Wkshp. on Caching, Coherence, and Consistency (WC3),
New York, NY, June 2002

JVM for a Heterogeneous Shared Memory System *

DeQing Chen, Chunqiang Tang,
Sandhya Dwarkadas, and Michael L. Scott

Computer Science Department, University of Rochester

Abstract

InterWeave is a middleware system that supports the sharing of strongly typed data structures across heterogeneous languages and machine architectures. Java presents special challenges for InterWeave, including write detection, data translation, and the interface with the garbage collector. In this paper, we discuss our implementation of J-InterWeave, a JVM based on the Kaffe virtual machine and on our locally developed InterWeave client software.

J-InterWeave uses bytecode instrumentation to detect writes to shared objects, and leverages Kaffe's class objects to generate type information for correct translation between the local object format and the machine-independent InterWeave wire format. Experiments indicate that our bytecode instrumentation imposes less than 2% performance cost in Kaffe interpretation mode, and less than 10% overhead in JIT mode. Moreover, J-InterWeave's translation between local and wire format is more than 8 times as fast as the implementation of object serialization in Sun JDK 1.3.1 for double arrays. To illustrate the flexibility and efficiency of J-InterWeave in practice, we discuss its use for remote visualization and steering of a stellar dynamics simulation system written in C.

1 Introduction

Many recent projects have sought to support distributed shared memory in Java [3, 16, 24, 32, 38, 41]. Many of these projects seek to enhance Java's usefulness for large-scale parallel programs, and thus to compete with more traditional languages such as C and Fortran in the area of scientific computing. All assume that application code will be written entirely in Java. Many—particularly those based on existing software distributed shared memory (S-DSM) systems—assume that all code will run on instances of a common JVM.

*This work was supported in part by NSF grants CCR-9988361, CCR-0204344, and EIA-0080124, and by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460.

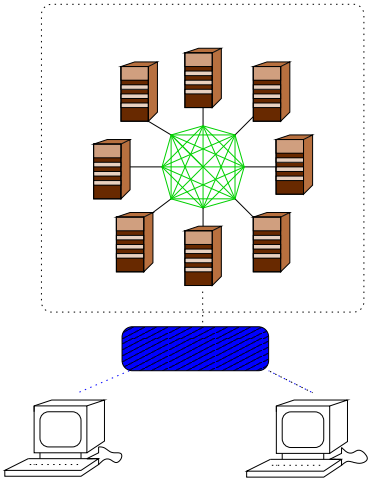

Figure 1: A Heterogeneous Shared Memory System

We believe there is a growing need to share state among heterogeneous JVM nodes, and between Java applications and code written in other programming languages. Figure 1 illustrates this sharing. It shows a an environment in which a simulation (written in Fortran, say) runs on a high-performance cluster consisting of multiple SMP nodes and a low-latency interconnect. Connected to the cluster via the Internet are one or more satellite machines, located on the desktops of researchers physically distant from the cluster. These satellites run visualization and steering software written in Java and running on top of a JVM, operating system, and hardware architecture altogether different from those of the compute cluster.

In the current state of the art, the satellite nodes must communicate with the compute cluster using either a network file system or an application-specific sockets-based message passing protocol. Existing systems for shared memory in Java cannot help, because the simulator itself was written in Fortran, and is unlikely to be re-written. Although Java has many advantages from the point of view of safety, reusability, maintainability, etc. (many of which are appealing for scientific computing [32]), Java implementations still lag significantly behind the performance of C and Fortran systems [18]. Moreover the fact that C

has yet to displace Fortran for scientific computing suggests that Java will be unlikely to do so soon.

Even for systems written entirely in Java, it is appealing to be able to share objects across heterogeneous JVMs. This is possible, of course, using RMI and object serialization, but the resulting performance is poor [6].

The ability to share state across different languages and heterogeneous platforms can also help build scalable distributed services in general. Previous research on various RPC (remote procedure call) systems [21, 29] indicate that caching at the client side is an efficient way to improve service scalability. However, in those systems, caching is mostly implemented in an ad-hoc manner, lacking a generalized translation semantics and coherence model.

Our on-going research project, InterWeave [9, 37], aims to facilitate state sharing among distributed programs written in multiple languages (Java among them) and running on heterogeneous machine architectures. InterWeave applications share strongly-typed data structures located in *InterWeave segments*. Data in a segment is defined using a machine and platform-independent interface description language (IDL), and can be mapped into the application's local memory assuming proper InterWeave library calls. Once mapped, the data can be accessed as ordinary local objects.

In this paper, we focus on the implementation of InterWeave support in a Java Virtual Machine. We call our system J-InterWeave. The implementation is based on an existing implementation of InterWeave for C, and on the Kaffe virtual machine, version 1.0.6 [27].

Our decision to implement InterWeave support directly in the JVM clearly reduces the generality of our work. A more portable approach would implement InterWeave support for segment management and wire-format translation in Java libraries. This portability would come, however, at what we consider an unacceptable price in performance. Because InterWeave employs a clearly defined internal wire format and communication protocol, it is at least possible in principle for support to be incorporated into other JVMs.

