Transparent Distributed Java

Douglas S. J. De Couto Brandon W. Porter Michael B. Taylor*

November 30, 1997

Abstract

We describe our modifications to the Kaffe Java runtime to support transparent distribution of Java threads and objects across multiple processors and machines. We also discuss the design of a shared object system used by the Java runtime to provide transparent object sharing to Java applications. Although we also implemented a version for symmetric multiprocessors (SMPs), this paper focuses on a version designed for networks of workstations (NOWs). We describe our experiences and difficulties in modifying Kaffe, and present some ideas on how to best construct a distributed Java runtime system which includes support for transparent distributed threads and objects. Finally, we present performance results based on micro and application level benchmarks. These results show that a single-threaded Java runtime can be converted into a multi-thread distributed runtime without great loss of performance. The results also however show that the same issues arise with transparency, scalability and performance as in any distributed parallel system.

1 Introduction

Java [5] [8] is one of the fastest growing computer languages. This phenomenon is due in part to Java's massive industry support. But it is also due to Java's convenient language structures, libraries, and runtime system. Because a Java program can be compiled into a single set of bytecodes which will run on different machines, and because the Java language has built-in threading support, Java provides a natural platform for distributed heterogeneous computing on networks of workstations (NOWs) [1] and symmetric multiprocessors (SMPs).

We undertook this project with the vision of modifying a Java runtime system to transparently execute multithreaded Java programs across a single processor, an SMP, or a NOW. When a programmer starts a thread in a Java program, that thread should be able to run on any processor in the system, whether that processor is on a remote computer or is the local processor. Moreover, the programmer should not have to explicitly specify whether a thread is remote or local. Also, threads running on different machines should be able to share Java object data, and the Java synchronization semantics for multiprocessing Java programs (on an SMP or a NOW) should be the same as for Java programs running on a single processor.

A distributed Java runtime system like the one we have created enables many interesting parallel applications to be written quickly and easily in Java. This enables large scale distributed computation for applications with light synchronization and inter-thread communication needs; the canonical example is breaking cryptography keys using a large number of Internet clients. But the system that we envision will also support more classical parallel applications, such as ray tracers and matrix multiplies, or sorts and searches.

We chose to construct this distributed Java runtime system by modifying Kaffe[1], a freely available Java intepreter. We modified Kaffe to run distinct Java

*{decouto,bwporter,mbt}@mit.edu

[1] Available at http://www.kaffe.org.

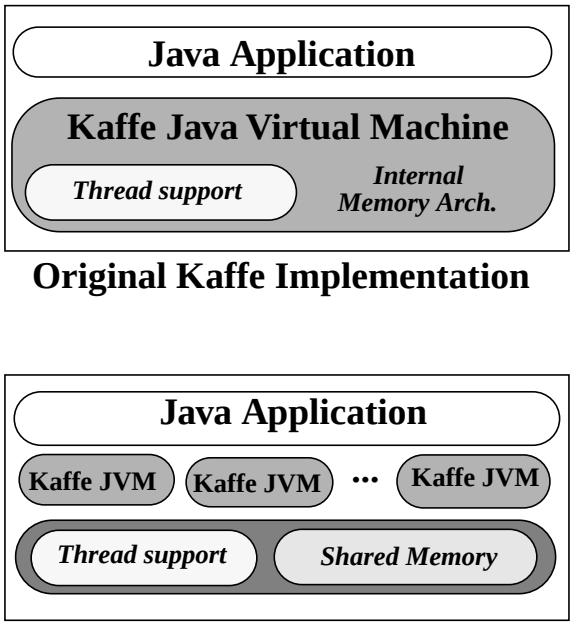

Figure 1: System overview.

threads in distinct processes. These processes share data through a shared object interface. These processes can run on the same processor, on different processors on an SMP, or on different machines in a NOW. We also implemented a distributed locking system to support Java's synchronization semantics across multiple machines.

To keep the project within the scope of a class project, there are a number of challenges which we do not attempt to address. Most notably, we do not try to make sense of distributed I/O issues such as disk, display, and network access operations. Additionally, our implementation does not perform any garbage collection on shared data, and does not use the Kaffe "just-in-time" compiler for Java bytecodes. Finally, we do not attempt to address the time-honored challenges of dynamic load-balancing and thread migration.

Our performance evaluations show that our modifications work reasonably well. Microbenchmarks show that our modified virtual machine operations do not add significant overhead, and application level benchmarks of some classical parallel applications show that we can (with careful attention to the memory system) obtain near linear speedup for small numbers of processors, with a constant factor performance degradation.

We present a brief review of other distributed Java work in Section 2. Section 3 describes the design and implementation of our changes to Kaffe and the subsytems that we created to support a distributed Java system. We discuss our experiences and problems modifying Kaffe in Section 4. Section 5 presents measurement and analysis of the performance of our modifications, and is followed by our conclusions on how to build a better distributed Java runtime environment in Section 6.

2 Related Work

There are several current approaches to implementing a distributed Java system. The Java Remote Method Invocation (RMI) [6] standard is a remote procedure call mechanism that allows programmers to explicitly manage and distribute computation across multi-

ple machines. RMI also supports heterogeneous platforms through data conventions. In practice, RMI implementations require a relatively high degree of programming overhead – up to 66% more code for some applications[9]. A distributed thread implementation uses the built-in language constructs for threads, and consequently requires less programming effort by application developers.

