JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations

To Appear in Cluster 2003

Michael Factor
IBM Research Lab in Haifa
Haifa University Campus
Haifa 31905, Israel
factor@il.ibm.com

Assaf Schuster
Computer Science Department
Israel Institute of Technology
Technion City
Haifa 32000, Israel
assaf@cs.technion.ac.il

Konstantin Shagin
Computer Science Department
Israel Institute of Technology
Technion City
Haifa 32000, Israel
konst@cs.technion.ac.il

Hong Kong, December 1-4

ABSTRACT

This paper describes the design and presents the preliminary performance evaluation of *JavaSplit*, a portable runtime for distributed execution of multithreaded Java programs. JavaSplit *transparently* distributes threads and objects of an application among the participating nodes. Thus, it gains augmented computational power and increased memory capacity without modifying the Java multithreaded programming conventions, allowing the programmer to be unaware of the distributed nature of the underlying environment.

JavaSplit works by rewriting the bytecodes of a given parallel application, transforming it into a distributed application that incorporates all the runtime logic. Each runtime node carries out its part of the resulting distributed computation using nothing but its local standard (unmodified) Java Virtual Machine (JVM). This is unlike previous Java-based distributed runtime systems, which use a specialized (modified) JVM or utilize unconventional programming constructs. Since the proposed runtime is *orthogonal* to the implementation of a local JVM, it achieves portability across any existing platform and allows each node to locally optimize the performance of its JVM, *e.g.,* via a just-in-time compiler (JIT).

The JavaSplit runtime is designed to be highly *scalable*, never requiring global cooperation of nodes and using an efficient and scalable fine-grain distributed shared memory (DSM) protocol.

Keywords: Java, Network-based Distributed Computing, Single-System Image, Bytecode Instrumentation, Portability.

1. INTRODUCTION

Interconnected collections of commodity workstations can yield a cost-effective substitute to dedicated parallel computers for executing 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 environment provides a shared memory abstraction.

Since the introduction of Java, the Java programming language has become widely accepted and popular. Due to its built-in support for multithreading and synchronization, Java has no rival in convenience of constructing parallel applications with shared memory abstraction (henceforth multithreaded applications). However, these parallel Java programs execute on a single Java Virtual Machine (JVM), which traditionally assumes an underlying shared memory machine.

There have been several works devising a distributed runtime for Java [2][20][1][22] [17][11][16]. Some works [2][20][22] install a customized cluster-aware JVM on each participating node. Others, [1][17], compile the source code of a given application into native machine code while adding DSM capabilities. Both these approaches sacrifice one of the most important features of Java: the cross-platform portability. Another drawback of these systems is the difficulty of integrating the existing JVM facilities, e.g., local garbage collection and just-in-time (JIT) compiler. Therefore, most of these works either invest considerable efforts in implementing the facilities mentioned above or do not implement them at all. The systems, which are able to execute a Java program on top of collection of standard JVMs, e.g., [11][16], introduce unorthodox programming constructs and often fail to provide a complete single system image.

In contrast to previous work, JavaSplit achieves portability while preserving Java's original programming paradigm. It is able to execute a pre-existing, multi-threaded application on any given heterogeneous set of machines. To accomplish this, a multithreaded Java program is tied to the runtime logic (also written in Java) by automatic instrumentation of the application's bytecodes. The instrumentation intercepts events that are important in the context of a distributed runtime, namely: (i) object field and array element accesses, (ii) thread creation, (iii) synchronization operations, and (iv) calls to methods implementing low-level I/O operations. When combined with the code of the runtime, the instrumented program becomes a distributed application, ready to be executed on multiple nodes (Figure 1). Note that, unlike [1][17][11], this approach does not require the source code of the application, which can be unavailable or confidential.

In JavaSplit, each node executes its part of the rewritten application on its local standard Java Virtual Machine (JVM). This approach is completely orthogonal to the implementation of a JVM. Any node with a conventional JVM installed can be utilized by our runtime without any modifications to its software base. Another significant benefit from independence of JVM implementation is the ability of each node to locally optimize the performance of its JVM, e.g., via a JIT.