We review related work in Java distributed shared state in Section 2 and provide a brief overview of the InterWeave system in Section 3. A more detailed description is available elsewhere [8, 37]. Section 4 describes the J-InterWeave implementation. Section 5 presents the results of performance experiments, and describes the use of J-InterWeave for remote visualization and steering. Section 6 summarizes our results and suggests topics for future research.

2 Related Work

Many recent projects have sought to provide distributed data sharing in Java, either by building customized JVMs [2, 3, 24, 38, 41]; by using pure Java implementations (some of them with compiler support) [10, 16, 32]; or by using Java RMI [7, 10, 15, 28]. However, in all of these projects, sharing is limited to Java applications. To communicate with applications on heterogeneous platforms, today's Java programmers can use network sockets, files, or RPC-like systems such as CORBA [39]. What they lack is a general solution for distributed shared state.

Breg and Polychronopoulos [6] have developed an alternative object serialization implementation in native code, which they show to be as much as eight times faster than the standard implementation. The direct comparison between their results and ours is difficult. Our experiments suggest that J-Interweave is at least equally fast in the worst case scenario, in which an entire object is modified. In cases where only part of an object is modified, InterWeave's translation cost and communication bandwidth scale down proportionally, and can be expected to produce a significant performance advantage.

Jaguar [40] modifies the JVM's JIT (just-in-time compiler) to map certain bytecode sequences directly to native machine codes and shows that such bytecode rewriting can improve the performance of object serialization. However the benefit is limited to certain types of objects and comes with an increasing price for accessing object fields.

MOSS [12] facilitates the monitoring and steering of scientific applications with a CORBA-based distributed object system. InterWeave instead allows an application and its steerer to share their common state directly, and integrates that sharing with the more tightly coupled sharing available in SMP clusters.

Platform and language heterogeneity can be supported on virtual machine-based systems such as Sun JVM [23] and Microsoft .NET [25]. The Common Language Runtime [20] (CLR) under the .NET framework promises support for multi-language application development. In comparison to CLR, InterWeave's goal is relatively modest: we map strongly typed state across languages. CLR seeks to map all high-level language features to a common type system and intermediate language, which in turn implies more semantic compromises for specific languages than are required with InterWeave.

The transfer of abstract data structures was first proposed by Herlihy and Liskov [17]. Shasta [31] rewrites binary code with instrumentation for access checks for fine-grained S-DSM. Midway [4] relies on compiler support to instrument writes to shared data items, much as we do in the J-InterWeave JVM. Various software shared memory systems [4, 19, 30] have been designed to explicitly asso-

ciate synchronization operations with the shared data they protect in order to reduce coherence costs. Mermaid [42] and Agora [5] support data sharing across heterogeneous platforms, but only for restricted data types.

3 InterWeave Overview

In this section, we provide a brief introduction to the design and implementation of InterWeave. A more detailed description can be found in an earlier paper [8]. For programs written in C, InterWeave is currently available on a variety of Unix platforms and on Windows NT. J-InterWeave is a compatible implementation of the InterWeave programming model, built on the Kaffe JVM. J-InterWeave allows a Java program to share data across heterogeneous architectures, and with programs in C and Fortran.

The InterWeave programming model assumes a distributed collection of servers and clients. Servers maintain persistent copies of InterWeave segments, and coordinate sharing of those segments by clients. To avail themselves of this support, clients must be linked with a special InterWeave library, which serves to map a cached copy of needed segments into local memory. The servers are the same regardless of the programming language used by clients, but the client libraries may be different for different programming languages. In this paper we will focus on the client side.

In the subsections below we describe the application programming interface for InterWeave programs written in Java.

3.1 Data Allocation and Addressing

The unit of sharing in InterWeave is a self-descriptive data *segment* within which programs allocate strongly typed *blocks* of memory. A block is a contiguous section of memory allocated in a segment.

Every segment is specified by an Internet URL and managed by an InterWeave server running at the host indicated in the URL. Different segments may be managed by different servers. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block number/name and offset (delimited by pound signs), we obtain a machine-independent pointer (*MIP*): "`foo.org/path#block#offset`".

To create and initialize a segment in Java, one can execute the following calls, each of which is elaborated on below or in the following subsections:

```
IWSegment seg = new IWSegment(url);
seg.wl_acquire();
MyType myobj =
    new MyType(seg, blkname);
myobj.field = ... ...
seg.wl_release();
```

In Java, an InterWeave segment is captured as an `IWSegment` object. Assuming appropriate access rights, the `new` operation of the `IWSegment` object communicates with the appropriate server to initialize an empty segment. Blocks are allocated and modified after acquiring a write lock on the segment, described in more detail in Section 3.3. The `IWSegment` object returned can be passed to the constructor of a particular block class to allocate a block of that particular type in the segment.

Once a segment is initialized, a process can convert between the MIP of a particular data item in the segment and its local pointer by using `mip_to_ptr` and `ptr_to_mip` where appropriate.