JavaParty [9] allows programmers to annotate threads and classes in Java code as "remote". It then preprocesses the annotated code into regular Java code that uses RMI to distribute objects and computation across a network. JavaParty also allows programs to access objects stored locally without the overhead of RMI.

Javelin [3] provides an application level framework for distributing computation over the Internet. It is essentially a broker system where clients send tasks to a broker; the broker then distributes the tasks to participating computers. It does not leverage Java's built in threading model, and provides a limited programming model.

Java/DSM [13] is the system most similar to our work. It too aims to provide a transparent distributed Java runtime environment on heterogeneous computers. Java/DSM uses the TreadMarks [7] software shared memory system to share memory and perform synchronization; Java/DSM also performs data type conversion and garbage collection across different machines. According to [13], Java/DSM is also not a complete distributed Java runtime: it does not fully support I/O, and does not have location transparent threading. There is no information about how Java/DSM implements distributed threads, nor are there any reported performance measurements.

Our work differs significantly from Java/DSM in that we are using an object-based software shared memory system that supports variable object sizes (see Section 3.2). Because Java/DSM is built upon TreadMarks, it shares page sized chunks of data at a time. Java/DSM does take extra steps to reduce overheads due to page sharing, but there is no indication about what sorts of penalties page sharing imposes upon the Java/DSM system or how well the Java/DSM optimizations work. Another difference between our work and Java/DSM is that we provide location transparent thread distribution.

A more detailed discussion of these distributed Java systems and software shared memory systems can be found in [4].

3 Design and Implementation

The salient parameters for any parallel Java runtime are: how threading is implemented, how sharing of objects is facilitated between these threads, if constant pool objects and internal data structures are shared objects, how garbage collecting is performed on these objects, how I/O is performed, and finally, the manner and degree to which the bytecodes are processed to accelerate the computation.

In attempting to answer the above questions, this section describes the design and implementation of our changes to the Kaffe interpreter to support distributed Java. We first describe how we distributed and managed Kaffe's Java threads across multiple processors and machines. We then discuss the options available for implementing the software-based shared memory system which allows those threads to communicate, and describe why we decided to implement an object-based system. We discuss relevant aspects of our shared memory implementations for SMP machines and NOWs. We conclude with an explanation of how we partitioned Kaffe's internal data structures and constant pools across multiple processors and the shared memory. As was stated in the overview, we do not attempt to address the issues of garbage collection, I/O or bytecode translation.

3.1 Threading Design

We designed a new threading system to replace Kaffe's built-in threading system. This new system executes each thread in its own different process, possibly on different machines.

Our high level model (Figure 1) is that there will be multiple instances of Kaffe running. Some instances of

Kaffe will execute threads started by other instances of Kaffe; they in turn can start their own threads. When executing a Java application, a main Kaffe instance will start up. When it needs to start a thread, it send that thread to a waiting Kaffe instance via a central thread manager. This requires a pool of Kaffe instances waiting to receive threads to work on. However, this is conducive to the ideal distributed Java model! Computers on a network can leave Kaffe instances running, waiting for work. When someone starts a Java application, its threads will be transparently distributed across any waiting Kaffe instances.

Unmodified Kaffe, however, implements Java threads at user level in a single process on a signal thread, in the "cthreads" manner. To emulate scheduling interrupts, Kaffe installs a handler for the UNIX alarm signal; when Kaffe is signalled, it switches its stack to the next thread stack using assembly code, reschedules the alarm, and returns from the alarm handler.

We modified Kaffe to activate a new process for each new Java thread. A third alternative was to run a single multithreaded instance of Kaffe on each machine in a NOW, and keep track of which threads are local and which threads are running in remote Kaffe instances. However, this approach would have involved excessively complicated changes to Kaffe (discussed in Section 4.3). Also, while processes have more heavyweight startup times and contexts, a one-process-per-thread approach has the advantage of having greater compatibility across unix systems.

Our threading design consists of three parts: a thread manager, thread clients, and thread servers. The thread manager coordinates sending threads from thread clients to thread servers. Thread clients initiate thread execution using *dt_send_thread(id)*. This call tells the thread manager that it should start a thread somewhere that executes the `run` method of the object identified by *id*. Thread servers call *dt_get_thread()*, which waits for an object identifier from the thread manager. When the manager returns an object identifier *id* to the server, the thread server executes the `run` method of the object identified by *id*. These calls are summarized in Table 1.

dt_send_thread(id)	Send the object identified by *id* to the thread manager; manager will direct thread server to execute the object's `run` method.
id dt_get_thread()	Recieve an object identifier from the thread manager; this constitutes a promise to execute that object's `run` method.

Table 1: Distributed thread operations.

We modified the Kaffe interpreter to run in two modes. The first mode is the regular mode, where the interpreter receives a class name from the command line, and executes the `main` method of that class. The second mode is the *thread server* mode. In this mode, the interpreter initalizes its data structures and loops forever, calling *dt_get_thread* on each loop iteration to ask the thread manager for threads to run.

We created our own implementation of the Java thread class so that when a Java program running in Kaffe calls a thread object's `start` method, Kaffe will make a *dt_send_thread* call with that object's identifier. This call sends the object identifier to the thread manager, which forwards the indentifier to a waiting thread server.

The thread manager keeps two queues. One queue holds the incoming object identifiers, while the other queue holds a list of thread servers waiting for a thread to run. When a thread client sends an object identifier to the thread manager, the manager sends that identifier to the first thread server listed in the thread server queue, and removes that thread server from the queue. If there are no thread servers waiting, the object identifier goes on the back of the object identifier queue. When a thread server requests an object identifier from the thread manager, the manager returns the first object identifier on the object identifier queue, and removes it from the queue. If there are no object iden-

tifiers in the object identifier queue, the thread manager places the thread server on the back of the thread server queue to wait for an object identifier.

