International Journal of Parallel Programming, Vol. 34, No. 2, April 2006 (© 2006)
DOI: 10.1007/s10766-006-0007-0

A Platform-Independent Distributed Runtime for Standard Multithreaded Java

Michael Factor,[1] Assaf Schuster,[2]
and Konstantin Shagin[2,3]

JavaSplit is a portable runtime environment for distributed execution of standard multithreaded Java programs. It gains augmented computational power and increased memory capacity by distributing the threads and objects of an application among the available machines. Unlike previous works, which either forfeit Java portability by using a nonstandard Java Virtual Machine (JVM) or compromise transparency by requiring user intervention in making the application suitable for distributed execution, JavaSplit automatically executes standard multithreaded Java on any heterogenous collection of Java-enabled machines. Each machine carries out its part of the computation using nothing but its local standard (unmodified) JVM. Neither the programmer nor the person submitting the program for execution needs to be aware of JavaSplit or its distributed nature. We evaluate the efficiency of JavaSplit on several combinations of operating systems, JVM implementations, and communication hardware.

KEY WORDS: distributed computing; java; bytecode instrumentation; distributed shared memory.

1. INTRODUCTION

Interconnected collections of commodity workstations can yield a cost-effective alternative to dedicated parallel computers for executing

[1]IBM Research Lab in Haifa, Haifa University Campus, Haifa 31905, Israel. E-mail: factor@il.ibm.com
[2]Computer Science Department, Israel Institute of Technology, Technion City, Haifa 32000, Israel. E-mail: {assaf,konst}@cs.technion.ac.il
[3]To whom correspondence should be addressed.

computation-intensive applications. Message passing programming in this environment, however, is far from easy. The programmer's task can be significantly simplified if the programming framework provides a shared memory abstraction.

In the years following its introduction, Java has become widely accepted. Due to its built-in support for multithreading and synchronization, Java is unrivaled when it comes to the convenient construction of parallel applications with shared memory abstraction (henceforth *multithreaded applications*). However, these programs execute on a single Java Virtual Machine (JVM), which traditionally assumes an underlying shared memory machine.

Several works have devised a distributed runtime for Java.[1–9] Some works[2,8,9] install a customized cluster-aware JVM on each participating node. Others, such as[1] and,[7] compile the code of a given application into native machine code while adding distributed shared memory (DSM) capabilities. Both these approaches sacrifice one of the most important features of Java: the cross-platform portability. Existing systems that are able to execute a Java program on top of a collection of standard JVMs[3–6] are not transparent. Some works, such as[3] and,[4] introduce unorthodox programming constructs. Others, such as[5] and,[6] require user intervention to enable distributed execution of existing programs.

We present JavaSplit, a distributed runtime for Java, which, in contrast to previous works, combines transparency with portability. It executes a standard multithreaded Java program written for a single standard JVM on any given heterogeneous set of Java-enabled machines. It distributes the threads and objects of a given application among the available machines. To accomplish this, JavaSplit automatically instruments the program to intercept events that are interesting in the context of distributed execution, e.g., accesses to shared objects, synchronization, creation of new threads, and I/O operations. Shared data is managed by a DSM. The instrumentation combines the original application with the runtime logic (e.g., DSM), which is implemented in pure Java.

Each runtime node carries out its part of the distributed computation using nothing but its local standard unmodified JVM. Unlike systems that utilize specialized networking hardware [1,7] JavaSplit employs IP-based communication, accessing the network through the standard Java socket interface. The use of standard JVMs in conjunction with IP-based communication enables JavaSplit to perform computation on any given heterogenous set of machines interconnected by an IP network, such as the Internet or the intranet of a large organization. In addition, each node can locally optimize the performance of its JVM, e.g., via a just-in-time compiler (JIT).

Neither the programmer nor the person submitting the application for execution needs to be aware of JavaSplit or its distributed nature. This significantly contributes to JavaSplit's usability. First, any Java programmer can write applications for JavaSplit. There is no need to use nonstandard libraries or unconventional programming constructs to create an application. Second, it allows distributed execution of preexisting programs without modifying them manually. Finally, unlike in[5] and,[6] the user does not need to know the structure of an application in order to execute it in a distributed fashion.

Due to its enhanced portability and use of a well-known parallel programming paradigm, JavaSplit enables rapid development of distributed systems (such as distributed servers) composed entirely of commodity hardware. Scalable design allows JavaSplit to utilize a large heterogeneous collection of machines. Given certain security permissions, the Java Applet technology enables new nodes to join the runtime simply by pointing a Java-enabled browser to a Web page containing the applet executing the code of a worker node. Since browsers often use a built-in JVM, this would be impossible in systems with customized JVMs.

We evaluate the performance of JavaSplit in several different settings. In our experiments we employ workstations running on Windows XP and Red Hat Linux, using various standard JVMs: (i) Sun JDK 1.4.2 for Windows, (ii) IBM JDK 1.3.0 for Windows, and (iii) Sun JDK 1.4.1 for Linux. The utilized communication hardware is 100 Mbps Ethernet and 10 Gbit Infiniband. For portability considerations, the latter was used in IP-over-Infiniband (IPoIB) mode, producing only 2.86 Gbps throughput.

The portability and ease of use of JavaSplit is best demonstrated by an experiment in which we have successfully executed a multithreaded Java program on a set of workstations with all the possible combinations of operating systems and JVM implementations mentioned above. The participating stations did not need to install any software other than a thin Java client, which could be incorporated in a screen saver or in a Web page applet. It is only required that the machines receive the bytecodes of the application and the runtime modules, which can be accomplished through the customizable Java class loading mechanism.

The main contributions of this work are as follows. First, we develop bytecode instrumentation techniques that allow distributed execution of a standard, possibly preexisting Java application. Second, we extend a line of works in the area of distributed shared memory by improving the scalability of a well-known DSM protocol. Finally, we create a convenient and portable infrastructure for distributed computing.

The rest of the paper is structured as follows. Section 2 gives an overview of JavaSplit. JavaSplit memory management is presented in Section 3.

In Section 4 we describe our class file instrumentation techniques. The performance results are presented in Section 5. Section 6 discusses related systems. We conclude in Section 7.

2. JAVASPLIT OVERVIEW

The JavaSplit runtime administers a pool of worker nodes. An external resource management system may be employed to discover available machines and add them to the pool. Alternatively, an available machine can explicitly request to join the runtime. This request can be placed by simply visiting a web site that contains a Java applet whose code (automatically) initiates machine's enrollment to the worker pool. A worker node that has been assigned to execute application threads may leave the pool only when these threads terminate. If a node leaves the pool before the application finishes, it must transfer all the shared objects created by its threads to an active worker.

Each worker may execute one or more application threads. The computation begins by starting the application's *main* method on an available worker. Each newly created thread is placed for execution on one of the worker nodes, according to a plug-in load balancing function. (Currently, we use a function that places a new thread on a worker with the least number of active application threads.)

JavaSplit instruments all classes used by the application submitted for distributed execution and combines them with the runtime modules. Thus, the code becomes aware of the distributed nature of the underlying environment. JavaSplit uses a novel instrumentation technique, described in Section 4, which allows transformation of both user-defined classes and Java *standard library classes*, including the library classes with native methods. The instrumentation may be performed at run time, when the class is loaded into the JVM, or before the execution begins. In any case, during the distributed computation, all participating JVMs use the instrumented classes instead of the originals.

The employed instrumentation technique allows interception of the I/O operations. Therefore, it is possible to reimplement them in an arbitrary fashion. Since handling of I/O in a distributed environment is not our primary research goal, the current implementation simply redirects all I/O calls to a single node.

To support the shared memory abstraction, JavaSplit incorporates a DSM. The design choice of using a DSM (*data shipping*) rather than employing the master-proxy model (*method shipping*) is orthogonal to the bytecode instrumentation approach. In Java, instrumentation allows interception of accesses to shared data (required to preserve consistency of the

DSM in data shipping) as well as redirection of methods calls to remote machines (needed in method shipping).