It should be emphasized that `mip_to_ptr` is primarily a bootstrapping mechanism. Once a process has one pointer into a data structure (e.g. the root pointer in a lattice structure), any data reachable from that pointer can be directly accessed in the same way as local data, even if embedded pointers refer to data in other segments. InterWeave's pointer-swizzling and data-conversion mechanisms ensure that such pointers will be valid local machine addresses or references. It remains the programmer's responsibility to ensure that segments are accessed only under the protection of reader-writer locks.

3.2 Heterogeneity

To accommodate a variety of machine architectures, InterWeave requires the programmer to use a language- and machine-independent notation (specifically, Sun's XDR [36]) to describe the data types inside an InterWeave segment. The InterWeave XDR compiler then translates this notation into type declarations and descriptors appropriate to a particular programming language. When programming in C, the InterWeave XDR compiler generates two files: a `.h` file containing type declarations and a `.c` file containing type descriptors. For Java, we generate a set of Java class declaration files.

The type declarations generated by the XDR compiler are used by the programmer when writing the application. The type descriptors allow the InterWeave library to understand the structure of types and to translate correctly between local and wire-format representations. The local representation is whatever the compiler normally employs. In C, it takes the form of a pre-initialized data structure; in Java, it is a class object.

3.2.1 Type Descriptors for Java

A special challenge in implementing Java for InterWeave is that the InterWeave XDR compiler needs to generate correct type descriptors and ensure a one-to-one correspondence between the generated Java classes and C structures. In many cases mappings are straight forward: an XDR struct is mapped to a class in Java and a `struct` in C, primitive fields to primitive fields both in Java and

C, pointers fields to object references in Java and pointers in C, and primitive arrays to primitive arrays.

However, certain "semantics gaps" between Java and C force us to make some compromises. For example, a C pointer can point to any place inside a data block; while Java prohibits such liberties for any object reference.

Thus, in our current design, we make the following compromises:

- An InterWeave block of a single primitive data item is translated into the corresponding wrapped class for the primitive type in Java (such as Integer, Float, etc.).
- Embedded struct fields in an XDR struct definition are flattened out in Java and mapped as fields in its parent class. In C, they are translated naturally into embedded fields.
- Array types are mapped into a wrapped `IW_Array` with a static final integer field for array length and a corresponding array field in Java.

Each Java class generated from the InterWeave XDR compiler for Java is derived from a root InterWeave class `IW_Object` (including the `IW_Array` class). Each such class is generated with a default constructor with parameters for its parent segment object and the block name. The constructor is called once when a block of this type is allocated in an InterWeave segment.

Before a type descriptor is used to allocate data blocks, it must first be registered with the InterWeave run-time system through a call to `RegisterClass`.

3.3 Coherence

Once mapped, InterWeave segments move over time through a series of internally consistent states under the protection of reader-writer locks (static functions `wl_acquire, wl_release, rl_acquire, and rl_release` in `IWSegment`). Within a tightly-coupled cluster or a hardware-coherent node, data can be shared using data-race-free [1] shared memory semantics.

Unfortunately, strict coherence models usually require more network bandwidth than the Internet can provide. To adapt to different network conditions, InterWeave employs two bandwidth-reducing techniques: relaxed coherence models and the use of diffs to update segment versions.

First, we observe that many applications do not always need the very most recent version of a shared segment; one that is "recent enough" will suffice. When writing to a segment, an application must acquire exclusive write access to the most recent version. When reading from a segment, it only needs to fetch a new version if its local cached copy is no longer "recent enough". InterWeave supports six different definitions of recent enough [8], and is designed in such a way that additional definitions can be added.

The second bandwidth reducing technique relies on *diffs* to update segment versions. In InterWeave a server always keeps the most recent version of a segment. When a client acquires a lock on the segment and needs a more recent version, the server computes the differences between the client's current version and its own most recent version. It then sends these differences (and nothing else) to the client. Conversely, when a client releases a write lock on a segment, it computes a diff and sends only its modifications to the segment server.

3.4 InterWeave library

InterWeave has additional features that cannot be described in the space available here. Among these are mechanisms for persistence, security, notification of asynchronous events, and 3-level sharing—SMP (*level-1*), S-DSM within tightly coupled cluster (*level-2*), and version-based coherence and consistency across the Internet (*level-3*). All of this functionality has been implemented in its C library with about 25,000 lines of code, upon which the Java implementation is built.

4 J-InterWeave Implementation

In an attempt to reuse as much code as possible, we have implemented the J-InterWeave system using the C InterWeave library and the Kaffe virtual machine. The implementation comprises three interacting components: `IWSegment` class and its Java native interface (JNI) library, the Kaffe component, and the existing InterWeave C library.

The `IWSegment` class functions as the interface between J-InterWeave applications and the underlying Kaffe and C InterWeave library. The Kaffe component modifies the original Kaffe bytecode execution engine and garbage collector to cooperate with InterWeave objects. The InterWeave C library implements InterWeave functionality.