Our thread manager can support multiple distributed Java programs running simultaneously. Also, because each Kaffe thread server continuously asks for threads, each thread server may perform computation for several different applications over time.

An important part of any threading model is the synchronization support provided. Our design leverages a locking server which supports the acquiring and releasing of locks over TCP/IP. This server is used to implement the basic synchronization primitives of Java. Additionally, our system supports sequential consistency, which can be useful in specifying the behaviour of the system. We do not however recommend that programs take advantage of these semantics, since they are not gauranteed by the Java standard.

3.2 Software Shared Memory Design

The design of the software shared memory system was motivated by the desire to share memory between multiple instances of the Kaffe interpreter. There are two important determinant bases for classifying shared memory systems: granularity and transparency. Our system is fine-grained but not transparent. A shared memory system is fine-grained if it allows the application to tailor its basic unit of sharing to the underlying data structures. It is transparent if the process to transform a non shared memory application into a software shared memory application does not require programmer intervention.

The Shasta system [11] is both transparent and fine-grained. It rewrites binary executables to transparently reimplement load and store instructions as necessary. It customizes the size of the basic sharing unit, and performs analysis so that memory on the stack is not shared unnecessarily.

The SAM system [10] is finegrained but not transparent. It requires the programmer to make semantic changes in the actual application code.

An example of a coarse-grained and transparent shared memory system is the TreadMarks page-based system [7], which requires minimal changes to the application code and works at the virtual memory management level. However, in these systems, the unit of sharing is not related to the application, which causes them to be prone to false sharing.

The software shared memory system which we implemented is transparent to the Java application developer, but not to Kaffe, which uses it to implement Java objects (i.e. the heap). (The stack and local variables of the virtual machine are held locally, since they can not be referenced by other threads according to the Java standard.) It is an application level, fine grained, variable block size shared "object" system, where each Java object can be located in its own discrete chunk of shared memory. This attempts to prevent much false sharing between different Java objects used in different Java threads, and we believe leads to the best performance results given the object-oriented nature of Java. Given our project time constraints, we decided that we could more successfully implement an application level shared object system (as opposed to a transparent one); this is supported by the fact that we were able to create two implementations of our shared object system.

Our shared memory system supports shared read and exclusive write access to variable sized objects, or "chunks" of shared memory with sequential consistency semantics. Each chunk of memory is identified by an opaque identifier. Also, the semantics of the data in the memory are completely controlled by the application; our system does not enforce any structure upon the bytes that it stores.

The interface to our system supports five client operations (Table 2). *Alloc(n)* returns an identifier to a fresh chunk of n bytes of memory. *Load(id)* acquires shared read access for the chunk of memory named by *id*, and returns a pointer to a copy of the chunk's data in local memory. *Lock(id)* acquires exclusive write access for the chunk of memory named by *id*, and returns a pointer to a copy of the chunk's data in local memory. *Release(id)* relinquishes the client's access to the chunk of memory named by *id*. If the chunk was accessed in exclusive write mode, changes to the data are committed to the shared object system as necessary. All *re-*

id alloc(n)	Allocate and return name for a new memory chunk of *n* bytes.
void *load(id)	Load the chunk named by *id* for shared read access; return pointer to local copy of data.
void *lock(id)	Load the chunk named by *id* for exclusive write access; return pointer to local copy of data.
release(id)	Release the chunk named by *id* from local memory; commit data changes is necessary and the chunk was acquired for write access.
free(id)	Delete the chunk named by *id* from the system.

Table 2: Shared object system operations.

lease calls on a particular chunk must be preceded by a corresponding *load* or *lock* call for that chunk. *Free(id)* deletes the chunk named by *id* from the shared object system; it is an error to delete a chunk if any client has read or write access to the chunk.

Multiple clients can obtain shared read access to a chunk of memory simultaneously. If a client gains exclusive write access to a chunk, any following *load* or *lock* calls on the chunk by other clients will block until the chunk is available. If a client has shared read access to a chunk, *load* calls on the chunk by other clients will succeed immediately, while *lock* calls on the chunk by other clients will block until the chunk is available for exclusive write access. We also make the restriction that a client can only access one chunk at a time, whether for read or write access; this access is atomic. Also, as discussed in Section 4.3, this atomicity constraint helps to ensure that our changes to the Kaffe interpreter memory system do not create deadlocks.

3.2.1 SMP Implementation

We have produced two implementations of our software shared memory system. The first implementation runs on SMP systems, and supports sharing chunks of memory between different processes running on the same machine. We produced it to help start integrating and testing our changes to Kaffe while we were still completing our NOW shared memory implementation.

In the SMP implementation, when clients get access to a chunk they obtain a pointer directly to that chunk in the UNIX shared memory region. Because of this, our SMP shared object implementation can not enforce the read/write constraints mandated by the interface, except through client co-operation. Each client is an instance of our modified Kaffe Virtual Machine; consequently we needed to ensure that our modified Kaffe implementation obeyed the read/write constraints for the shared memory system.

3.2.2 NOW Implementation

Our implementation of the shared object system for NOWs is more sophisticated. It consists of an object server which communicates with client processes over TCP, using a read-modify-invalidate protocol (Table 3) with write-back caching. For fast access to memory chunks the clients cache memory chunks in a hash table indexed by their opaque identifier. This cache mirrors a hardware shared memory cache except that it does not suffer from conflict or capacity misses.