3. DISTRIBUTED SHARED MEMORY

The DSM incorporated in JavaSplit is object-based. The object-based approach fits well on top of the object-based memory management of a JVM. By contrast, the *page-based* approach would be quite unnatural in the current context, since the hardware paging support for detection of memory accesses is unattainable without modifying the JVM. In addition, allocation of multiple objects on the same page would result in *false sharing*.

Any shared memory, distributed or not, is a subject to a set of constraints, which constitute the *memory model*. For example, the *Java Memory Model* (JMM) sets constraints on the implementation of shared memory in the context of a single JVM. Currently, the JMM is being redefined.[10] The new JMM is expected to be based on the principles of *Lazy Release Consistency* (LRC).[11] Therefore, we employ a DSM protocol that complies with LRC. Since there is a natural mapping for Java volatile variables to the release-acquire semantics of LRC in the revised JMM, we encapsulate accesses to volatile variables with acquire-release blocks.

We use a novel DSM protocol that is inspired by the state-of-the-art *Home-based Lazy Release Consistency* (HLRC)[12] protocol. In home-based protocols, each object is assigned to a node that manages the object's master copy. The threads that need to access the object create local (cached) copies, derived from the master copy. At certain synchronization points, thread modifications are flushed from the cached copies to the master copy. Our DSM protocol refines the scalability of HLRC and other existing home-based LRC protocols, such as,[13] by restricting the size of the *timestamp* attached to each coherency unit (CU) (object in the context of our system).

To reduce overhead, our DSM dynamically classifies objects into *local* and *shared*, managing only the latter. Shared objects can be accessed by several threads, whereas local objects cannot. An object is considered *local* until a reference to it needs to be shipped to another node. In this case, the object receives a globally unique ID, henceforth *global ID*, and is registered with the DSM. Discovery of shared objects is incorporated into the JavaSplit object serialization procedure. A reference (contained in an object) is serialized into a global ID of the target object. If the target object does not have a global ID (i.e., it is local), it gets one and becomes shared. (In a shared object, a global ID is stored in a field added by the instrumentation. In a local object, this field contains an illegal

value.) Dynamic detection of shared objects contributes to the system's performance because the maintenance cost of a local object is lower. It also allows local objects to be collected by the standard garbage collector. Since in many Java applications only a small portion of objects is shared,[14] the gain can be significant.

3.1. Refining Scalability of Home-based Protocols

Home-based implementations of LRC[12,13] associate a vector timestamp with each copy of a CU and each CU modification record, called *write notice*. The timestamps ensure that a version not older than required is brought from home. They also allow checking whether the required version is locally available before fetching it from the home node. Traditionally, each timestamp is a vector of integers with a number of entries equal to the number of threads (or nodes) in the system. Conceptually, such a timestamp is similar to a snapshot of a global *vector clock*.

A timestamp is attached to each valid CU and each write notice. Therefore, in a large-scale environment vector timestamps can occupy a considerable amount of space. Moreover, since nodes often exchange the timestamps of modified CUs, the timestamp size has a nonnegligible impact on communication overhead. In a Java-based environment such as ours, both the space and communication overheads are even more significant, because there because there are much more CUs (i.e. objects) and consequently a greater number of write notices.

We reduce the timestamp to a fixed-size value. In our implementation a timestamp consists of the ID of the most recently (locally) released lock and a counter of release operations associated with that lock. Only timestamps with the same lock ID are comparable. However, since in most cases the same lock is used to protect accesses to a particular object, the timestamps of an object's copies are usually comparable. Unlike the scalar timestamp scheme mentioned in,[15] our protocol does not require that during lock transfer a thread block until all recent modifications are flushed to the home nodes.

3.2. Distributed Synchronization Mechanism

In JavaSplit, the queue of a lock is managed by its current owner and is passed along with the lock ownership. This differs from most systems, which keep the lock's queue distributed among the nodes that wish to acquire it. Each lock is managed by the home node of the associated object. All lock requests are sent to the home node of the lock, which forwards them to the current owner. The current owner adds the forwarded

requests into the queue. Since there is little lock contention in Java,[16] the size of the queue is usually not too big and therefore the communication overhead of its transfer is negligible. The proposed algorithm has two advantages in the context of Java. First, it supports thread priorities: the owner always needs to pass ownership to the requester with the highest priority. Second, the operations *wait*, *notify*, and *notifyAll*[17] require no communication and are completely local.

To implement *wait*, *notify*, and *notifyAll*, a *wait queue* is managed alongside the regular *request queue* and is passed from releaser to acquirer. Since the above operations in Java can only be performed by the current owner of a lock and should affect only the request queue and the wait queue, there is no need for communication when performing them. For example, a thread T performing a *notify* operation on a lock L transfers a lock request from L's wait queue to L's request queue. Since T has to be the current owner of a lock, both queues of L are guaranteed to be locally available.

4. CLASS FILE INSTRUMENTATION

In JavaSplit, the main goals of instrumentation are (i) distributing of the threads and objects of an application among the worker nodes and (ii) preserving consistency of data accesses and lock operations. In order to achieve these goals, JavaSplit must instrument any class used by the original application, including the Java *standard library classes*, also known as *system classes*.

Due to their special status within the JVM, system classes are difficult to instrument statically and almost impossible to instrument dynamically. Most JVMs make assumptions regarding the structure and loading order of system classes. If these assumptions do not hold, a JVM may terminate abnormally. For example, if the instrumentation process augments one of the classes `java.lang.Object`, `java.lang.Class` or `java.lang.String` with a field, the JVM crashes. Moreover, dynamic instrumentation using custom class loaders is hindered by the fact that a subset of system classes (approximately 200 in Sun JDK 1.4.2) is already loaded by the JVM, before the class loading mechanism can be modified to enable rewriting. Finally, some system classes have *native methods*, i.e., methods whose implementation is not expressed in bytecode, but rather in a machine dependent assembler.

To enable sound instrumentation of system classes, we employ a bytecode instrumentation strategy, which we call the *Twin Class Hierarchy* approach (TCH).[18] While other bytecode instrumentation-based systems such as[5] and[6] perceive system classes as *unmodifiable code* and try to

find various workarounds to deal with the inability to instrument them, JavaSplit employs TCH to rewrite system classes for distributed execution.

The rest of this section is structured as follows. First we overview the TCH approach. (An interested reader should see[18] for more details.) Then, we describe JavaSplit specific transformations, which allow the distributed execution. Finally, we discuss the timing of instrumentation, which can be performed before the execution begins or during run time.

4.1. Twin Class Hierarchy

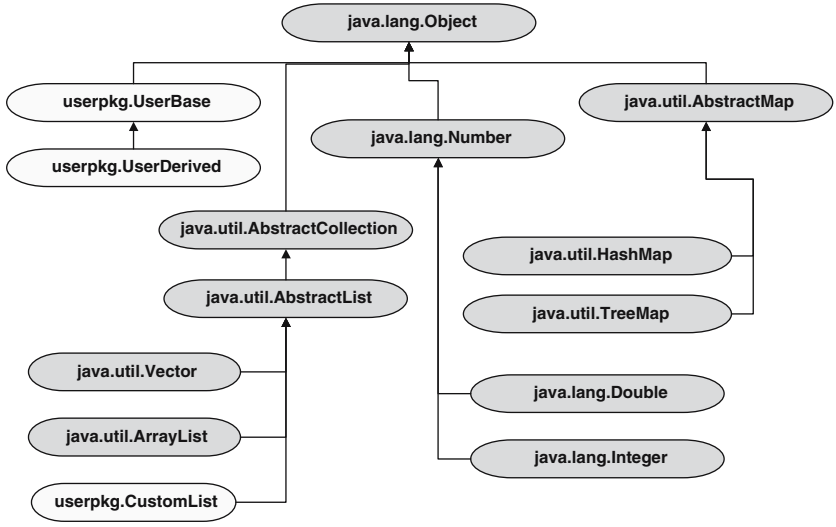
At the heart of the TCH approach lies the idea of renaming all the instrumented classes. In JavaSplit, for each original class `mypackage.MyClass` we produce a rewritten version `javasplit.mypackage.MyClass`. Thus, we create a hierarchy of classes, parallel to the original one, encapsulated in a package called `javasplit`. Figure 1 illustrates this transformation.