4.1 IWSegment Class and its JNI Library

In J-InterWeave, the basic InterWeave functionality described in Section 3 is provided to Java applications with the `IWSegment` class (see Figure 2). InterWeave's segment-wise operations (e.g. lock acquire and release) are defined as public methods. System-wide operations (e.g. segment creation or dereference of a MIP) are static member functions. All are native functions, implemented in a JNI system library.

The `IWSegment` constructor is generated by the J-InterWeave XDR compiler. An InterWeave programmer can override it with customized operations.

```
public class IWSegment {
  public IWSegment(String URL,
                   Boolean iscreate);
  public native static
         int RegisterClass(Class type);
  public native static
         Object mip_to_ptr(String mip);
  public native static
         String ptr_to_mip(IWObject Ob-
ject obj);
  ... ...
  public native int wl_acquire();
  public native int wl_release();
  public native int rl_acquire();
  public native int rl_release();
  ... ...
}
```

Figure 2: IWSegment Class

4.1.1 JNI Library for IWSegment Class

The native library for the `IWSegment` class serves as an intermediary between Kaffe and the C InterWeave library. Programmer-visible objects that reside within the IWSegment library are managed in such a way that they look like ordinary Java objects.

As in any JNI implementation, each native method has a corresponding C function that implements its functionality. Most of these C functions simply translate their parameters into C format and call corresponding functions in the C InterWeave API. However, the creation of an InterWeave object and the method `RegisterClass` need special explanation.

Mapping Blocks to Java Objects Like ordinary Java objects, InterWeave objects in Java are created by "new" operators. In Kaffe, the "new" operator is implemented directly by the bytecode execution engine. We modified this implementation to call an internal function `newBlock` in the JNI library and `newBlock` calls the InterWeave C library to allocate an InterWeave block from the segment heap instead of the Kaffe object heap. Before returning the allocated block back to the "new" operator, `newBlock` initializes the block to be manipulated correctly by Kaffe.

In Kaffe, each Java object allocated from the Kaffe heap has an object header. This header contains a pointer to the object class and a pointer to its own monitor. Since C InterWeave already assumes that every block has a header (it makes no assumption about the contiguity of separate blocks), we put the Kaffe header at the beginning of what C InterWeave considers the body of the block. A correctly initialized J-InterWeave object is shown in Figure 3.

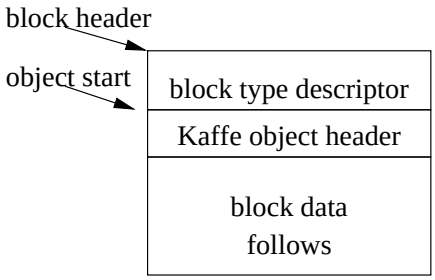

Figure 3: Block structure in J-InterWeave

After returning from `newBlock`, the Kaffe engine calls the class constructor and executes any user customized operations.

Java Class to C Type Descriptor Before any use of a class in a J-InterWeave segment, including the creation of an InterWeave object of the type, the class object must be first registered with `RegisterClass`. `RegisterClass` uses the *reflection* mechanism provided by the Java runtime system to determine the following information needed to generate the C type descriptor and passes it to the registration function in the C library.

1. type of the block, whether it is a structure, array or pointer.
2. total size of the block.
3. for structures, the number of fields, each field's offset in the structure, and a pointer to each field's type descriptor.
4. for arrays, the number of elements and a pointer to the element's type descriptor.
5. for pointers, a type descriptor for the pointed-to data.

The registered class objects and their corresponding C type descriptors are placed in a hashtable. The `newBlock` later uses this hashtable to convert a class object into the C type descriptor. The type descriptor is required by the C library to allocate an InterWeave block so that it has the information to translate back and forth between local and wire format (see Section 3).

4.2 Kaffe

J-InterWeave requires modifications to the byte code interpreter and the JIT compiler to implement fine-grained write detection via instrumentation. It also requires changes to the garbage collector to ensure that InterWeave blocks are not accidentally collected.

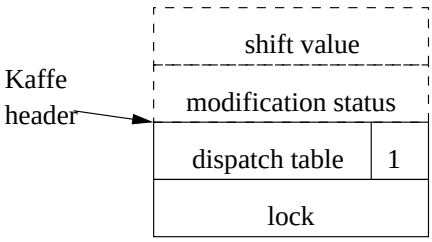

Figure 4: Extended Kaffe object header for fine-grained write detection

4.2.1 Write Detection

To support diff-based transmission of InterWeave segment updates, we must identify changes made to InterWeave objects over a given span of time. The current C version of InterWeave, like most S-DSM systems, uses virtual memory traps to identify modified pages, for which it creates pristine copies (twins) that can be compared with the working copy later in order to create a diff.

J-InterWeave could use this same technique, but only on machines that implement virtual memory. To enable our code to run on handheld and embedded devices, we pursue an alternative approach, in which we instrument the interpretation of *store* bytecodes in the JVM and JIT.