Client processes can request a new memory chunk or delete existing memory chunks from the server via the *new* and *delete* messages respectively. Clients request a memory chunk from the server using the *readAccess* and *writeAccess* messages for shared read and exclusive write access respectively. When the server receives a *writeAccess* message, three different things can happen. If any clients have shared read copies of the memory chunk, the server sends *invalidate* messages to these clients and returns the requested memory chunk to the process requesting write access. Or, if a client has exclusive write access to the requested chunk, the server sends a *flush* message to the client with write access; when the server receives a response to the flush message with the modified data, it returns the modified data to the new requester. If a client receives an *invalidate* or *flush* request for a memory chunk, it removes

Client initiated messages

Message	Response
new	A fresh memory chunk name.
delete	none
readAccess, writeAccess	Memory chunk data.
die	Possible *flush* messages, followed by an *okToDie* message.

Server initiated messages

Message	Response
invalidate	none
flush	Modified object data
okToDie	none

Table 3: SMP shared object implementation consistency protocol messages.

that chunk from its cache. Finally, if no clients have a copy of a memory chunk, then the server can simply respond to a *writeAccess* message with the memory chunk data.

The server responds to a *readAccess* message in the same way that it responds to a *writeAccess* message, except that it does not need to send *invalidate* messages to clients who already have read access to the memory chunk.

Flush responses are sent to the server on a "bypass" TCP connection, separate from the regular server-client message connection. This allows the server to quickly receive flush responses from clients without having to process all other outstanding client messages.

Finally, clients can send a *die* message to the server before exiting; this causes the server to send a *flush* message to the client for each memory chunk for which the client has exclusive write access. When the client responds to all the flush messages, the server sends it a final *okToDie* message, which tells the client that it is allowed to exit.

On each call into the client interface, the client processes as many outstanding messages from the server as possible. The client then sends a message to the server if appropriate. *Lock* and *load* calls produce *readAccess* and *writeAccess* messages respectively. *Alloc* and *free* calls produce *new* and *delete* messages respectively. The client interface does nothing for *release* calls, except note that the relevant chunk can now be invalidated or flushed by the server. The client cannot respond to *flush* or *invalidate* messages from the server until the appropriate memory chunks have been freed by the client program; since the client program cannot lock or load more than one memory chunk at a time, this is not a problem.

3.3 Virtual Machine Internals

Due to our decision to leverage the existing Kaffe virtual machine, our design choices were significantly limited and in some cases, determined, by the particulars of the base implementation. One of these important design choices was how to manage Kaffe's internal data structures once the runtime became distributed.

There are three main sets of data structures that are required by all threads running in a program on the virtual machine (Figure 2). The first data structure is the Java constant pool; there is one constant pool per Java class. The constant pool contains information about the class, such as its name, field names and types, method names, types and bytecode. The constant pool also contains string, numeric, and other constants for the class. String constants are used to name functions called in the class method code: the bytecode calls functions by name. Kaffe build its other data structures by reading the constant pool; it also caches various pointers to data inside the constant pool after processing the method pool data.

The second data structure is the class data structure, which represents a Java class. This structure points to all the class's metadata: its method dispatch table, information about its fields, its superclass and interface classes, and other miscellaneous information such as the class name. Arrays are treated as a special case of the class data structure.

Finally, there is the object data structure, which represents an instance of a Java object. This contains one field, which points to the class method dispatch ta-

```
struct Hjava_lang_class {
  Hjava_lang_object head;
  struct Hjava_lang_class *superclass;
  /* constant pool */
  struct _constants constants;
  Field *fields;
  struct _dispatchtable dtable;
  struct Hjava_lang_class **interfaces;
};

struct Hjava_lang_Object {
  struct _dispatchTable*  dtable;
  /* Data follows on immediately */
};
```

Figure 2: Essential Kaffe data structures.

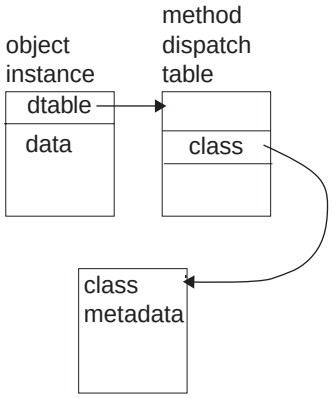

Figure 3: The Kaffe object data structure.

ble; since the dispatch table points back to the class data structure, this also allows Kaffe to associate each object instance with its class metadata (Figure 3). Object instance data follows the dispatch table pointer in memory.

This taxonomy of data structures is further complicated because of the semantics of the Java runtime. There is an object instance associated with each Java class; the class data structure contains an object data structure within it.

We had originally planned to make all three of these common data structures shared objects in our shared object system. Unfortunately, this meant that we would have had to change practically all of Kaffe's pointer accesses into object references. This also would have required us to analyze Kaffe's behaviour in order to formulate a deadlock free locking strategy for data structures that could be traversed by multiple instantiations of Kaffe. Rather than rewrite the majority of Kaffe, we decided on another plan, which involved identifying the internal Kaffe data structures that actually absolutely had to be shared through the shared memory system. We found that placing only object instance data into the shared object system sufficed. Each virtual machine would maintain its own private instance of the constant pool data and class metadata. The two issues raised by this choice are whether modifications to the constant pool will be reflected in other kaffe instantiations and whether the local references (ie pointers) in the constant pool can some how end up being passed to non-local interpreters. The first issue is trivially solved because the constant pool is never modified. The second issue is solved because constant pool objects and metadata are not generally accessable directly by bytecode programs. The exception is string objects, which we must explicitly create "shadow" global objects for when they are loaded onto the Java execution stack. We assume that all the Kaffe processes can access the same Java class files, to load class code and data.