In a rewritten class, all referenced class names are replaced with new *javasplit* names. For example, in the bytecode, the renaming affects such instructions as `instanceof`, `invokevirtual`, `new` and `getfield`. During the execution, the runtime uses the *javasplit* classes instead of the originals. Figure 2 demonstrates this change at the source code level. In practice, however, the transformation is performed in the bytecode.

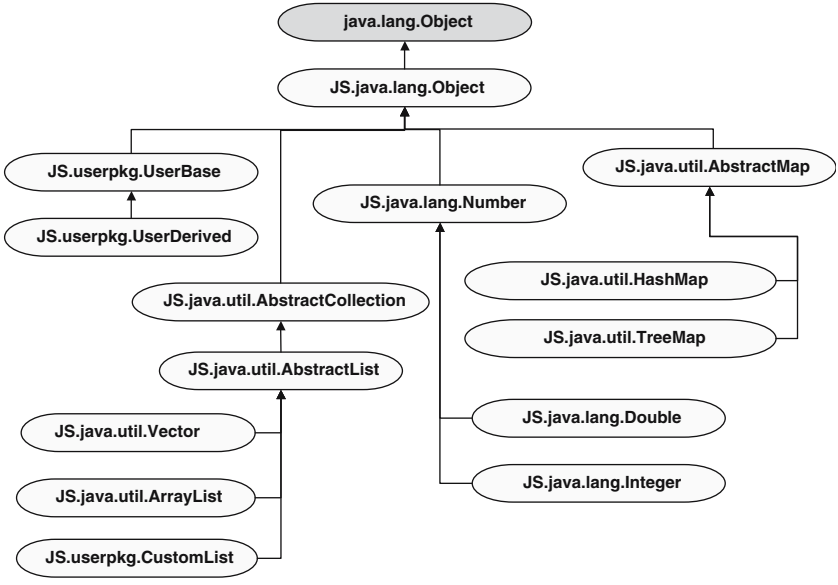
As a result of the TCH class name transformation, the rewritten system classes become user classes and thus get rid of their special status within the JVM. This eliminates most of the difficulties related to their instrumentation. However, the renaming approach may come into conflict with Java features such as inheritance, exceptions, reflection and native methods. The description of the arising problems and their solutions can be found in.[18] Here, we discuss only the most prominent problem: instrumentation of classes with native methods.

Native methods are not portable and cannot be automatically transformed. Nevertheless, in order to preserve transparency, TCH supports system classes with native methods (henceforth *native classes*). Native methods in user classes are not supported.

Native functionality is bound to class name. Therefore, after the TCH class renaming, it is unavailable in the instrumented system classes. To correct this, the instrumentation system must provide these classes with an alternative implementation that simulates their original API. The access to the original native functionality is regained by using the original version of a class in the implementation of its instrumented counterpart. In most cases, the instrumented class is implemented as a simple wrapper around an instance of its original version. The instrumented class



(a) Original class hierarchy



(b) Instrumented class hierarchy

Fig. 1. A fragment of the class hierarchy, before and after the class renaming transformation. The instrumented versions of system classes become user classes. (Java system classes are designated by dark gray).

```
class A extends somepackage.C {
    // fields
    private int myIntField;
    public B myRefField;
    public java.util.Vector myVectorField;
    // methods
    protected void doSomething(B b, int n) {
        if(b instanceof java.util.List){ ... }
        java.lang.Class vecClass = java.lang.Class.forName("java.util.Vector");
        ...
    }
    public B doSomethingElse(java.lang.String str)  {
        java.lang.System.out.println(str);
        java.io.File f = new java.io.File(str);
        ...
    }
}
```

(a) Original class

```
class JS.A extends JS.somepackage.C {
    // fields
    private int myIntField;
    public JS.B myRefField;
    public JS.java.util.Vector myVectorField;
    // methods
    protected void doSomething(JS.B b, int n) {
        if(b instanceof JS.java.util.List){ ... }
        // The string parameter of forName is not augmented with the prefix "JS."
        // Instead, the implementation of JS.java.lang.Class.forName is modified
        // to produce an instance of JS.java.lang.Class representing JS.java.util.Vector
        JS.java.lang.Class vecClass = JS.java.lang.Class.forName("java.util.Vector");
        ...
    }
    public JS.B doSomethingElse(JS.java.lang.String str)  {
        JS.java.lang.System.out.println(str);
        JS.java.io.File f = new JS.java.io.File(str);
        ...
    }
}
```

(b) Instrumented class

Fig. 2. A Java class before and after instrumentation.

delegates all API method invocations to the encapsulated instance of the original class. If necessary, the wrapper methods convert the parameters and the return type from TCH form to the original form and vice-versa. The delegation pattern, which is illustrated in Fig. 3, is highly suitable when the native methods are used to access operating system resources, such as the clock, the file system, or the network. In general, delegation is applicable to those native classes that have a distinct purpose and whose operation does not affect instances of other classes. Note that the above delegation approach works well in presence of `protected` and `private` native methods. The wrapper class defines only the API methods, which are `public`. Therefore, it needs to invoke only the public methods of the encapsulated (original) class.

```
// Returns the current time in milliseconds.
public static long currentTimeMillis(){
    return java.lang.System.currentTimeMillis();
}
```

(a) `JS.java.lang.System.currentTimeMillis`

```
// Gets the name of the local host
public JS.java.lang.String getLocalHostName() {
   // origImpl_ is a private field of type java.net.Inet4AddressImpl
   java.lang.String name = origImpl_.getLocalHostName();
   // convert the name into a JS.java.lang.String and return it
   return JS.java.lang.String.__JS__convertFromJavaString(name);
}
```

(b) `JS.java.net.Inet4AddressImpl.getLocalHostName`

```
// Calculates the square root of its parameter
public static strictfp double sqrt(double arg){
    return java.lang.StrictMath.sqrt(arg);
}
```

(c) `JS.java.lang.StrictMath.sqrt`

```
// Determines if the specified Class object represents a primitive type
public boolean isPrimitive(){
    // origImpl_ is a private field of type java.lang.Class
    return origImpl_.isPrimitive();
}
```

(d) `JS.java.lang.Class.isPrimitive`

Fig. 3. Implementation of several originally native methods. Nonstatic methods are implemented using an encapsulated instance of the original class. The encapsulated object provides the required native functionality.

Implementation of TCH versions of some native classes requires techniques that are similar to but somewhat more complex than delegation. We identify two main categories of these classes: (i) classes that are structurally incompatible with the delegation approach and (ii) classes with native methods that invoke or implement Java-specific mechanisms. (The latter can be perceived as classes semantically incompatible with delegation).

The class `java.lang.Class` (from Sun JDK 1.4.2) belongs to the first category. It is structurally incompatible with delegation because it has a *package-private* method `Class getPrimitiveClass(String)`. This method returns the class that represents the primitive type indicated by its parameter. Package-private methods can be invoked only by a class from the package of the target class. Since the instrumented classes are defined outside the original package, they cannot delegate calls to package-private methods of the original class. The solution is context-specific: upon invocation of the wrapper method, the requested class is obtained from the public field `TYPE` of the class representing the boxed primitive type. For instance, the class representing a `long` is read from the field `TYPE` of `java.lang.Long`. A more general but less secure solution is to modify the permission of the original method.