In our implementation, only writes to InterWeave block objects need be monitored. In each Kaffe header, there is a pointer to the object method dispatch table. On most architectures, pointers are aligned on a word boundary so that the least significant bit is always zero. Thus, we use this bit as the flag for InterWeave objects.

We also place two 32-bit words just before the Kaffe object header, as shown in Figure 4. The second word—`modification status`—records which parts of the object have been modified. A block's body is logically divided into 32 parts, each of which corresponds to one bit in the modification status word. The first extended word is pre-computed when initializing an object. It is the `shift value` used by the instrumented *store* bytecode code to quickly determine which bit in the modification status word to set (in other words, the granularity of the write detection). These two words are only needed for InterWeave blocks, and cause no extra overhead for normal Kaffe objects.

4.2.2 Garbage Collection

Like distributed file systems and databases (and unlike systems such as PerDiS [13]) InterWeave requires manual deletion of data; there is no garbage collection. Moreover the semantics of InterWeave segments ensure that an object reference (pointer) in an InterWeave object (block) can never point to a non-InterWeave object. As a result, InterWeave objects should never prevent the collection of unreachable Java objects. To prevent Kaffe from accidentally collecting InterWeave memory, we modify the garbage collector to traverse only the Kaffe heap.

4.3 InterWeave C library

The InterWeave C library needs little in the way of changes to be used by J-InterWeave. When an existing segment is mapped into local memory and its blocks are translated from wire format to local format, the library must call functions in the `IWSegment` native library to initialize the Kaffe object header for each block. When generating a description of modified data in the write lock `release` operation, the library must inspect the modification bits in Kaffe headers, rather than creating diffs from the pristine and working copies of the segment's pages.

4.4 Discussion

As Java is supposed to be "Write Once, Run Anywhere", our design choice of implementing InterWeave support at the virtual machine level can pose the concern of the portability of Java InterWeave applications. Our current implementation requires direct JVM support for the following requirements:

1. Mapping from InterWeave type descriptors to Java object classes.
2. Managing local segments and the translation between InterWeave wire format and local Java objects.
3. Supporting efficient write detection for objects in InterWeave segments.

We can use class reflection mechanisms along with pure Java libraries for InterWeave memory management and wire-format translation to meet the first two requirements and implement J-InterWeave totally in pure Java. Write detection could be solved using bytecode rewriting techniques as reported in BIT [22], but the resulting system would most likely incur significantly higher overheads than our current implementation. We didn't do this mainly because we wanted to leverage the existing C version of the code and pursue better performance.

In J-InterWeave, accesses to mapped InterWeave blocks (objects) by different Java threads on a single VM need to be correctly synchronized via Java object monitors and appropriate InterWeave locks. Since J-InterWeave is not an S-DSM system for Java virtual machines, the Java memory model(JMM) [26] poses no particular problems.

5 Performance Evaluation

In this section, we present performance results for the J-InterWeave implementation. All experiments employ a J-InterWeave client running on a 1.7GHz Pentium-4 Linux machine with 768MB of RAM. In experiments involving

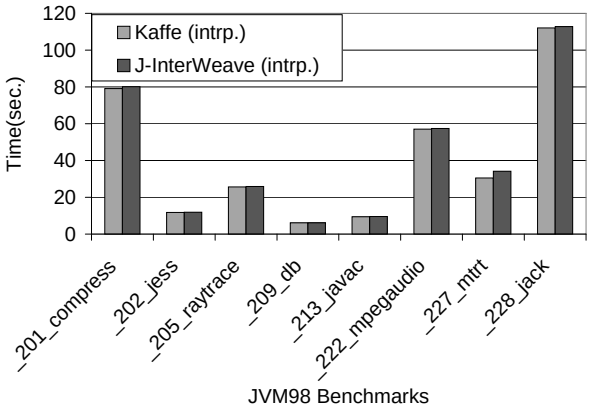

Figure 5: Overhead of write-detect instrumentation in Kaffe's interpreter mode

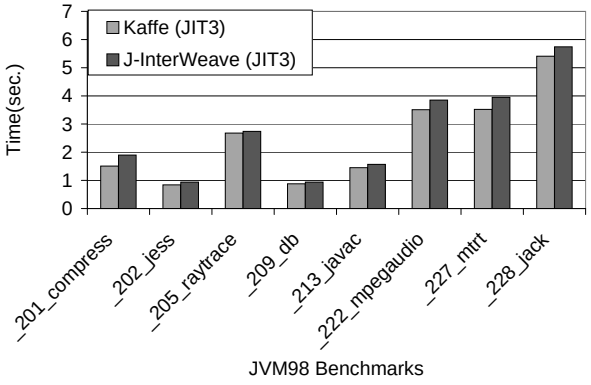

Figure 6: Overhead of write-detect instrumentation in Kaffe's JIT3 mode

data sharing, the InterWeave segment server is running on a 400MHz Sun Ultra-5 workstation.