We changed the Kaffe interpreter loop by modifying the implementations of the relevant bytecodes to use our opaque object references instead of C pointers to object data. We modified the implementation of the Java NEW bytecode to allocate objects from our shared object system; all bytecodes that expect object references, such as field stores and fetches, were changed to *load* and *release* the object instance from our shared objects system instead of following a memory pointer.

Three factors complicate the simple concept behind this change. First, to create the Java runtime environ-

ment, Kaffe allocates a large number of objects internally without allocating them through the implementation of the Java NEW bytecode. It expects to be able to access these objects through a simple dereference of a pointer, and does so throughout the code. Furthermore, some of these objects are referenced by interpreted bytecodes in the runtime support libraries. This means that Java objects encountered on the Java stack in the interpreter loop could be either shared objects allocated in our shared object system, or objects allocated locally. The code that implements a bytecode must be able to tell if a Java object reference on the stack is an opaque shared object identifier, or if the reference is a local memory pointer.

We used a "magic cookie" (or "magic number") scheme to differentiate between local object references and shared object references [12]. Each shared class instance is represented locally with a modified form of the object data structure (Figure 4). This stucture starts with a special, unique magic cookie value, and contains the dispatch tables and corresponding shared object id. The magic cookie value is an invalid address and causes a segmentation fault when dereferenced, and is easily noticed in the debugger. In the location where we store the magic cookie, the regular object instance structure stores a pointer. The interpreter can thus easily determine what type of object it is processing. Furthermore, if somehow Kaffe gets ahold of one of our non-local objects, it will cause a segment fault and we can track it down and rework the offending code.

Object references stored in object fields are also handled simlarly. We assume that any shared object contains only opaque references (identifiers) to shared objects, and that locally allocated object contains only local object object references. In practice this inability to mix object types is not very limiting, since most of the "dirty" objects that Kaffe creates are not accessable in the namespace of a user program.

The second main complication is that Kaffe expects to find the class metadata for a particular object instance by following the dispatch table pointer stored at the top of the object instance. But it is not possible to share local dispatch table pointers for a shared object

```
#define kMagicCookieValue 0xCAFE0001

struct mbt_Hjava_lang_Object;
{
  jint magicCookie;
  struct _dispatchTable *dtable;
  sob_id sobId;
  /* Data follows on immediately */
};
```

Figure 4: Modified Kaffe object data structure.

instance across different Kaffe processes; we need a different way to associate shared objects with their class metadata. Our solution is to expand the shared object system client interface to let us associate a class name (stored as an object itself) with each shared object instance. When operating on local objects, the modified bytecode implementations can directly access the object's dispatch table pointer. When operating on shared objects, the bytecode implementations can find the object's class name from the shared object interface (a local hashtable can be used to speed up this process) and use this name to lookup the dispatch table of the class. The pointer to this dispatch table is cached in the modified object structure.

The third problem with our modification of Kaffe's bytecode implementations is native method support. Most of Kaffe's Java runtime support is implemented using native methods (written in C) that are called by the Kaffe interpreter. This code relies on the layout of the Kaffe data structures, especially Kaffe's representation of Java objects. By changing the representation of some Java objects, we introduce the possibility that some native methods will try to access an object pointer that is really a shared object identifier. In general, native methods deal with machine-specific resources such as I/O operations. In our original proposal, we stated that we would not be attempt in our timeframe to support machine-specific resources such as I/O capabilities. Consequently, we avoid the problems associated with native methods by disallowing

I/O operations and other calls that might require native method support except for threading (discussed in Section 3.1). For practical reasons, we did put in place a bytecode combination which will temporarily cause our implementation of Kaffe to create local objects instead of shared objects. This allows us to call functions (like System.Out) which thunk down to native functions. Use of this feature is at best a risky endeavour, and only recommended as a last resort.

4 The Kaffe Experience

Changing the memory and threading semantics of any application are two of the most difficult modifications that can be undertaken. As we mentioned previously, many of our design choices in modifying Kaffe were constrained by our ability to find suitable hooks into the Kaffe system. There were three main types of problems that made it hard to modify Kaffe: our modifications were not in any way anticipated by the Kaffe designers; poor software engineering practices within Kaffe; and system complexities of Kaffe and the Java runtime system that Kaffe implements. We discuss those problems in this section, as well as some issues in measuring the effort required to change Kaffe.

4.1 Unanticipated Modifications

Not only was the Kaffe program itself designed to run in a single address space[2], but the Java runtime environment that Kaffe implements was designed to reside in single address space. Our intent was to modify Kaffe to implement a Java runtime environment that could span multiple address spaces and machines.

In Kaffe, all Java objects and runtime constructs are implemented as C data structures, including transparent pointers; there is no clear way to map these sorts of structures into a shared address space with the semantics described in Section 3.2.

Systems that are meant to run across multiple address spaces are carefully designed to be distributed across more than one address space; the Java runtime environment implemented by Kaffe was not. Nevertheless, we succeeded in leveraging the Java language semantics that Kaffe implements to successfully change the address space model of the Kaffe system, although with much pain.