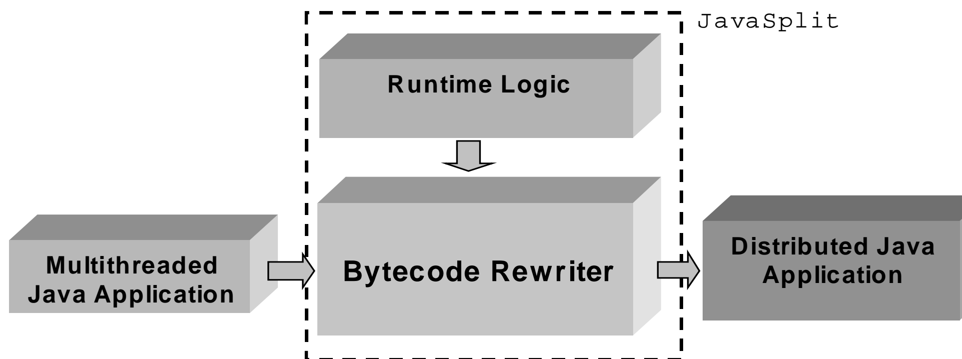

Figure 1. Bytecode transformation of the input application

In order to provide an illusion of a global address space, JavaSplit incorporates a fine-grain distributed shared memory (DSM). The semantics of our DSM are consistent with the proposed revisions to the Java Memory Model (JMM) [24]. To implement it, we have devised a novel scalable and multithreaded DSM protocol with efficient support for Java-specific synchronization primitives, *e.g., wait* and *notify*.

Due to its enhanced portability and use of a well-known parallel programming paradigm, JavaSplit enables rapid development of distributed systems, *e.g.*, distributed servers, composed entirely of commodity hardware. Scalable design allows JavaSplit to efficiently utilize a large heterogeneous collection of machines, making it suitable for wide-area *cycle stealing*. Independent of the usage, 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.

The main contributions of this work are as follows. First, we develop techniques that allow distributed execution of a standard, possibly pre-existing Java application through instrumentation of its bytecodes. Second, we extend a line of works in the area distributed shared memory to support efficient multithreading and improve scalability. Finally, we create a convenient and portable infrastructure for distributed computing.

The structure of the rest of this paper is as follows. Section 2 gives a brief overview of the runtime. Section 3 presents our DSM protocol. In Section 4 we describe the employed bytecode instrumentation techniques. Related works are discussed in Section 5. Section 6 shows the results of preliminary performance evaluation. In Section 7 we conclude.

2. SYSTEM OVERVIEW

The JavaSplit runtime administers a pool of worker nodes. Each worker node can execute an arbitrary number of application threads. Hence, the number of available workers does not restrict the number of threads. During execution, new workers can join the system and execute newly created threads.

The bytecodes of an application submitted for distributed execution are transformed and combined with the runtime modules. These bytecode modifications are performed using the Bytecode Engineering Library (BCEL) [5]. The resulting rewritten classes are sent to one of the worker nodes that starts executing the application's *main* method. Each newly created application thread is placed for execution on one of the worker nodes, according to a plug-in load balancing function[1].

JavaSplit supports the shared memory abstraction by using an integrated object-based distributed shared memory system, implementing *Lazy Release Consistency* (LRC) [10]. In our DSM, the node on which the object is created manages the object's *master copy*, the most up-to-date version of the object. The threads that need to use the object create local (cached) copies, based on the master copy. At certain synchronization points the modifications are flushed from the cached copies to the master copy.

In order to reduce overhead, the DSM system dynamically classifies the objects into *local* and *shared*, managing only the latter. Shared objects can potentially be accessed by several threads, whereas *local* objects cannot. A newly created object is always *local*. A local object becomes *shared* if the system detects that more than one thread can access it. In this case, the object receives a globally unique id (64-bit long) and is registered with

[1] Currently, we use the simplest load-balancing function, placing a new thread on the least loaded worker.

the DSM. Dynamic detection of shared objects contributes to the system's performance, since the general 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 among threads, the gain can be significant.

Unlike systems that utilize specialized networking hardware, *e.g.* [1][17], JavaSplit employs standard IP-based communication. This enables us to perform computation on any given set of machines connected by an IP network, *e.g.,* the Internet or intranet of a large organization. To preserve portability, a node accesses the IP network through the standard Java socket interface.

The DSM protocol requires that the state of an arbitrary object can be serialized and deserialized. We do not use Java's built-in serialization mechanism, since it is too slow for our purposes, including many unneeded features, *e.g.*, serialization of referenced objects (deep copy), ability to serialize only a part of object's fields, *etc*. Instead, we augment each rewritten class with class-specific methods serialization and deserialization methods (see Section 4).

3. DISTRIBUTED SHARED MEMORY

To support the shared memory abstraction, JavaSplit incorporates an object-based distributed shared memory (DSM). 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 constrains on the implementation of shared memory in the context of a single JVM. Currently, the JMM is being redefined [24]. The new JMM is expected to be based on the principles of *Lazy Release Consistency* (LRC) [10]. 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 called *Multithreaded Scalable Home-based Lazy Release Consistency* (MTS-HLRC). It is inspired by the state-of-the-art *Home-based Lazy Release Consistency* (HLRC) [23]. MTS-HLRC refines the scalability of HLRC and augments it with efficient support for multithreading.

Similarly to HLRC, our protocol is *home-based*: each object has a node called its *home*. The home maintains the master copy of the object, from which all cached copies are derived. The protocol supports multiple writers: updates to an object are detected and propagated from a writer to its home as a difference (*diff*) between the modified object and a reference copy (twin) created before the first write following invalidation.