5.1 Cost of write detection

We have used SPEC JVM98 [33] to quantify the performance overhead of write detection via bytecode instrumentation. Specifically, we compare the performance of benchmarks from JVM98 (medium configuration) running on top of the unmodified Kaffe system to the performance obtained when all objects are treated as if they resided in an InterWeave segment. The results appear in Figures 5 and 6.

Overall, the performance loss is small. In Kaffe's interpreter mode there is less than 2% performance degradation; in JIT3 mode, the performance loss is about 9.1%. The difference can be explained by the fact that in interpreter mode, the per-bytecode execution time is already quite high, so extra checking time has much less impact than it does in JIT3 mode.

The Kaffe JIT3 compiler does not incorporate more recent and sophisticated technologies to optimize the generated code, such as those employed in IBM Jalepeno [35] and Jackal [38] to eliminate redundant object reference and array boundary checks. By applying similar techniques in J-InterWeave to eliminate redundant instrumentation, we believe that the overhead could be further reduced.

5.2 Translation cost

As described in Sections 3, a J-InterWeave application must acquire a lock on a segment before reading or writing it. The `acquire` operation will, if necessary, obtain a new version of the segment from the InterWeave server, and translate it from wire format into local Kaffe object format. Similarly, after modifying an InterWeave segment, a J-InterWeave application must invoke a write lock `release` operation, which translates modified portions of objects into wire format and sends the changes back to the server.

From a high level point of view this translation resembles *object serialization*, widely used to create persistent copies of objects, and to exchange objects between Java applications on heterogeneous machines. In this subsection, we compare the performance of J-InterWeave's translation mechanism to that of object serialization in Sun's JDK v.1.3.1. We compare against the Sun implementation because it is significantly faster than Kaffe v.1.0.6, and because Kaffe was unable to successfully serialize large arrays in our experiments.

We first compare the cost of translating a large array of primitive `double` variables in both systems. Under Sun JDK we create a Java program to serialize double arrays into byte arrays and to de-serialize the byte arrays back again. We measure the time for the serialization and deserialization. Under J-InterWeave we create a program that allocates double arrays of the same size, releases (unmaps) the segment, and exits. We measure the release time and subtract the time spent on communication with the server. We then run a program that acquires (maps) the segment, and measure the time to translate the byte arrays back into doubles in Kaffe. Results are shown in Figure 7, for arrays ranging in size from 25000 to 250000 elements. Overall, J-InterWeave is about twenty-three times faster than JDK 1.3.1 in serialization, and 8 times faster in deserialization.

5.3 Bandwidth reduction

To evaluate the impact of InterWeave's diff-based wire format, which transmits an encoding of only those bytes that have changed since the previous communication, we modify the previous experiment to modify between 10 and 100% of a 200,000 element double array. Results appear in Figures 8 and 9. The former indicates translation time, the latter bytes transmitted.

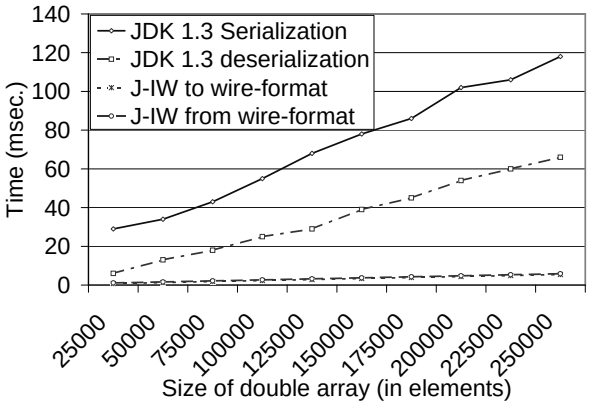

Figure 7: Comparison of double array translation between Sun JDK 1.3.1 and J-InterWeave