The class `java.lang.String` belongs to the second category. The semantics of its native method `String intern()` do not allow simple delegation. This method must return the canonical representation of a given string. In Java, for two equal strings *s* and *t*, `s.intern()` and `t.intern()` must always return the same string object whose value is equal to that of *s* and *t*. If implemented by simple delegation, the `intern` method of the wrapper class would wrap the canonical string returned by the original method into a new TCH string. Therefore, even two consecutive calls to the intern method on the same TCH string would not return the same object. To support the semantics of the intern method, the TCH string class should make sure that each canonical string is always wrapped into the same TCH string. This is achieved by maintaining a mapping between canonical strings and the TCH strings that are used to wrap them. This however, is a deviation from the delegation pattern.

To sum up, transformation of a native class requires intimate knowledge of its API and the side effects of its operation. We must understand whether the class is semantically (and structurally) suitable to be implemented through delegation. If not, context-specific solutions are employed to simulate the original API. Even if a native class does not fit into the delegation pattern and is, therefore, implemented in a specialized way, its instrumented version is created only once (and not for each application).

Normally, only a small portion of the system classes are native. Most of them are related to reflection, GUI, I/O, and networking. In Sun JDK 1.4.2, the packages `java` and `javax` contain 100 native classes, which constitutes around 2.5% of all the system classes in these packages. A much smaller portion is required to run most programs without a GUI. We have successfully executed various applications, including those that perform I/O and networking.

4.2. JavaSplit Specific Transformations

The bytecode instrumentation intercepts events that are important in the context of a distributed runtime. First, the bytecodes that start execution of new threads are replaced by calls to a handler that ships the thread to a node chosen by the load balancing function. Second, the lock-related operations, identified by the instructions `monitorenter` and `monitorexit` or by *synchronized methods*, are replaced by synchronization handlers. To support conditional synchronization in a distributed environment, calls to the methods `wait`, `notify` and `notifyAll` are also intercepted. Third, in order to preserve memory consistency, the rewriter inserts access checks before accesses to fields and array elements, e.g., `getfield`, `putstatic`, `iaload`, and `lastore` (see Fig. 4). If an access check fails, a newer version of the object is obtained from another node. The rewriter does not intercept calls to I/O operations from the application classes. Instead, the system classes that perform low-level I/O are modified to achieve the desired functionality.

In addition to the above modifications, the classes are augmented with utility fields and methods. The class at the top of the inheritance tree is augmented with fields indicating the state of the object during the execution, e.g., access permission, version, and whether it is locked. This approach enables quick retrieval of the state information and allows the garbage collector to discard it together with the object. The utility fields increase object size by 21 bytes. Each class is augmented with several utility methods, which are generated on the basis of the fields of the specific class. The most important utility methods are for serializing, deserializing, and comparing (*diffing*).

In the context of distributed execution, certain language features require special attention. Initialization of *static* fields must be performed only once, by the first node that uses the class. The implementation

….	….
ALOAD 1/	/l oadt he instance of classA
DUP	
GETFIELD	A::byte __javasplit__state__
IFNE	// jump to then extG ETFIELD
DUP	
INVOKESTATIC	Handler::readMiss
GETFIELD	A::myIntField
….	…..

Fig. 4. Instrumented read access of the field *myIntField* in class *A*. The instructions in bold are added by the instrumentation. If the object is valid for read, i.e., the value of the state field is not 0, only the first 3 added instructions are executed.

of the `intern` method in the instrumented string class requires managing a distributed pool of canonical (instrumented) strings. Unless all clocks are synchronized, accesses to the clock must be redirected to the same node. For any object, a call to the originally native method `java.lang.Object.hashCode()` should return the same value on any JVM implementation participating in the distributed execution. To achieve the latter, instrumented versions of classes that originally do not override `java.lang.Object.hashCode()` are provided with an alternative implementation of this method, which returns the lower 32 bits of their ID.

In Java applications there are a lot of unnecessary lock operations.[14] Often, especially in the system classes, locks protect accesses to objects that are accessed by no more than one thread during the entire execution. The overhead of the unnecessary locking may be negligible in Java. However, when rewriting bytecodes for distributed execution, one must be extra careful to avoid performance degradation, which may result from the increased cost of the transformed lock operations, which were unnecessary to begin with.

JavaSplit reduces the cost of lock operations of local objects by avoiding the invocation of lock handlers when locking a local object. Instead, a counter is associated with a local object, counting the number of times the current owner of the object has locked it since becoming its owner. (In Java, a thread can acquire the same lock several times without releasing it, and the object is considered locked until the owner performs the same number of releases.) Acquire (lock) and release (lock) operations on a local object simply increase and decrease the counter. Thus, the object is locked only when the counter is positive. If the object becomes shared when another thread wishes to acquire it, the lock counter is used to determine whether the object is locked. The cost of lock counter operations is not only low in comparison to the invocation of lock handlers for shared objects, but is also cheaper than the original Java acquire (`monitorenter`) operation (see Table I in Section 5.1).

Table I. Local Acquire Cost (*ns*)

	Original	Local obj.	Shared obj.
Sun JDK 1.4.2	90.6	19.6	281
IBM JDK 1.3.0	93.4	54.7	327

4.3. Instrumentation Timing

In Java, unless the JVM is modified, a class can be instrumented before the execution begins (*static instrumentation*), or while it is being loaded into the JVM (*dynamic instrumentation*). The advantage of static instrumentation is that the rewriting is performed offline and therefore does not affect the execution time. The main advantage of dynamic instrumentation is that it does not require *a priori* knowledge of the set of classes used by the program (*closed world assumption*). Since reflection allows the loading of classes whose identity is determined at runtime, it may be impossible to determine the transitive closure of classes used by a program. Moreover, classes created during the execution can only be instrumented dynamically.

JavaSplit can instrument classes in both modes. In the static mode, all system classes are instrumented only once per system lifetime. Their rewritten versions are put into a class file repository. Before the first execution of an application, its classes are transformed and are also placed in the repository. A node that wishes to load a class `javasplit.SomeClass` fetches it from the repository and loads it as is. In the dynamic mode, the original versions of user-defined and system classes are stored in a class file repository. A node that wishes to load a class `javasplit.SomeClass` fetches the class `SomeClass` from the repository, transforms it, and loads the instrumented version `javasplit.SomeClass` into the JVM.

5. PERFORMANCE EVALUATION

We have evaluated the efficiency of JavaSplit in several different settings. Our tests were performed on several combinations of operating systems, JVM implementations, and IP-compliant communication hardware. (This supports the claim of the Java-like portability of our system.)

The structure of the current section is as follows. First we present the instrumentation overheads and communication latency and throughput. Then we describe the performance of our benchmark applications on a collection of Intel Xeon dual processor (2×1.7 MGHz) machines running on Windows XP and interconnected by a 100 Mbps Ethernet. Note that this configuration is not very advantageous in the context of distributed execution, since the ratio of bandwidth to CPU power is considerably smaller than in performance evaluations of similar systems.[1,4,19] The results obtained by this hardware setting when running on Red Hat Linux are not presented, due to their similarity to the results obtained under Windows XP. Finally, we compare the performance of a less scalable application, on a collection of Intel 2×2.4 MGHz dual processor

machines running on Red Hat Linux in two communication settings: 100 Mbps Ethernet and 10 Gbps Inifiniband in IP-over-Infiniband (IPoIB) mode. (The effective bandwidth provided in IPoIB is only 3.5 Gbps.) Infiniband significantly improves performance, obtaining a scalable speedup.

In its current state, the system does not perform any optimizations on the bytecode rewriting process. We believe that existing optimizations[1,20] based on flow analysis of the original bytecodes, e.g., access check elimination and batching, can reduce most of the instrumentation overhead. These optimizations have not been applied to JavaSplit yet, because, in contrast to scalability, they are not among our main research goals. Note, however, that there are certain applications for which the instrumentation overhead is negligible.

5.1. Instrumentation Overhead

The most significant changes, performance-wise, introduced by the JavaSplit bytecode rewriter are: (i) the addition of access checks before accesses to heap data (i.e., accesses to object fields and array elements), and (ii) replacing lock operations (e.g., `monitorenter` and `monitorexit`) with distributed synchronization code.