While Kaffe appears to attempt to support native threads, we are not certain that Kaffe is actually thread safe for multiprocessing threads, and the designers certainly did not anticipate a distributed threads implementation. The built-in thread package that Kaffe uses runs all Java threads and Kaffe code sequentially in a single process. We have not seen any successful editions of Kaffe using a multiprocessing threads package, and the version of Kaffe which we are modifying does not come close to supporting such a package.

4.2 Poor Software Engineering Practices

Another difficulty that we encountered while modifying Kaffe was coping with the poor software engineering practices employed by the Kaffe implementors. These are some of the most serious problems:

Documentation There is no design documentation for Kaffe which describes the overall design philosophy, module layout, or interfaces. The closest approximation to documentation is the Kaffe mailing list, which contains technical discussions on bugs in Kaffe, patches, and other assorted comments.

Lack of Abstraction The Kaffe system does not make very good use of abstraction, nor does it have very good interfaces between modules. For instance, Kaffe's garbage collection, locking and threading modules would be ideal candidates for abstract interfaces, through which the rest of Kaffe handles locking and threading. Indeed, the Kaffe sources are organized in a way that suggests that threading and locking are handled by self-contained modules and sets of functions and macros that can be replaced. However, the locking and threading data structures are often manipulated directly througout the Kaffe program. As a

[2] This is a reasonable assumption for a program to make.

```
#define CHECK_NULL(_i, _s, _n)  \
  cbranch_ref_const_ne((_s), 0, \
      reference_label(_i, _n)); \
  softcall_nullpointer();       \
  set_label(_i, _n)

/* broken usage: */
if (foo)
  CHECK_NULL(bar, baz, bak);
else
  do_something();
```

Figure 5: A bad macro in Kaffe. This macro may not work correctly when used in an `if` statement because the compiler is unable to know how the macro should be used.

result, changes in either of these modules must be propagated throughout the entire Kaffe program. In fact, we were unable to even remove the default threading and garbage collection code from Kaffe after three days of hacking. Instead, we ended up making surgical changes to these sections (set context switch quantum to infinity, never garbage collect) which allow these modules to continue to execute without interfering with our replacement modules. Because of this, we were grudgingly forced to leave the existing Java `Thread` class in place. The programmer must instead derive their threads from our `RThread` class, which implements the same `start` method as the `Thread` class. We had similar problems with the locking and exceptions mechanisms.

Bad Use of Macros Kaffe uses macros extensively, both to try and create interfaces and to reduce typing. These macros are used inconsistently throughout Kaffe, are undocumented, and contribute heavily to code obfuscation. Often, the macros have cryptic names which, when combined with the lack of documentation, make understanding the Kaffe code extremely difficult. The macros are often used recursively, or are contructed in ways that make them hard to use correctly (Figure 5). Finally, and most egregiously, Kaffe uses macros which have side effects on local variables that are not listed as parameters in the macro call. This means that to understand a piece of Kaffe code with macros in it, we have to locate the implementation of every macro to understand the actual effects of the macro and then finally the effects of the code itself. Recursive grep became our friend.

4.3 Complexities and Limitations in the Kaffe Runtime System

The third difficulty we encountered while modifying Kaffe is the complexity of the Java runtime system that Kaffe implements. The Java Virtual Machine is not completely specified; also, there are some inconsistencies and errors in the Java Virtual Machine specification which complicate the Kaffe implementation [2]. For instance, in Java, every class is associated with an object instance that contains information about that class. But every object instance must also be associated with a class. Thus there is a circularity problem when loading the root of the Java class hierarchy, `Java.lang.Object`, which must be resolved by special handling in the Kaffe interpreter [2].

Even the parts of the Java runtime environment which are completely specified are complex and hard to understand. When Kaffe initializes its run time environment, it has to construct and initialize many objects to support threads, I/O, strings, and other various aspects of the Java programming environment. However, Kaffe implements many parts of this system with native code, so the Kaffe initialization sequence is a mixture of Java interpretation, native functions, Java resource allocation, and resource allocation by Kaffe. Because of this, there are many complex and fragile interdependencies which must be accommodated by our changes to Kaffe. Since there is no documentation for the Kaffe design, finding out which parts of the runtime are implemented as native code and which parts must be replaced is a tedious process.

Base Kaffe code	20,116
SMP shared object code	1,548
NOW shared object code + distributed thread support	3,194

Table 4: Total line counts for original Kaffe code and for added support code. The Kaffe line count does not include Java language library code, or the Kaffe "just-in-time" compiler.

file	**add**	**chg**	**del**	**orig**	**new**
exception.c	13	4	0	262	275
gc-inc.h	0	2	0	124	124
kaffe.def	859	226	46	3,273	4,086
support.c	19	10	0	424	443
thread.c	70	12	0	333	403
main.c	81	14	1	294	374
machine.c	33	12	1	235	267
New glue code	458	–	–	–	458
Total	1,533	280	48	4,945	6,431

Table 5: Source file changes. The first three columns of numbers show how many lines were added to each file, changed in each file, and deleted from each file. The last two columns show how many lines each file contained before and after our changes respectively. The line labelled "New glue code" describes new code that was written to emulate many of Kaffe's internal object accessor functions, but using the shared memory subsytem instead.

4.4 Evaluating the Difficulty of Change

It is hard to objectively evaluate the difficulty of change to a piece of software. One commonly used metric is the number of lines changed in the source code; we present these figures in Table 5. We also had to implement memory and threading susbsystems from scratch; the line counts for these subsystems are presented in Table 4, including the line counts for the base Kaffe source code.