Similarly to the traditional implementation of LRC [10], our DSM protocol is *invalidation-based*, *i.e.*, it maintains coherence through modification records called *write notices*. The releaser passes write notices to the acquirer of a lock when the ownership passes from the former to the latter. The acquirer invalidates the cached objects according to these write notices and saves them, so that it can pass them forward.

Local writes are grouped into *intervals* delimited by *release* operations that transfer lock ownership to another thread. At the end of each interval, the diffs are calculated and sent to the corresponding homes. A thread's local logical time is defined as a counter of local intervals. A local vector of logical times, a *vector clock*, tracks the (partial) order of releases occurring at any thread.

3.1 Refining Scalability of Home-based Protocols

We refine the scalability of the existing home-based LRC protocols in the context of data *timestamps* and maintenance of write notices.

Home-based implementations of LRC [23][13], associate a *vector timestamp* with each copy of a coherency unit (CU) (here object) and each write notice. For valid CUs the timestamps indicate their version. In a write notice, the timestamps indicate the version that the current thread needs to obtain when accessing the local copy of CU. The timestamps ensure that a version not older than required is brought from home. It also allows checking whether the required version is locally available before fetching it.

Vector timestamps do not scale well because each timestamp is a vector of integers with a number of entries equal to the number of threads in the system. Since a timestamp is attached to each valid CU and each write notice, the total space occupied by timestamps can be significant in a large-scale environment. Moreover, since nodes constantly exchange the timestamps of modified CUs, the timestamp's size has a non-negligible impact on communication overheads. In a Java-based environment, such as ours, both the space and communication overheads are even more significant, because there is a greater number of coherency units (*i.e.,* objects) and consequently a greater number of write notices.

Our DSM protocol uses *scalar timestamps* (Section 2.2 in [8]) instead of vector timestamps, thus restricting the size of a timestamp to a single integer. The downside of using scalar timestamps is the requirement to delay completion of a lock transfer, until diffs are propagated to the corresponding homes. Multithreading overlaps some amount of the waiting time with computation of local threads, decreasing the additional overhead. For medium-scale and smaller clusters that do not justify this scalability-efficiency tradeoff, our protocol can be easily modified to use vector timestamps.

Another scalability aspect in which MTS-HLRC refines the home-based protocols is maintenance of write-notices. In [23][13] each thread (process) keeps all write notices that it has ever seen. If not collected, the number of write notices stored on a node is unbounded. Since, as already claimed above, the number of write notices in a fine-grained system is much greater that in a page-based one, their storage can cause memory overflow (especially, in long-running applications).

In general, a write notice can be discarded only after every thread in the system has received it. In order to determine this, knowledge of a per-entry maximum of *vector clocks* of all the threads in the system is required. Therefore, only a global garbage collection mechanism can collect unneeded write notices.

In large-scale systems global collection of write notices is undesirable due to scalability considerations. Instead of global collection, MTS-HLRC limits the number of write notices stored per thread by keeping only the most recent write notice for each CU. The drawback of this approach is that it does not allow threads of the same node to share their write notice records, as do threads in HLRC-SMP [13]. For medium-scale and smaller clusters the standard write notice maintenance scheme can be used.

3.2 Distributed Synchronization Mechanism

The most popular approach to distributed synchronization is *distributed queue*. In this approach, the lock requests are sent to a node managing the lock, which forwards them to the last requester, *i.e.*, the last process in the distributed queue. Each process in the queue that receives a forwarded request remembers the identity of the requester, and thus knows to whom pass the lock ownership when it is no longer needed. Unfortunately, the distributed queue algorithm does not support priorities, releasing the lock to the next requester in queue disregarding thread/process priorities. Since in Java the threads can have priorities, it is desirable that the synchronization scheme favors the threads with the higher priority. Another drawback of the distributed queue approach is the need for communication when performing the *wait*, *notify* and *notifyAll* operations.

In JavaSplit the queue is not distributed but is rather managed by the current owner of the lock, and is passed along with the lock ownership. Similarly to the distributed queue approach, each lock is managed by some node (here, by the *home node* of the associated object). All lock requests are sent to the manager of the lock, which forwards them to the current owner. The current owner adds the forwarded requests into the queue. Since in Java there is little lock contention [3], the size of the queue is usually not too big and therefore the communication overhead of its transfer is negligible. The proposed algorithm can support thread priorities: the current owner needs always to pass ownership to the requester with the highest priority.

In the proposed synchronization mechanism the operations *wait*, *notify* and *notifyAll* [6] require no communication and are completely local. To implement these operations a *wait queue* is managed alongside the regular *request queue* and is passed from releaser to

acquirer. Since in Java the above operations 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 these operations. For example, a thread *T* performing a *notify* operation on a lock *L*, transfers one of the requests from *L's* wait queue to its request queue. Since *T* has to be the current owner of a lock, it is guaranteed that both queues of *L* are locally available.