Table II shows the cost of heap data accesses in the rewritten code in comparison with their cost in the original Java program. These figures were obtained on a 2×1.7 GHz station running on Windows.

In Sun JDK 1.4.2 for Windows, heap data accesses introduce a slowdown of between 2.2 and 5.6, whereas in IBM JDK 1.3.0 for Windows they seem to introduce a slowdown of between 12 and 55. This great difference is due to the fact that IBM JDK optimized away the repeated

Table II. Heap Access Latency (*ns*). IBM JDK Optimized Away the Data Accesses in the Employed Microbenchmarks

	Original		Rewritten		Slowdown	
	Sun	IBM	Sun	IBM	Sun	IBM
Field read	0.84	0.07	1.82	1.63	2.2	24.9
Field write	0.97	0.06	2.48	0.74	2.6	12.2
Static read	0.84	0.06	1.84	1.61	2.2	26.9
Static write	0.97	0.06	2.97	0.73	3.1	11.9
Array read	0.98	0.09	5.45	4.99	5.6	55.1
Array write	1.23	0.19	5.05	4.98	4.1	25.7

accesses to the same data in the original code. In the instrumented code, however, it appears that the access checks stand in the way of these optimizations. In real applications the data access patterns are not as trivial as in the employed microbenchmarks. Therefore, there are very few opportunities for such JVM optimizations. In fact, none of the tested applications has ever exhibited a slowdown greater than 2. In any case, existing techniques[1,20] can be employed to eliminate a large portion of access checks and thus reduce the overhead of the heap data accesses.

Table I shows the cost of a local acquire operation, i.e., an acquire which does not result in communication. Although there is considerable overhead in acquiring a shared object, acquiring local objects costs less than in original Java. This is due to the lock optimization described in Section 4.2.

5.2. Communication Latency and Throughput

The experiments presented in the following sections utilize two types of communication hardware: 100 Mbit Ethernet and 10 Gbit Infiniband.

Table III presents the latency of the utilized communication layer for different message sizes. During the execution, the nodes exchange messages ranging from several bytes to dozens of megabytes (depending on the application). For messages shorter than 65 kilobytes we use UDP messages, whereas longer messages are sent through the TCP protocol. In general, the latency of the utilized interconnect is important in the context of short messages, while the throughput is significant in the context of large ones.

The message latency of Sun JDK 1.4.2 is the highest. The latency of Infiniband is the lowest, one order of magnitude lower than in Sun JDK

Table III. Message Latency (*ms*)

Msg. size	Windows		Linux	
	Ethernet		Ethernet	Infinband
	Sun 1.4.2	IBM 1.3.0	Sun 1.4.1	Sun 1.4.1
1	0.66	0.09	0.19	0.04
65	0.65	0.09	0.17	0.04
650	0.84	0.41	0.30	0.04
6500	1.76	0.65	0.83	0.09
65000	7.08	3.43	3.97	0.28

Table IV. Throughput (Mbit/s)

	Windows		Linux	
	Ethernet		Ethernet	Infinband
	Sun 1.4.2	IBM 1.3.0	Sun 1.4.1	Sun 1.4.1
Native	100	100	100	3500
Java	90.44	90.40	89.76	2860.95
Overhead %	9.56	9.60	10.24	18.26

1.4.2. The latency of IBM JDK 1.3.0 for Windows is similar to Sun JDK 1.4.1 for Linux.

Table IV describes the throughput of the interconnects and compares it to the bandwidth available from the native socket interface. The measurements show that Java sockets utilize around 90% of Ethernet bandwidth and 80% of Infiniband bandwidth. The overhead is due to the indirect access to the network interface, i.e., through the JVM. Note that the bandwidth available in the IPoIB mode to the native sockets is only 3.5 Gbps, despite the fact we use a 10 Gbps Infiniband.

5.3. Benchmark Applications

In our performance evaluation we use eight applications, the first three of which are from the *Java Grande Forum Benchmark Suite*.[21]

(1) *Series.* Computes the first N Fourier coefficients of the function $f(x) = (x+1)x$. The calculation is distributed between threads in a block manner. We run Series for $N = 1,000,000$.
(2) *SparseMatmult.* This benchamark multiplies unstructured sparse $N \times N$ matrices stored in compressed-row format with a prescribed sparsity structure. It exercises indirect addressing and non-regular memory references. We use $N = 500,000$.
(3) *RayTracer.* Renders a scene containing 64 spheres at resolution of $N \times N$ pixels. The worker threads of this application render different rows of the scene. We run the Ray Tracer for $N = 2000$.
(4) *TSP.* The TSP application searches for the shortest path passing through all N vertices of a given graph. The threads eliminate some permutations using the length of the minimal path known so far. A thread discovering a new minimal path propagates its length to the rest of the threads. During the execution

the threads also cooperate to ensure that no permutation is processed by more than one thread by managing a global queue of jobs. We run TSP for $N = 21$.

(5) *FileCrypt.* Encrypts a (very large) input file using Triple DES. The file is divided into blocks. In each iteration a thread gets a block, encrypts it, and writes the result into a separate file. The output of the program is an enumerated set of encrypted blocks. In our tests we encrypt a 1 GB file into 5 MB blocks. This benchmark demonstrates the I/O capabilities of the system.

(6) *PrimeFinder.* Searches for prime numbers in the range $[2, N]$. The detected primes are written to the standard output. For better load balancing, the range is dynamically distributed among threads which cooperate to avoid checking the same number more than once. In our experiments $N = 3,500,000$.

(7) *MaxClique.* Finds a the maximal clique in an arbitrary graph. We use a random graph with 52 vertices and edge density of 0.5. The execution of a thread is composed of iterations. In each iteration a thread checks whether a clique of a given size containing a given vertex exists. Whenever a thread finds a clique larger than known so far it propagates its size to the rest of the threads.

(8) *KeySearch.* Implements a known plain-text attack on DES. The inputs of the application are an original message and the result of its encryption. The threads cooperate to find the encryption key in the range 2^{24} of keys. Coordination among threads is needed to allow load balancing and in order to ensure the threads terminate as soon as the key is found.

5.4. Windows/Ethernet Measurements

The intranets of most large organizations use Windows-based workstations interconnected by a 100 Mbps Ethernet. Therefore we chose this as our default setting. We employed the latest versions of the two most popular JVM implementation for Windows: Sun JDK 1.4.2 and IBM JDK 1.3.0. We used 2×1.7 GHz dual processor workstations.

In all measurements two application threads were executed on each of the dual-processor nodes. We performed separate measurements for different JVMs. The executed programs were compiled using Sun JDK 1.4.2. To calculate the speedup, we divided the execution time of the original (unmodified) Java application with two threads on a single dual-processor machine by the execution time in JavaSplit. In the following graphs, the X-axis indicates the number of utilized CPUs rather than the number of stations. The speedup is calculated separately for each JVM, each