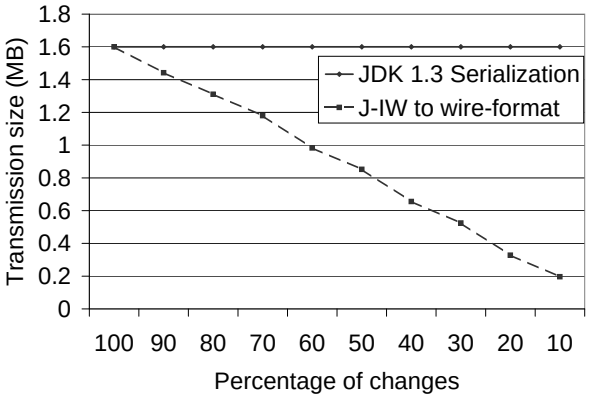

Figure 9: Bandwidth needed to transmit a partly modified double array

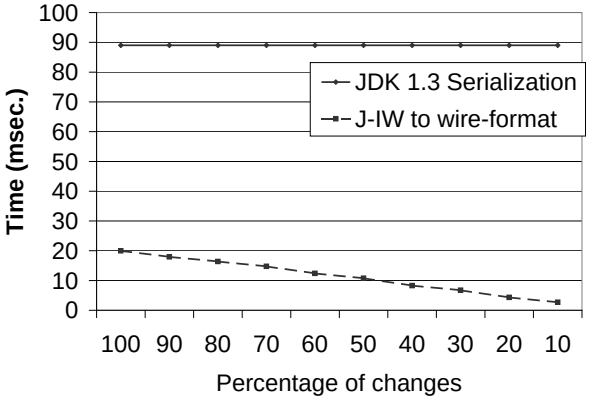

Figure 8: Time needed to translate a partly modified double array

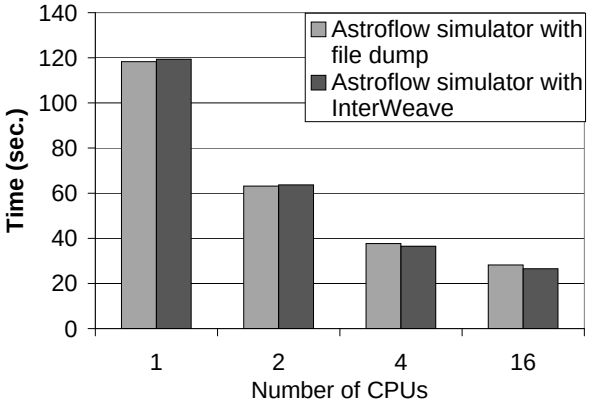

Figure 10: Simulator performance using InterWeave instead of file I/O

It is clear from the graph that as we reduce the percentage of the array that is modified, both the translation time and the required communication bandwidth go down by linear amounts. By comparison, object serialization is oblivious to the fraction of the data that has changed.

5.4 J-InterWeave Applications

In this section, we describe the `Astroflow` application, developed by colleagues in the department of Physics and Astronomy, and modified by our group to take advantage of InterWeave's ability to share data across heterogeneous platforms. Other applications completed or currently in development include interactive and incremental data mining, a distributed calendar system, and a multiplayer game. Due to space limitations, we do not present these here.

The Astroflow [11] [14] application is a visualization tool for a hydrodynamics simulation actively used in the astrophysics domain. It is written in Java, but employs data from a series of binary files that are generated separately by a computational fluid dynamics simulation system. The simulator, in our case, is written in C, and runs on a cluster of 4 AlphaServer 4100 5/600 nodes under the Cashmere [34] S-DSM system. (Cashmere is a *two-level* system, exploiting hardware shared memory within SMP nodes and software shared memory among nodes. InterWeave provides a third level of sharing, based on distributed versioned segments. We elaborate on this three-level structure in previous papers [8].)

J-InterWeave makes it easy to connect the Astroflow visualization front end directly to the simulator, to create an interactive system for visualization and steering. The architecture of the system is illustrated in Figure 1 (page 1).

Astroflow and the simulator share a segment with one header block specifying general configuration parameters and six 256×256 arrays of doubles. The changes required to the two existing programs are small and limited. We wrote an XDR specification to describe the data structures we are sharing and replaced the original file operations with shared segment operations. No special care is required to support multiple visualization clients or to control the frequency of updates. While the simulation data

	Kaffe with File I/O	J-InterWeave
Per frame operation	74 msec.	25 msec.

Table 1: Astroflow Java visualization client performance using J-InterWeave

is relatively simple (and thus straightforward to write to a file), a simulator with more complex data structures would need no special code when using InterWeave. In comparison to writing an application-specific message protocol for the simulator and the satellite, we find data sharing to be a much more appealing mechanism for many applications, leading to code that is shorter, faster to write, and easier to understand.

Figure 10 shows the performance of the simulator using InterWeave, rather than writing its results to files. The time is measured on different configurations of the Cashmere cluster, from 1 CPU on a single node to four CPUs on each of 4 SMP nodes. The InterWeave version of the code uses temporal coherence [8], in which the server "pulls" updates from the simulator at the same frame rate that the file-based version of the code writes to disk. This represents the worst-case scenario in which the visualization client requires the latest values. Particularly at large configurations, InterWeave has a smaller impact on run time than does the file I/O.

In the visualization front end, Table 1 compares the time required to read a frame of data from a file to the time required to lock the segment and obtain an update from the server. For a typical frame J-InterWeave is three times faster than the file-based code. (In this particular application the use of diffs in wire format doesn't lead to much savings in bandwidth: most of the simulation data is modified in every frame.)

Performance differences aside, we find the qualitative difference between file I/O and InterWeave segments to be compelling in this application. Our experience changing Astroflow from an off-line to an on-line client highlighted the value of letting InterWeave hide the details of network communication, multiple clients, and the coherence of transmitted simulation data.

6 Conclusion

In this paper, we have described the design and implementation of a run-time system, J-InterWeave, which allows Java applications to share information directly with applications potentially written using multiple languages and running on heterogeneous platforms, using ordinary reads and writes. To the best of our knowledge, J-InterWeave is the first system to support a shared memory programming model between Java and non-Java applications.