4. BYTECODE INSTRUMENTATION

The bytecode rewriter of JavaSplit individually transforms each class of the input application. For each original class *mypackage.MyClass*, it produces a rewritten version *javasplit.mypackage.MyClass*. Thus, we create a hierarchy of classes, parallel to the original one, encapsulated in a package called *javasplit*. In a rewritten class, all referenced class names are replaced with the new, *javasplit* names (Figure 2). Therefore, during the distributed execution, the runtime uses only the *javasplit* classes, never using the originals.

In addition to modifications of the referenced classes, the bytecode rewriter performs the following changes.[2] First, the bytecode segments that start execution of new threads are substituted with calls to a handler that ships the thread to a node chosen by the load balancing function. Second, the synchronization operations, identified by the instructions *monitorenter* and *monitorexit* or by *synchronized* methods, are substituted with the appropriate synchronization handlers. Third, before accesses to objects' fields and array elements (*e.g., getfield*, *putstatic, iaload*, *lastore*, *etc.*), the rewriter inserts access checks in order to preserve memory consistency (Figure 3). Fourth, the instrumented class is

[2] Here, we mention only the most significant changes

Original class

```
class A extends somepackage.C {
   // fields
   private int myIntField;
   public char myCharField;
   public B myRefField;
   public java.lang.Vector myVectorField;

   // methods
   protected void doSomething
           (B b, int n)
   { … }

   public B doSomethingElse
           (java.lang.String str)
   { … }
   …
   …
}
```

Instrumented class

```
class javasplit.A extends javasplit.somepackage.C {
   // fields
   private int myIntField;
   public char myCharField;
   public javasplit.B myRefField;
   public javasplit java.lang.Vector myVectorField;

   // methods
   protected void doSomething
           (javasplit.B b, int n)
   {…}
   public javasplit B doSomethingElse
           (javasplit.java.lang.String str)
   {…}
   …
       //DSM fields inherited from the super class
       public byte __javasplit__state;
       public int    __javasplit__version;
       public int    __javasplit__locking_status;
       public long __ javasplit__global_id;
   …
   // DSM utility methods
   public void DSM_serialize
   (DSM_ByteOutputStream out)
   {
           super.DSM_serialize(out);
           out.writeInt(myIntField);
           out.writeChar(myCharField);
           out.writeGlobalIdOf(myRefField);
   }

   public void DSM_deserialize
   (DSM_ByteInputStream in)
   {
           super.DSM_deserialize(in);
           myIntField    = out.readInt();
           myCharField = out.readChar();
           myRefField    = out.readReference();
   }

   public __javasplit__Diff  DSM_diff
   (javasplit.A twin)
   {
      __javasplit__Diff diff = super.DSM_diff(twin);
       if(myIntField != twin.myIntField) {// add to diff }
       if(myCharField != twin.myCharField) {…}
       if(myRefField    != twin.myRefField    ) {…}
       return diff;
   }
   …
}
```

Figure 2. Original class vs. corresponding instrumented class. While the bytecode changes are described in terms of Java source code for the benefit of the reader, the actual changes are at the level of the bytecodes.

….	….
ALOAD 1	// load the instance of class A
DUP	
GETFIELD	A::byte __javasplit__state__
ICONST 0	
IF_ICMPNE	// jumps to the next GETFIELD
DUP	
INVOKESTATIC	Handler::readMiss
GETFIELD	A::myIntField
….	…..

Figure 3. Instrumented read access of the field *myIntField* in class A. The bytecode instructions in bold are added by the instrumentation. Note that the read miss handler takes the missed object as a parameter. If the object is valid for read, *i.e.*, the value of the *state* field is not *0*, only the first 4 added instructions are executed.

augmented with fields indicating the state of the object during the execution (*e.g.,* access permission, version, whether it is locked, *etc*.).[3] This approach enables quick retrieval of the state information and allows the garbage collector to discard it together with the object. Finally, each class is augmented with several DSM utility methods. The utility methods are generated based on the fields of the specific class. The most important utility methods are for serializing, deserializing and comparing (diffing) (Figure 2). The bytecode rewriter does not handle the I/O operations in the user application. Instead, the Java built-in classes that perform low-level I/O are modified[4], as described in Section 4.1.

The Java language allows classes to declare methods, written in a native language, *e.g.*, *C++*, and compiled to a machine-specific code. These methods, called *native methods*, are executed directly on the given hardware rather than on a JVM. Since they are neither portable nor can they be automatically transformed, we do not support user-defined classes with native methods.

4.1 Rewriting Bootstrap Classes

To preserve data consistency and ensure correctness of synchronization operations, we need to transform *all* classes utilized by the runtime. Consequently, *bootstrap classes* (*i.e.*, the standard classes supplied with the JVM) also need to be rewritten.

Some bootstrap classes contain native methods and therefore cannot be automatically rewritten (entirely). To allow execution of applications that use bootstrap classes with native methods, we have manually created *javasplit* versions of those native bootstrap classes that are most commonly used, e.g., *java.io.FileOutputStream*. In many cases, such *javasplit* classes are implemented as a wrapper around the original class.