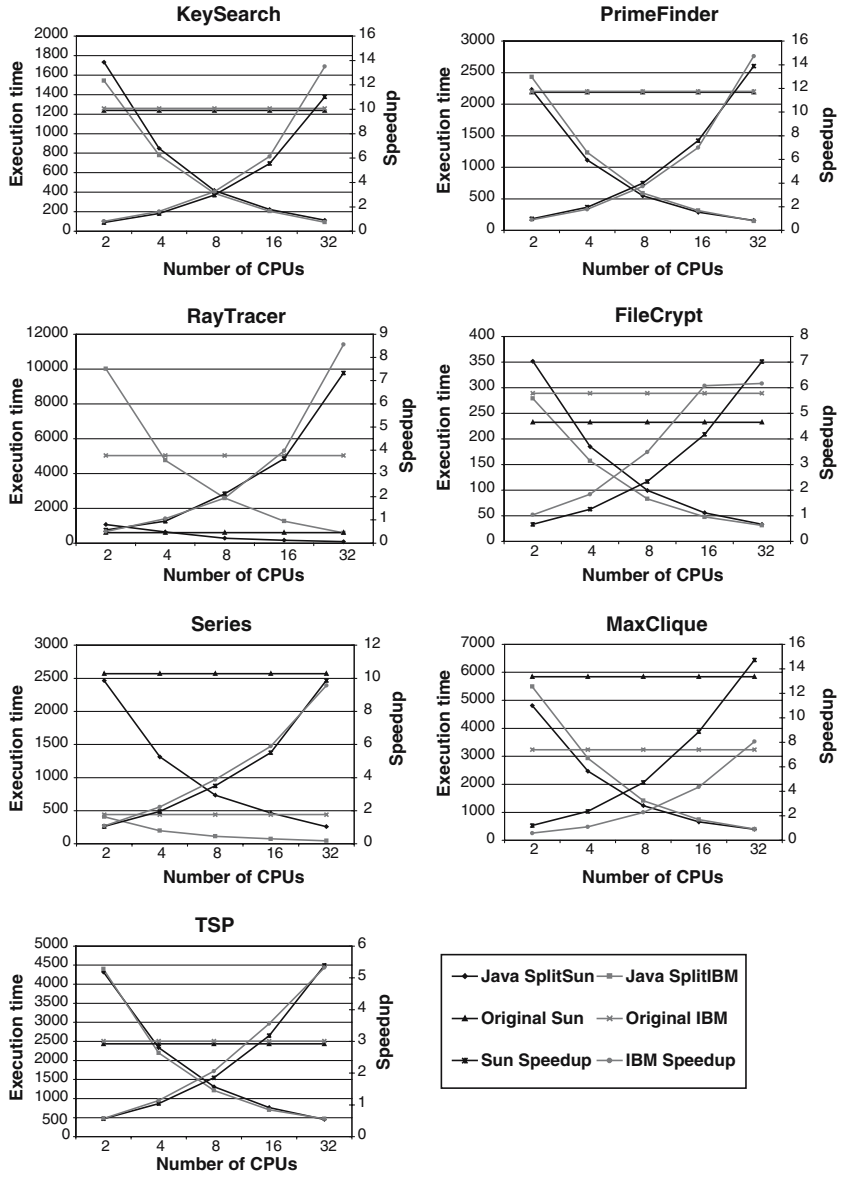
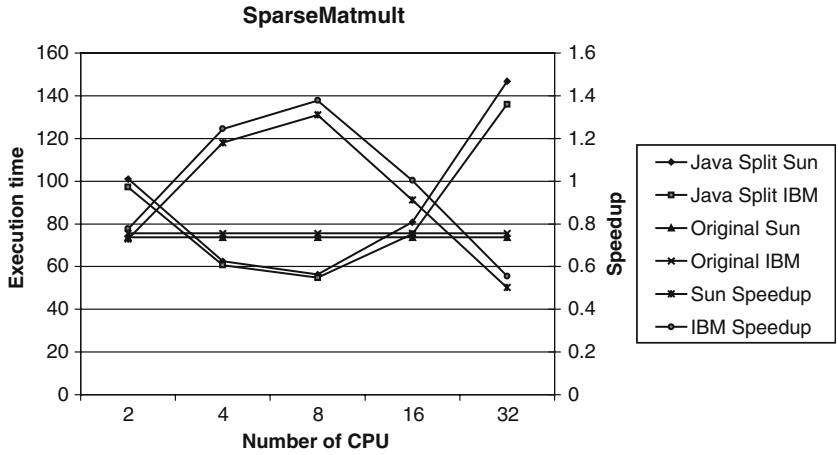
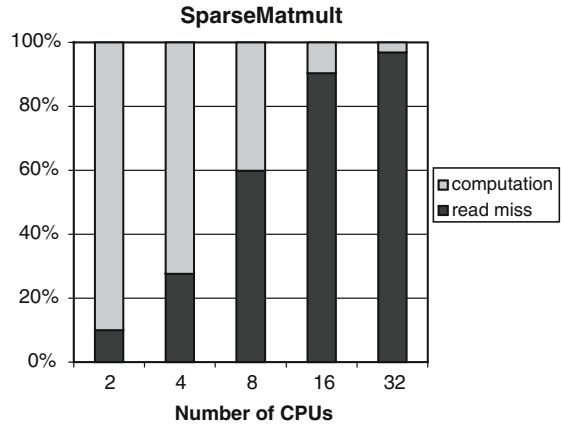


Fig. 5. Execution times (s) and speedups.



(a) Execution time (sec) and speedup



(b) Breakdown

Fig. 6. SparseMatmult results in the Windows/Ethernet setting.

calculation based on the times produced by the same JVM. The results are presented in Figs 5 and 6. Tables V and VI characterize network traffic and memory consumption of the employed benchmarks, respectively.

In KeySearch and PrimeFinder the two JVMs exhibit similar results and produce quite scalable speedups. In RayTracer and Series we observed that the execution times of the original and the rewritten programs on one JVM differ greatly from their counterparts on the second JVM, but are proportional to them respectively. Thus, the obtained speedups are almost identical.

Table V. Network Traffic (32 processes)

Application	Bytes	Messages
KeySearch	690387	5159
PrimeFinder	2929838	20961
RayTrace	2520866	20485
FileCrypt	603993	4540
Series	442046246	621
MaxClique	3256265	30630
TSP	16854791	73708
SparseMatmult	1542528177	894

Table VI. Memory Consumption (MB)

Application	Uninstrumented app. with 2 threads		Single JavaSplit app. thread	
	Used	Heap	Used	Heap
KeySearch	1.4	2.0	8.1	9.5
PrimeFinder	1.2	2.0	4.2	5.2
RayTrace	49.0	58.2	42.4	66.5
FileCrypt	10.5	11.8	12.3	24.1
Series	16.4	22.0	39.1	48.0
MaxClique	1.2	2.1	6.1	9.3
TSP	0.8	2.0	4.1	5.0
SparseMatmult	92.0	101.1	48.0	89.1

In FileCrypt, the reduced speedup for 32 processors by both JVMs is mostly due to the fact that the execution time has a lower bound around the 30 s mark. At this point the application is limited by the hard disk access speed. Note that despite the differences on a smaller number of processors, both JVMs achieve exactly the same result on 32 CPUs. Without this bound, a higher speedup would be achieved.

The relatively low speedup obtained by TSP is caused by the instrumentation overhead and lock contention between threads residing on different nodes. The instrumentation slowdown of TSP is approximately 2. (Note that the TSP speedup for 2 CPUs is around 0.5.)

Figure 6 shows that SparseMatmult does not scale well. Its performance degrades with the number of processors due to its unscalable communication pattern. During the execution, the worker threads fetch big pieces of matrices from the *main* thread that creates the matrices. Since

the main thread sends these matrix parts in a sequential manner, a greater number of threads results in a greater average fetch request delay, as shown in Fig. 6(b).

5.5. Linux/Infiniband Measurements

Our infiniband network was limited to 5 dual-processor 2×2.4 GHz stations running on Linux. In the experiments described below we utilized these stations in two different settings: (i) 100 Mbps Ethernet connection, and (ii) 10 Gbps Infiniband connection (in IPoIB mode). All measurements were performed with Sun JDK 1.4.1. In order to give the reader a more comprehensive view despite the few available workstations, we present results for each even number of processors in the range [2,10].

Figure 7 presents the performance of SparesMatmult on Ethernet and on Infiniband. With Ethernet the performance does not improve after a certain number of processors. With Infiniband, however, we obtain a speedup proportional to the number of CPUs. Figure 8 presents the breakdown of the execution time for both settings. The wider bandwidth of Infiniband significantly reduces the read miss overhead.