We have demonstrated the efficiency and ease of use of the system through an evaluation on both real applications and artificial benchmarks. Quantitative data shows that the overhead incurred by our implementation is small and that the implementation is much faster in translating data than the object serialization implementation in standard JDK. Experience also indicates that J-InterWeave provides a convenient interface for programmers writing distributed applications in a heterogeneous environment by hiding the details of data translation, coherence, and low-level message exchange.

We are actively collaborating with colleagues in our own and other departments to employ InterWeave in three principal application domains: remote visualization and steering of high-end simulations (enhancing the Astroflow simulation visualization we demonstrated in Section 5), incremental interactive data mining, and human-computer collaboration in richly instrumented physical environments.

References

[1] S. V. Adve and M. D. Hill. A unifi ed formulation of four shared-memory models. *IEEE TPDS*, 4(6):613–624, June 1993.

[2] G. Antoniu, L. Boug, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, March 2001.

[3] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A High Performance Cluster JVM Presenting a Pure Single System Image. In *JAVA Grande*, 2000.

[4] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[5] R. Bisiani and A. Forin. Arichitecture Support for Multilanguage Parallel Programming on Heterogeneous Systems. In *Proc. ASPLOS II*, 1987.

[6] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Java Grande/ISOPE'01*, pages 173–180, 2001.

[7] P. Cappello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D.Wu. JavaLin: Internet Based Parallel Computing Using Java. In *1997 ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997.

[8] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Beyond S-DSM: Shared State for Distributed Systems. Technical Report 744, URCS, 2001.

[9] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level Shared State for Distributed Systems. In *Proceedings of the 2002 International Conference on Parallel Processing*, Vancouver, BC, Canada, August 2002.

[10] G. Cohen, J. Chase, and D. Kaminsky. Automatic Program Transformation with JOIE. In *1998 USENIX Annual Technical Symposium*.

[11] G. Delamarter, S. Dwarkadas, A. Frank, and R. Stets. Portable Parallel Programming on Emerging Platforms. *Current Science Journal published by the Indian Academy of Sciences*, April 2000.

[12] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20.

[13] P. Ferreira and et. al. PerDis: Design, Implementation and Use of a PERsistent DIstributed Store. Technical Report 3532, INRIA, Oct 1998.

[14] A. Frank, G. Delamarter, R. Bent, and B. Hill. Astrofbw simulator. http://astro.pas.rochester.edu/∼delamart/Research/Astrofbw/Astrofbw.html.

[15] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for Internet cooperative applications. In *Middleware'98*, The Lake District, England, 1998.

[16] M. Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *WCAA*, Jan 1999.

[17] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, Oct 1982.

[18] J.M.Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientifi c Applications. In *Java Grande/ISOPE'01*, pages 97–105, Palo Alto, CA USA, 2001.

[19] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: high-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[20] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI'01*.

[21] R. Kordale and M. Abamad. Object Caching in a CORBA Compliant System. *Computing Systems*, (4):377–404, Fall 1996.

[22] H. B. Lee and B. G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, 1997.

[23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[24] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, and Z. Xu. JESSICA: Java-enabled single system image computing architecture. In *ICPDPTA*, pages 2781–2787, June 1999.

[25] E. Meijer and C. Szyperski. What's in a name: .NET as a Component Framework. In *1st OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 22–28, 2001.

[26] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[27] K. Organization. Kaffe Virtual Machine. http://www.kaffe.org.

[28] M. Philppsen and M. Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997.

[29] T. Sandholm, S. Tai, D. Slama, and E. Walshe. Design of object caching in a CORBA OTM system. In *Conference on Advanced Information Systems Engineering*, 1999.

[30] H. S. Sandhu, B. Gamsa, and S. Zhou. The shared region approach to software cache coherence on multiprocessors. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 229–238, 1993.

[31] D. J. Scales and K. Gharchorloo. Towards Transparent and Effi cient Software Distributed Shared Memory. In *SOSP*, 1997.

[32] 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.

[33] Standard Performance Evaluation Corporation. Spec JVM98. http://www.specbench.org/osg/jvm98/.

[34] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Priciples*, St. Malo, France, Oct. 1997.

[35] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, 39(1), 2000.

[36] Sun Microsystems, Inc. *Network Programming Guide — External Data Representation Standard: Protocol Specification*, 1990.

[37] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Support for Machine and Language Heterogeneity in a Distributed Shared State System. Technical Report 783, URCS, 2002. Submitted for publication.

[38] R. Veldema, R. Hofman, R. Bhoedjang, and H. Bal. Runtime optimizations for a Java DSM implementation. In *Java Grande/ISOPE'01*, pages 153–162, 2001.

[39] S. Vinoski. CORBA: Integrating Diverse Applications with Distributed Heterogeneous Environments. *IEEE Communication Magazine*, Feb. 1997.

[40] M. Welsh and D. Culler. Jaguar: enabling effi cient communication and I/O in Java. *Concurrency - Practice and Experience*, 12(7):519–538, 2000.

[41] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency – Practice and Experience*, 9(11), 1997.

[42] S. Zhou, M. Stumm, M. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):540–554, September 1992.