[3] Actually, not all classes are augmented with the special fields but only the top of the inheritance tree.

[4] Currently non-trivial usage of files makes sense only if the nodes share a distributed file system

Different JVM implementations may come with different implementations of bootstrap classes, all complying with the standard API. We cannot use *javasplit* bootstrap classes originating from different JVM brands in the same execution, because the same classes from different origins may not be interoperable. Therefore, we transform bootstrap classes of a single JVM brand by automatically rewriting their bytecodes and manually creating *javasplit* versions of the classes with native methods. Since the resulting *javasplit* bootstrap classes consist of pure Java bytecodes, they can be correctly executed on any JVM brand. JavaSplit successfully uses transformed bootstrap classes originating from the Sun JDK 1.4.0 on various brands of JVMs.

4.2 Static Fields Transformation

Static fields are variables that have exactly one incarnation, no matter how many instances are created (possibly zero) of the class in which they were defined. Semantically, a static field is a field of the corresponding class object.

For each class *C* with static variables we create a special class *C_static*, incorporating the static variables of *C* as regular, non-static fields. The rewritten class *javasplit.C* is augmented with a constant static reference field, pointing to an instance of *C_static*, which is treated as any other shared object. In the bytecodes, an access to a static field of C is substituted with an access to the corresponding field of the C*_static* instance, preceded by an access check to verify the validity of the *C_static* object. Thus, we use the same memory coherency mechanism for management of both static and regular fields. Although the access pattern to static variables is usually read-only, no optimizations are required because our DSM scheme works well for read-only objects.

4.3 Array Transformation

Although Java treats arrays as objects, creating *javasplit* versions for arrays is harder than for objects. The main problem with rewriting arrays is the fact that there is no array class definition, but merely language constructs declaring arrays based on definitions of existing classes. Therefore we cannot augment an array, as is, with the *javasplit* fields, which are needed by the DSM logic. We solve this problem by creating a wrapper class for each utilized array type, augmenting the wrapper with the desired fields and methods. For example, a wrapper of an array of elements belonging to a class *C* (*C[]*), called *javasplit.array.C*, incorporates the *javasplit* fields and methods as well as a reference to an array of type *javasplit.C[]*. An access to an array is transformed to an access to the array field in the wrapper object, preceded by an access check. Again, we use the same coherency mechanism for regular objects and arrays.

Although currently we treat each array as a single coherency unit, in the future we plan to divide big arrays into several coherency units. The wrapper approach allows this extension by allocating several instances of the *javasplit* fields, one for each region.

4.4 Rewriting Synchronization Operations

In Java applications there is a great amount of unneeded synchronization [4]. Often, (especially in the bootstrap classes), synchronization protects accesses to objects that are not accessed by more than one thread during the whole execution. The overhead of the unnecessary synchronization may be negligible in Java. However, when rewriting bytecodes for distributed execution, one must be extra careful to avoid performance degradation due to the increased cost of the transformed unneeded synchronization.

JavaSplit reduces the synchronization cost of local objects by avoiding invocation of the handlers used for synchronization of shared objects 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 and release operations on a local object simply increase and decrease the counter. Thus, the object is unlocked when the counter equals zero, and locked when it 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 synchronization handlers for shared objects, but also is cheaper than the original Java acquire (*monitorenter*) operation (Section 6.1).

5. RELATED WORK

Java/DSM [22], Cluster VM for Java (former cJVM) [2], and JESSICA2 [20] (based on JESSICA [21]) implement distributed JVMs. Unlike our work, in these systems each node contains a virtual machine that is a nonstandard JVM. In Java/DSM the local VM is very similar to a standard JVM, except that all objects are allocated on an existing C-based software DSM, called TreadMarks [9]. Similar to 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 possesses many desirable capabilities, *e.g.,* support for load balancing via thread migration, adaptive migrating-home protocol for the GOS and a dedicated JIT compiler. The latter feature distinguishes JESSICA2 from most of the similar works. For instance, Cluster VM for Java and Java/DSM, unable to use a standard JIT, do not implement a dedicated one. By contrast, the proposed system is able to utilize any standard JIT, supplied with the local JVM participating in the execution. Due to modification of node's local JVM none of the above works possesses true cross-platform portability.

Hyperion [1] and Jackal [17] compile the sources of a Java program into native machine code. Hyperion translates Java-bytecodes to C source code and then compiles it using a native C compiler. It employs existing DSM libraries to implement its DSM protocol. The DSM handlers are inserted during compilation of the C code. Hyperion does not require any changes in the programming paradigm. Unfortunately, it requires that a node contains hardware-specific libraries, thus compromising portability. 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. Jackal incorporates a distributed garbage collector and provides thread and object location transparency. Of all the Java-based HPC systems known to us, Jackal's approach is the closest to our work. Both systems use a fine-grained, home-based memory consistency protocol. The input of Jackal is source code, whereas the input of our system is

bytecodes. The disadvantage of both Jackal and Hyperion in comparison to JavaSplit is that they compromise portability by compiling the Java sources into machine dependent code. An additional slight drawback of this systems is the usage of the application's source code which can be unavailable or confidential.