6. RELATED WORK

There have been several works devising a distributed runtime for Java.[1–9] The existing systems can be classified into three main categories: (i) cluster-aware VMs, (ii) compiler-based DSM systems, and (iii) systems

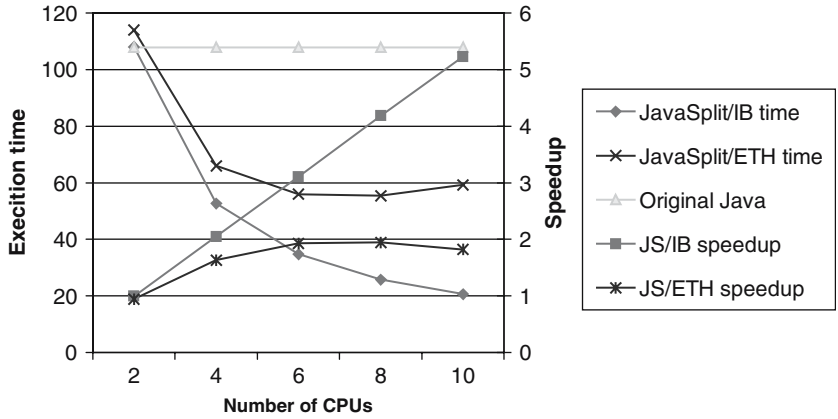
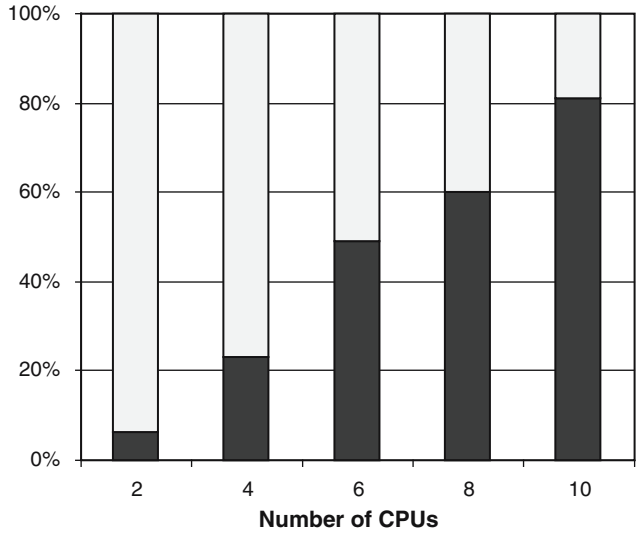
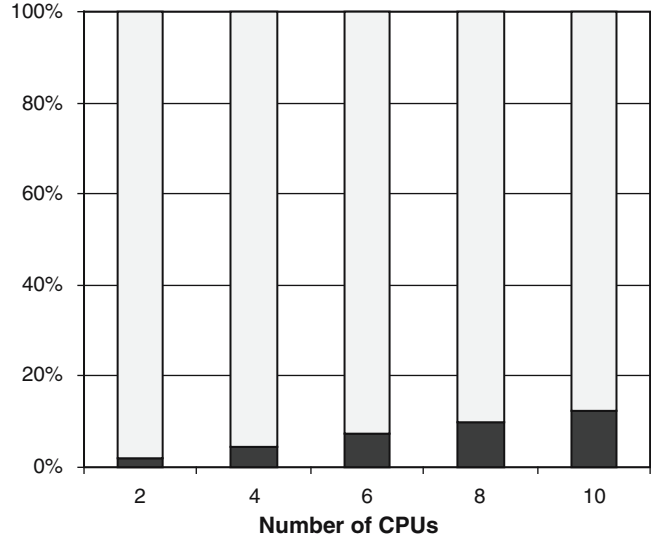


Fig. 7. Comparison of Infiniband and Ethernet runs on Linux. Infiniband exhibits linear speedup.



(a) Ethernet



(b) In niband

Fig. 8. Breakdown of the Linux runs. The black and the gray indicate the time spent in data misses and in local computation respectively.

using standard JVMs. JavaSplit belongs to the last category. JavaSplit's distinguishing feature in comparison to the previous work is the combination of transparency and portability.

6.1. Cluster-Aware Virtual Machines

Java/DSM,[8] Cluster VM for Java (formerly cJVM),[2] and JESSICA2[9] implement distributed JVMs. These systems require that each node contain a custom JVM.

In Java/DSM the local VM is similar to a standard JVM, except that all objects are allocated on an existing C-based software DSM, called TreadMarks.[22] Like our work, TreadMarks implements LRC. The single system image provided by Java/DSM is incomplete: a thread's location is not transparent to the programmer, and the threads cannot migrate between machines. In contrast, Cluster VM for Java and JESSICA2 provide a complete single system image of a traditional JVM.

Instead of using a DSM, Cluster VM for Java uses a proxy design pattern with various caching and object migration optimizations. JESSICA2 uses a home-based global object space (GOS) to implement a distributed Java heap. JESSICA2 has many desirable capabilities, e.g., support for load balancing via thread migration, an adaptive migrating-home protocol for the GOS, and a dedicated JIT compiler. The latter feature distinguishes JESSICA2 from the majority of similar systems. For instance, Cluster VM for Java and Java/DSM are unable to use a standard JIT and do not implement a dedicated one. By contrast, JavaSplit is able to utilize any standard JIT supplied with the local JVM.

These cluster-aware VMs are potentially more efficient than systems using standard JVMs: they are able to access machine resources, e.g., memory and network interface directly (rather than through the JVM). On the downside, because a node's local JVMs is modified in these systems, none of them has true cross-platform portability. Without the need to preserve portability, the cluster-aware VMs can use efficient nonstandard networking hardware.

6.2. Compiler-based DSM Systems

Compiler-based DSM systems compile Java programs into native machine code, while adding DSM capabilities. There are two compiler-based systems known to us: Hyperion[1] and Jackal.[7] Both systems support standard Java and do not require any changes in the programming paradigm.

Hyperion translates Java bytecodes to *C* source code and then compiles the *C* source using a native *C* compiler. The DSM handlers are inserted during the translation to *C*. The Java-bytecode-to-*C* translator performs various optimizations in order to improve the performance of the DSM. For example, if a shared object is referenced in each iteration of a loop (that does not contain synchronization), the code for obtaining a locally cached copy of the object is lifted out of the loop. Hyperion employs existing DSM libraries to implement its DSM protocol and is able to use various low-level communication layers.

Jackal combines an extended Java compiler and runtime support to implement a fine-grain DSM. The compiler translates Java sources into *Intel x86* code rather than Java bytecode. The Jackal compiler stores Java objects in shared regions and augments the program it compiles with access checks that drive the memory consistency protocol. Like Hyperion, it performs various optimizations, striving to achieve a more efficient distributed execution.[20] Jackal incorporates a distributed garbage collector and provides thread and object location transparency.

In the compiler-based systems, the use of a dedicated compiler allows various compiler optimizations to be performed. These have the potential to significantly improve performance. In addition, since the application is compiled to machine code, the speed of a local execution is increased without requiring a JIT. On the downside, the use of the native machine code sacrifices portability. Like the cluster-aware VMs, the compiler-based systems have direct access to the node resources.

6.3. Systems Using Standard JVMs

The existing systems[3–6,23–25] that use a collection of standard JVMs are not transparent. They either introduce unorthodox programming constructs and style or require user intervention to enable distributed execution of an existing program. By contrast, JavaSplit is completely transparent, both to the programmer and to the person submitting the program for execution.

Among the systems that deviate from the Java programming paradigm, JavaParty[3] and JSDM[4] are close enough to pure Java in order to be considered in the current context.

JavaParty supports parallel programming by extending Java with a preprocessor and a runtime. JavaParty modifies the Java programming paradigm by introducing a new reserved word, *remote*, to indicate classes that should be distributed across machines. The source code is transformed into regular Java code plus RMI hooks which are passed to the RMI compiler. The single system image is further flawed by the fact that the

programmer must also distinguish between remote and local method invocations, due to the differing argument passing conventions.

In JSDM, access checks, in the form of method invocation to memory consistency operations, are inserted manually by the user (or possibly a higher-level program translator) for field read/write accesses. JSDM requires that an input program be an SMPD-style multithreaded program. Moreover, the programmer must use special classes provided by JSDM when writing the program and mark the shared objects.

