and the efficiency for each of these benchmarks.

[7] Detailed description of the benchmarks can be found in [2]

6 Related Work

The tools and infrastructure supporting Java applications on a cluster range from completely explicit solutions to implicit solutions similar to our VM. Explicit approaches [14,12,11,3] assume multiple JVMs, handling remote objects and threads at the level of the Java frameworks. To varying degrees, these frameworks do not support SSI, e.g., the reflection APIs can detect proxies, local and remote method invocations have slightly different semantics, class loading does not present a single system image, etc.

In contrast with the aforementioned frameworks, are approaches based upon modified VMs. Those that we are aware of (other than ours) either build upon underlying support to hide the cluster and/or extend the semantics of Java.

Hicks, et al., [5] support distributed Java applications by extending Java with specialized operations to create remote objects. This work has a great deal of similarity to our work. For example, they only run class initializers on one node (although it is unclear how they load classes on other nodes); they use method shipping and avoid barriers prior to method invocations through the use of multiple method tables; and they use barriers prior to `get`/`put` instructions except for encapsulated accesses (although they do not make use of caching in proxies). However, our goals were slightly different – they aimed to support distributed applications written in extended Java whereas we aimed to push the limits of transparently distributing unmodified, multi-threaded Java applications.

JESSICA [8] is a modified JVM focusing on thread migration for purposes of load balancing. While it leveragess the semantics of the language to achieve a single system image, performance in JESSICA is obtained from a low-level, usage-neutral DSM system and not by leveraging the JVM semantics.

Hyperion [9] is an implementation of a JVM on top of an object-based, distributed shared memory. It is based on 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 origin at synchronization points. It should be contrasted to our use of method shipping enhanced by caching provided by smart proxies.

7 Conclusion and Future Work

This paper has described how we have leveraged the flexibility of the JVM specification to provide an efficient implementation of a VM on a cluster e.g., by the use of opaque object references as opposed to memory addresses and the use of symbolic evaluation to eliminate barriers.

We have also described the extent to which it is required to modify the VM or core classes to provide a solution for distributing an application with 100% transparency. Our analysis showed that primarily due to the presence of application class loaders, without modifying the VM, one cannot provide a SSI for class loading. We described our approach for class loading based upon leveraging the mechanisms from a non-distributed JVM implementation, but changing the way these mechanisms are used to avoid problems of non-idempotence.

Our approach to implementing a cluster VM for Java was invasive, changing the implementation of the bytecodes and classloading. While such an approach promises maximum performance (e.g., take advangate of semantics) it has downsides in terms of a propriety implementation of an entire VM, poor portability, and the necessity for multiple implementations for both an interpreter loop and a JIT. In this context, we are now considering an alternative approach that will replace the implementation of bytecode with techniqes such as aggressive code rewriting (e.g., as in JavaParty [11]), a Java utility class with native support to access the cluster (e.g., as in [3,4]) or a compiler-based approach to access the cluster directly (e.g., as in JaguarVIA [15]).

References

1. Yariv Aridor, Michael Factor, and Avi Teperman. Implementing Java on Clusters. Technical report, IBM Research Laboratory in Haifa, 2001. 722, 727
2. Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. Transparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing*, 60(10):1159–1193, October 2000. 722, 724, 726, 728, 729
3. Denis Caromel and Julien Vayssiere. A Java framework for seamless sequential, multi-threaded, and distributed programming. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998. ACM. 730, 731
4. Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998. ACM. 731
5. Michael Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore, and Cristian Ungureanu. Transparent communication for distributed objects in Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 160–170, Palo Alto, CA, 1999. ACM. 728, 730
6. Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, England, 1996. 728
7. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. 722
8. Matchy J. M. Ma, Cho-Li Wang, Francis C. M. Lau, and Zhiwei Xu. JESSICA: Java-enabled single-system-image computing architecture. In *Proceedings of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume VI, pages 2781–2787, Las Vegas, NV, June 1999. 730
9. Mark W. MacBeth, Keith A. McGuigan, and Philip J. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *Proceedings of the IBM Centre for Advanced Studies Conference*, Toronto, Canada, 1998. 730
10. http://www.myri.com/. 729
11. Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. 724, 730, 731

12. http://java.sun.com/products/jdk/rmi/index.html. 724, 730
13. http://www.spec.org/osg/jvm98. 729
14. http://www.objectspace.com/products/voyager/. 724, 730
15. Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12:519–538, December 1999. Special Issue on Java for High-Performance Applications. 731