Both JavaParty [11] and JSDM [16] use standard JVMs on each participating node, thus attaining full-fledged portability, similar to our system. However, unlike our system, they do not support the original multithreaded paradigm; rather they introduce unconventional programming constructs and style. JavaParty supports distributed parallel programming by extending Java with a preprocessor and a run-time. The main modification to the Java programming paradigm is the introduction of 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 is an SMPD-style multithreaded program. Moreover, the programmer must use special classes provided by JSDM when writing the program and mark the shared objects. By contrast, in our system, the programmer writes a standard Java application, without being aware of the distributed nature of the underlying system.

The distinguishing feature of the proposed runtime in comparison to the existing systems is the combination of transparency and portability.

6. PERFORMANCE EVALUATION

We have evaluated the prototype of JavaSplit using micro-benchmarks and several pre-existing Java applications. In order to verify the portability of our approach, we have performed our tests on JVMs from two different JDKs: Sun JDK 1.4.0 and IBM JDK 1.3.0. Moreover, we have successfully employed nodes with different types of JVMs in the same executions. The *javasplit* bootstrap classes originating from the Sun JDK worked correctly, as expected, when executing on the IBM's JVM.

In the presented performance evaluation we use collections of Intel's Xeon dual-processor machines (2x1.7 GHz) with 1 GB memory, interconnected by 100 Mbit Ethernet. Note that this configuration is not very advantageous in the context of distributed execution, since the relation of bandwidth to CPU power is considerably smaller than in performance evaluations of similar systems [1][16][19].

6.1 Micro-benchmarking

In our micro-benchmarks we evaluate the overhead of instrumentation and the latency of the employed communication layer. Table 1 shows the cost of heap data accesses in the rewritten code in comparison with their cost in the original Java program. In Sun's JVM heap data accesses introduce a slowdown of between 2.2 and 5.6, whereas in IBM's JVM they introduce slowdown between 12 and 55. The slowdown is mainly due to the access checks preceding a data access. The great difference in slowdown between the two JVMs is due to the optimized latency of repeated accesses to the same data in IBM's JVM, which is one order of magnitude smaller than the data access latency in Sun's JVM. Apparently, the access checks stand in the way of optimizations employed in the IBM's JVM. Despite the great slowdown shown for the IBM's JVM in Table 1, none of the

tested real applications has ever exhibited such instrumentation slowdown. We attribute this to non-trivial access patterns of these applications. To reduce the overhead of the heap data accesses, we are currently working on methods to eliminate unnecessary access checks, as well as try to minimize the cost of a singe access.

Table 2 shows the cost of a local acquire operation, *i.e.,* an acquire which does not result in communication. Although there is a considerable overhead acquiring a shared object, acquires of local objects cost less than the original Java acquire. This is due to the optimization described in Section 4.4. In Table 3 we present the latency of the utilized communication layer for different message sizes. During the execution, the nodes exchange messages ranging from several bytes to several thousands bytes.

	Sun			IBM		
	Original	**Rewritten**	**Slowdown**	**Original**	**Rewritten**	**Slowdown**
Field read	8.37E-04	1.82E-03	2.17	6.53E-05	1.63E-03	24.9
Field write	9.69E-04	2.48E-03	2.56	6.03E-05	7.36E-04	12.2
Static write	8.38E-04	1.84E-03	2.2	5.98E-05	1.61E-03	26.9
Static read	9.67E-04	2.97E-03	3.1	6.14E-05	7.32E-04	11.9
Array read	9.79E-04	5.45E-03	5.57	9.05E-05	4.99E-03	55.1
Array write	1.23E-03	5.05E-03	4.1	1.94E-04	4.98E-03	25.7

Table 1. Heap Data Access Latency (microseconds)

	Original	**Local Object**	**Shared Object**
Sun	9.06E-02	1.96E-02	2.81E-01
IBM	9.34E-02	5.47E-02	3.27E-01

Table 2. Local Acquire Cost (microseconds)

Message size (bytes)	**Sun**	**IBM**
65	0.6421	0.0917
650	0.6511	0.1963
6500	0.9966	0.8125
65000	6.3694	5.9984

Table 3. Communication Latency (milliseconds)

6.2 Benchmark Applications

We show performance evaluation of the following applications: (i) Traveling Salesman Problem, (ii) Fourier coefficient analysis (a.k.a. Series), and (iii) 3D Ray Tracer. The latter two applications are a part of the Java Grande Forum Benchmark Suite [14][15].