Like JavaSplit, Addistant[5] and jOrchestra[6] instrument Java bytecode for distributed execution (but not necessarily to achieve a better performance). Both systems require nontrivial user intervention to transform the classes used by the application. This intervention demands knowledge of the application structure, further compromising the transparency of these systems. Addistant and jOrchestra employ the master-proxy pattern to access remote objects. In contrast, JavaSplit uses an object-based DSM. Unlike JavaSplit, Addistant and jOrchestra are unable to instrument system classes and therefore treat them as *unmodifiable code*. This results in several limitations, mostly related to the placement of data. In contrast to jOrchestra and Addistant, JavaSplit supports arbitrary distribution of objects and does not require user intervention in the instrumentation process.

Addistant provides only class-based distribution: all the instances of a class must be allocated on the same host. The user has to explicitly specify whether instances of an unmodifiable class are created only by modifiable code, whether an unmodifiable class is accessed by modifiable code, and whether instances of a class can be safely passed by-copy. This information is application-specific and getting it wrong results in a partitioning that violates the original application semantics.

In jOrchestra, user involvement in the instrumentation process is less significant than in Addistant. The former provides a *profiler* tool that determines class interdependencies and a *classifier* tool that ensures the correctness of the partition chosen by the user. jOrchestra cannot support remote accesses from system classes to any remote object. To solve this problem, jOrchestra statically partitions the objects among the nodes, placing all (user-defined and system) objects that can be referenced from a certain system class on the same node with that class. Due to the strong dependencies of classes within Java packages, this usually results in partitions that coincide with package boundaries. The transparency of jOrchestra is further flawed by the incomplete support for Java synchronization: *synchronized* blocks and `wait/notify` calls do not affect remote nodes.

There are several advantages to using standard JVMs. First, the system can use heterogenous collections of nodes. Second, each node can

locally optimize its performance, via a JIT, for example. Third, local garbage collection can be utilized to collect local objects that are not referenced from other nodes. The main drawback of these systems is their indirect access to the node resources. Since JavaSplit uses only standard JVMs, it possesses all the advantages mentioned above.

7. CONCLUSION

We view JavaSplit as a first step towards providing a convenient computing infrastructure for large-scale and possibly nondedicated environments. The underlying question is whether (mostly idle) Internet and enterprise interconnects are fast enough and broad enough to efficiently support high-level programming paradigms that provide shared memory abstraction. Java, as a popular multithreaded programming language, is best suited for this experiment. The presented performance evaluation of JavaSplit shows that scalable speedups can be produced using commodity hardware.

REFERENCES

1. G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst, The Hyperion System: Compiling Multithreaded Java Bytecode for Distributed Execution, *Parallel Computing*, **27**(10):1279–1297 (2001). [Online]. Available: citeseer.nj.nec.com/antoniu01hyperion.html.
2. Y. Aridor, M. Factor, and A. Teperman, cJVM: A Single System Image of a JVM on a Cluster, *International Conference on Parallel Processing*, pp. 4–11 (1999). [Online]. Available: citeseer.nj.nec.com/aridor99cjvm.html.
3. M. Philippsen and M. Zenger, JavaParty—Transparent Remote Objects in Java, *Concurrency: Practice and Experience*, **9**(11):1225–1242 (1997). [Online]. Available: citeseer.nj.nec.com/philippsen97javaparty.html.
4. Y. Sohda, H. Nakada, and S. Matsuoka, Implementation of a Portable Software DSM in Java, *Java Grande*, Stanford University, CA, USA (2001). [Online]. Available: citeseer.nj.nec.com/sohda01implementation.html.
5. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano, A Bytecode Translator for Distributed Execution of "Legacy" Java Software, *Lecture Notes in Computer Science*, vol. 2072 (2001). [Online]. Available: citeseer.nj.nec.com/tatsubori01bytecode.html.
6. E. Tilevich and Y. Smaragdakis, J-Orchestra: Automatic Java Application Partitioning, *European Conference on Object-Oriented Programming* (*ECOOP*), Malaga, Spain (June 2002).
7. R. Veldema, R. A. F. Bhoedjang, and H. E. Bal, Distributed Shared Memory Management for Java, *Proceedings of the Sixth Annual Conference of the Advanced School for Computing and Imaging* (*ASCI 2000*), pp. 256–264 (2000).
8. W. Yu and A. L. Cox, Java/DSM: A Platform for Heterogeneous Computing, *Concurrency – Practice and Experience*, **9**(11):1213–1224 (1997). [Online]. Available: citeseer.nj.nec.com/yu97javadsm.html.

9. W. Zhu, C.-L. Wang, and F. C. M. Lau, JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support, *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA (September 2002). [Online]. Available: citeseer.nj.nec.com/zhu02jessica.html.
10. JSR 133, Java Memory Model and Thread Specification Revision, http://jcp.org/jsr/detail/133.jsp.
11. P. Keleher, A. L. Cox, and W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Annual Int'l Symp. on Computer Architecture* (*ISCA'92*), pp. 13–21 (1992). [Online]. Available: citeseer.nj.nec.com/keleher92lazy.html.
12. Y. Zhou, L. Iftode, and K. Li, Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems, *Proceedings of the 2nd Sympsium on Operating Systems Design and Implementation* (*OSDI'96*), Seattle, Washington, USA, pp. 75–88. (1996). [Online]. Available: citeseer.nj.nec.com/zhou96performance.html.
13. R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, Home-based SVM Protocols for SMP Clusters: Design, Simulations, Implementation and Performance, *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, Las Vegas, USA (1998).
14. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, Escape Analysis for Java, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), Denver, USA, pp. 1–19. (1999). [Online]. Available: citeseer.nj.nec.com/choi99escape.html.
15. L. Iftode, Home-based Shared Virtual Memory, Ph.D. dissertation, Princeton University, New Jersey, New York, USA (1998). [Online]. Available: citeseer.nj.nec.com/article/iftode98homebased.html.
16. D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano, Thin Locks: Featherweight Synchronization for Java, *SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, pp. 258–268. (1998). [Online]. Available: citeseer.nj.nec.com/bacon98thin.html.
17. J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, 2nd edn*, Addison-Wesley, Boston, Mass., (2000). [Online]. Available: citeseer.nj.nec.com/gosling00java.html
18. M. Factor, A. Schuster, and K. Shagin, Instrumentation of Standard Libraries in Object-Oriented Languages: The Twin Class Hierarchy Approach, *Object-Oriented Programming, System, Languages and Applications*, Vancouver, Canada (October 2004).
19. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal, Runtime Optimizations for a Java DSM Implementation, *Java Grande*, Stanford University, CA, USA, pp. 153–162 (2001). [Online]. Available: citeseer.nj.nec.com/veldema01runtime.html.
20. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal, Source-level Global Optimizations for Fine-grain Distributed Shared Memory Systems, *ACM SIGPLAN Notices*, **36**(7):83–92 (2001). [Online]. Available: citeseer.nj.nec.com/veldema01sourcelevel.html.
21. L. A. Smith and J. M. Bull, A Multithreaded Java Grande Benchmark Suite, *Proceedings of the Third Workshop on Java for High Performance Computing*, Amsterdam, Holland (2001).
22. P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, USA, pp. 115–131 (1994). [Online]. Available: citeseer.nj.nec.com/keleher94treadmark.html.

23. D. Caromel, W. Klauser, and J. Vayssière, Towards Seamless Computing and Metacomputing in Java, *Concurrency: Practice and Experience*, **10**(11–13):1043–1061 (1998). [Online]. Available: citeseer.nj.nec.com/article/caromel98towards.html.
24. M. Herlihy, The Aleph Toolkit: Support for Scalable Distributed Shared Objects, *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Orlando, USA (January 1999).
25. N. Camiel, S. London, N. Nisan, and O. Regev, The POPCORN Project: Distributed Computation Over the Internet in Java, *6th International World Wide Web Conference*, Santa Clara, CA, USA (1997).