These line counts show that most of our effort in actually changing the Kaffe code consisted of writing new glue code and modifying a few hundred lines in `kaffe.def`. However, these relatively small numbers are misleading. The fact is that our changes required us to understand the interaction between almost every line of the base Kaffe source code, which is much more work than is indicated in the tables.

5 Performance

To determine the performance of our change to the Kaffe interpreter, we ran a set of microbenchmarks, as well as two application-level benchmarks.

5.1 Microbenchmarks

We used a variety of microbenchmarks implemented in Java to compare our modified version of the Kaffe interpreter (Kaffe-NOW) with the original version of the interpreter (Kaffe-Ref). The Java code for these benchmarks is listed in Appendix A, and the results are presented in Table 6. These tests were designed to look at the performance overhead incurred by our shared object system.

We measured the times by modifying both versions of the interpreter to obtain the microsecond value of the system clock. When the program attempts to load a special constant value into a stack variabl, the program actual loads the system time. We can then perform calculations and print this time out from within the Java program itself.

As the table shows, code which uses local or stack storage, as well as arithmetic and control code experiences little slowdown. The Kaffe-NOW times for writing and reading array references for cached arrays and integer fields for cached objects are surprisingly close to the Kaffe-Ref times. The slowdowns come from added function calls (to lock objects) and "cookie checks", as well as other verification code. Reading of references are in general slower than writing because reading re-

Benchmark	NOW	Ref	Ratio
Empty Loop	108	106	1.01
New Object	29,135	1,750	16.65
Array Ref Write	279	223	1.25
Array Ref Read	372	190	1.96
Integer Field Write	587	393	1.49
Integer Field Read	590	366	1.61
Ref Field Write	597	281	2.12
Ref Field Read	1941	281	6.91
∞ Int Field Write	1,999,994	223	8,968.58
∞ Int Field Read	1,979,991	190	10,421.01
∞ Ref Field Write	1,999,952	282	7,092.02
∞ Ref Field Read	1,999,963	281	7,117.31

Table 6: Microbenchmark results. Times are in microseconds, and are for 100 operations. Benchmarks marked with ∞ are on uncached data. NOW is our modified implementation of Kaffe, while Ref is the unmodified Kaffe interpreter. Ratio is NOW/Ref.

quires that the types of the references be looked up (it is as fast as it is due to use of memoization). The reference field reading and writing tests slowdowns are in fact due to repeated type lookups.

Since using distributed objects over a network adds more complexity to the code, slowdowns can be expected for all heap data accesses in Kaffe-NOW. We use a variety of caching mechanisms for objects and their type information to minimize network communication; the table shows the drastic speed difference between the cached and uncached memory operations. It is clear that the time to access remote objects is limited by the speed of the network and network protocols. Although there are some software solutions for coping with less than desirable network performance, our philosophy on this is that the improvement of these network operations is orthogonal to our modifications to Kaffe. Our implementation is modular enough that the network interaction code could be easily swapped out. Regardless, the best way to accelerate the network operations would be to a more direct interface to the network. A good start might be using UDP datagrams instead of the more heavyweight TCP protocol. More extreme measure could be taken. The Shasta [11] project, for instance, provided the applications with direct access to the network device.

5.2 Application Benchmark: RayTrace

For our application-level benchmark, we searched for an application that had the following properties: it was easily parallelizable; it was already written as an existing Java program; and it was easily scalable to different sized tasks. We settled on a ray tracer, and scoured the World-Wide-Web for publically available ray tracer implementations. We finally chose Ron Mercer's implementation[3].

We parallelized the ray tracer by distributing an even number of scan-lines to each thread. We were able to change the number of threads that the ray tracer used, which allowed us to test the parallel implementation with different numbers of processors. We also settled upon using a 150x150 pixel image because it took under 1 minute to compute on the regular Java interpreter, while still giving meaningful performance results. After removing the calls to the AWT (which didn't seem to work under regular Kaffe anyways), we ran the tests, varying the number of processors.

The results were extremely disappointing. It was clear that we would not attain speedup on any number of processors! Closer examination of the code revealed that the ray tracer allocated new vector objects for the results of every vector arithmetic operation. As a result, some 6 million objects were slated to be created in the course of rendering the scene. As a result, our system spent most of its time requesting new objects from the server. Worse yet, though, since we had no garbage collection, the heap rapidly exploded. Finally, when the clients finish, their default action is to flush their object caches back to the server. This resulted in the flushing of millions of memory objects that would have normally been garbage collected. All of these issues could have eventually been solved by

[3] `http://www.cinenet.net/users/mercier/trace/jt.html`

our implementation. We could divide the namespace of memory objects so that clients could allocate new objects without notifying the server. If another client requests that object, the server will know which client to ask by looking at the object id. The memory protocols would have become much more complex, however. The clients could have been set to not flush their contents when they end their task. This means however that they must respond to flush events while they are waiting for a new thread to run. Finally, a garbage collector could be implemented to cull unused objects. But this experience taught us a bit about distributed Java. The first is that the increased ease of programming that garbage collection provides (versus say C) can easily make users less aware of how they are using memory. This is deadly in an environment where accessing remote objects can take more time than reading from a disk. The second is, a distributed garbage collector will perform horribly if it attempts to pull in all objects into one place to make garbage collection decisions. Although we could have reworked (we actually did rework it to prove it could be done) the ray tracer code to improve its performance, we thought that it stands better to leave this as an important reminder that no matter how transparent and easy it is to program a parallel system, making that system actually have good performance is seldomly a transparent process.