In all our measurements two application threads were executed on each of the dual-processor nodes. We have performed separate measurements for different JVMs. The bytecodes of the executed programs were produced by compilation of the benchmarks using the IBM JDK 1.3.0. To calculate the speedup, we divide the execution time of the original (unmodified) Java application with two threads on a single dual-processor machine by the execution time in JavaSplit. Note that the speedup is calculated separately for each JVM, each calculation based on the execution times produced by the same JVM.

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=18. The Series benchmark 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=100000. The 3D Ray Tracer renders a scene containing 64 spheres at resolution of N X N pixels. The worker threads of this application independently render different rows of the scene. We run the Ray Tracer for N=500.

In general, the performance of the three benchmarks exhibit speedups close to be proportional to the number of nodes. Partially this is due to relatively low inter-thread

cooperation inherent to these applications. The efficiency of each added machine remains almost constant, although is much below 100% due to the instrumentation slowdown factor of 1.5-6 for SUN and 3-5.5 for IBM. The difference in the instrumentation slowdown exhibited by the same JVM executing different applications is due to different types of data accesses prevailing in each application. In TSP there is a great number of array accesses, Series accesses mostly regular fields, whereas Ray Tracer frequently accesses static variables.

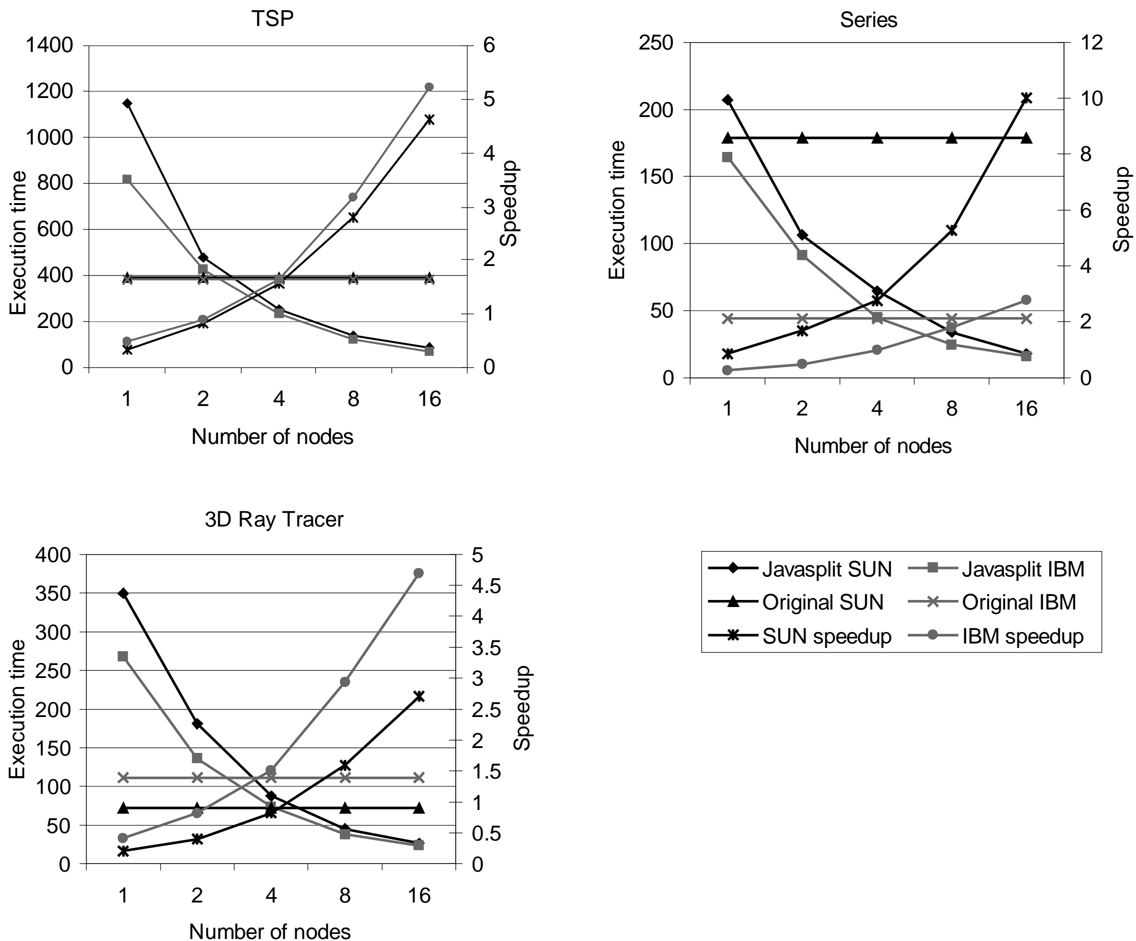

Table 4. Execution times (in seconds) and speedups.

In Series, the speedup obtained by the IBM's JVM is significantly lower than the one obtained by the Sun's JVM, although the execution times on JavaSplit are not very different. This is due to the much lower execution time of Series on a single IBM's JVM in comparison to the execution on a single Sun's JVM. In Ray Tracer we observe a similar phenomena, but this time the Sun's speedup is low due to lower execution time on a single JVM. We attribute this behavior to interference of the instrumented data accesses with various data access optimizations employed by the two JVMs. Since we are planning to employ aggressive access check elimination techniques such as those used in [19], we expect that in the future we will get similar speedups for different JVMs.

Having implemented the prototype of JavaSplit, we strongly feel that there are still various optimizations that have great potential to boost the performance of the system and to enable efficient execution of applications with more inter-thread cooperation. The instrumentation overhead can be significantly reduced by the means of escape analysis. Also, the runtime in general can be further improved by existing optimization techniques utilized in similar Java-based distributed systems, *e.g.*, [18] and [20].

7. CONCLUSION

In this paper we have described the general design and preliminary performance evaluation of JavaSplit, a runtime that executes a standard Java application on interconnected commodity workstations. Thus, we have extended a line of works in the area of Java-based distributed computing. The novelty of the proposed solution is in the combination of portability with the original Java programming paradigm. To enable efficient execution on large collection of workstations, the proposed system was designed to be highly scalable.

We view this work as a first step in providing a convenient computing infrastructure for using large-scale and possibly non-dedicated environments. The underlying question is whether (mostly idle) Internet and enterprise interconnects are sufficiently fast to efficiently support high-level programming paradigms, such as distributed shared memory. Java, as a popular multithreaded programming language, is best suited for this experiment.

8. REFERENCES

[1] G. Antoniu, L. Bouge, P. J. Hatcher, M. W. MacBeth, K. A. McGuigan and R. Namyst. Compiling Multithreaded Java Bytecode for Distributed Execution. *In the Proceedings of the European Conference on Parallel Computing*, Munich, Germany, August 2000.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 4--11, 1999.

[3] Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. Thin locks: Featherweight synchronization for Java. *In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada, May 1998), ACM, pp. 258--268.

[4] J. Choi, M. Gupta, M. Serrano, V. Shreedhar, and S. Midkiff . Escape analysis for Java. *In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* Nov. 1999.

[5] M. Dahm. Byte code engineering. *In Proceedings of JIT'99*, Germany, 1999.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Second Edition. *Addison-Wesley*, Boston, Mass., 2000.

[7] M. W. Hicks, S. Jagannathan, R. Kelsey, J. T. Moore, and C. Ungureanu. Transparent communication for distributed objects in Java. In *ACM Java Grande Conference*, pages 160--170, 1999.

[8] L. Iftode, *Home-based Shared Virtual Memory*, Ph. D. Thesis, Princeton University, Aug. 1998

[9] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *In Proc. of the Winter 1994 USENIX Conference*, pages 115--131, 1994.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. Of the 19th Annual Int'l Symp. on Computer Architecture*, pages 13--21, 1992.

[11] M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225--1242, 1997.

[12] Pugh, W. The Java memory model is fatally flawed. Concurrency: Practice and Experience 12, 6 (May 2000), 445—455.

[13] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design, simulations, implementation and performance. In *Proc. of the 4th International Symposium on High Performance Computer Architecture*, Las Vegas, February 1998.

[14] L. A. Smith and J. M. Bull, A Multithreaded Java Grande Benchmark Suite, in Proceedings of the *Third Workshop on Java for High Performance Computing*, Sorrento, Italy, June 2001.

[15] L. A. Smith, J. M. Bull and J. Obdrzalek, A Parallel Java Grande Benchmark Suite, in proceedings of SC2001, Denver, Colorado, Nov. 2001

[16] Y. Sohda, H. Nakada, and S. Matsuoka. Implementation of a portable software DSM in Java. *In Java Grande/ISOPE'01*, pages 163--172, Palo Alto, CA USA, 2001

[17] R. Veldema, R. A. F. Bhoedjang, and H. Bal. Distributed Shared Memory Management for Java. *In Proc. sixth annual conference of the Advanced School for Computing and Imaging* (ASCI 2000), pages 256--264, Lommel, Belgium, June 2000.

[18] 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. *In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP'01), Snowbird, Utah, 2001, pp. 83-92

[19] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime Optimizations for a Java DSM Implementation. *In Proc. Joint ACM JavaGrandeISCOPE 2001*.

[20] Wenzhang Zhu, Cho-Li Wang and Francis C.M. Lau, JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support, *IEEE Fourth International Conference on Cluster Computing* (CLUSTER 2002) Chicago, USA - September 23-26, 2002, pp. 381-388.

[21] C.-L. Wang, F. C. M. Lau, and Z. Xu, M. J. M. Ma. JESSICA: Java-enabled single system image computing architecture. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), pages 2781--2787, 1999.

[22] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 9(11):1213--1224, 1997.

[23] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 75--88, 1996.

[24] Java Specification Request 133: Java Memory Model and thread specification revision. http://jcp.org/jsr/detail/133.jsp