5.3 Application Benchmark: Matrix Multiply

For our second application benchmark, we chose Matrix Multiply. This one we coded ourselves, attempting to minimize memory transactions as much as possible. This test also falls under the "embarassingly parallel" category. However, it gives us an indication of what the overhead of our complete system is. The implementation was of the classic n-cubed algorithm. References to the quadrants of the matrices are passed to thread objects which compute the result. The master thread then assembles each of the result quadrants into one array. The result matrix was composed of several objects so that none of the compute threads thrashed trying to share write access. The source matrices were shared by all of the threads.

Threads	Time	Speedup
1	101s	.584
2	51s	1.16
4	27s	2.19

Table 7: Speedups for parallel execution. Speedup is relative to the reference implementation of Kaffe, with one thread. Kaffe's threading is relatively lightweight, so the times for different numbers of threads were identical.

Threads	Time	Overhead
1	101s	0%
2	51s	1%
4	27s	6.5%

Table 8: Overheads for multiple thread on Kaffe-Now. Overhead is defined as the deviance away from linear speedup.

As Table 7 shows, the results were unsurprising. The single threaded version showed slow downs due to the extra code required for our distributed Java implementation. With two threads, we actually begin to witness speedup, and with four we are running twice as fast.

As Table 8 shows, the overhead of parallization increases with the number of processors. This appears to be mostly due to contention on the server. This contention could be reduced by using a distributed software shared memory implementation.

6 Lessons

Although we were largely successful in building a distributed Java runtime environment, our conclusions are derived from the extreme difficulties of modifying Kaffe, and in the resulting fragility and complexity of the system. If Kaffe was barely extensible, our system

is probably impervious to change.

Our experience modifying Kaffe has led us to draw this conclusion about building a distributed Java runtime environment: there are enough complicated issues that it may not be desirable or even feasible to modify a regular Java interpreter (such as Kaffe) to provide support for distributed operations. The design of all major subsystems, such as memory management, garbage collection, locking, threading, and I/O, is so different for a distributed Java runtime that any modifications to the code for a single machine implementation of a Java runtime will not be able to use much of the existing code. Indeed, perhaps the only code that could be re-used would be the main interpreter loop and some of the data structure layout (but not implementation). The basic structure of objects, classes, constant pools, and method dispatch tables could be retained, but the implementation and linking should be completely reimplemented so that these data structures reside in the shared memory region. Essentially, one cannot successfully and elegantly implement the change that we chose to make to Kaffe without completely reimplementing a significant fraction of the system. In this sense, it is not a change but rather a complete reimplementaion of the system.

Our experience has shown that the best way to produce a system that properly and completely implements a distributed Java runtime environment is to carefully design and implement it from scratch. This will produce a system that is more robust, more flexible, and better performing. Also, if subsystems and interfaces are designed properly, such a system could support multiple implementations of software shared memory and could be specialized to run on different implementations of shared memory systems without having to change the rest of the system.

Other areas that would absolutely require considerable thought and design effort to attain acceptable performance would be the I/O system and garbage collection.

Finally, it would be highly desirable for a distributed Java system to take advantage of "just-in-time" compilation. In addition to overcoming interpretation overhead, it could be used to improve the distributed memory behaviour. For instance, it could be used to minimize object type lookups, or to allocate objects locally if their references are only propagated locally.

7 Conclusion

We have described our modifications to the Kaffe Java runtime to support transparent distribution of Java threads and objects across multiple processors and machines. We also discussed the design of a shared object system used by the Java runtime to provide transparent object sharing to Java applications. We explained our experiences and difficulties in modifying Kaffe. We presented performance results based on micro and application level benchmarks. These results show that a single-threaded Java runtime can be converted into a multi-thread distributed runtime without great loss of single-threaded performance. The results also however show that the same issues arise with transparency, scalability and performance as in any distributed parallel system. Finally, we presented some ideas on how to best construct a Java runtime system which includes support for transparent distributed threads and objects.

A Microbenchmark Code

For our microbenchmarks, we executed this code in sequence in the Kaffe-Ref and Kaffe-NOW versions of the interpreter. The pieces of code whose execution was measured are labelled in boldface.

Classes:

```
class SmallObject
{
  public int var1;
  public int var2;
  public SmallObject o;
  SmallObject() {}
  init() { o = new SmallObject(); }
}
```

Variables:

```
int         i, j;
SmallObject localObject;
SmallObject localArrayOfObject[]
  = new SmallObject[100];
```

Empty Loop

```
for(i = 0; i < 100; i++)
{
}
```

New Object

```
for(i = 0; i < 100; i++)
{
  localObject = new SmallObject();
}
```

Array Write

```
for(i = 0; i < 100; i++)
{
  localArrayOfObject[i] = localObject;
}
```

Array Read

```
for(i = 0; i < 100; i++)
{
  localObject = localArrayOfObject[i];
}
```

Integer Field Write

```
for(i = 0; i < 100; i++)
{
  localObject.var1 = i;
  localObject.var2 = i;
}
```

Integer Field Read

```
for(i = 0; i < 100; i++)
{
  j = localObject.var1;
  j = localObject.var2;
}
```

Setup Code

```
for (i = 0; i < 100; i++)
{
  so[i] = new SmallObject();
  so[i].init();
}

flushCache();
localObject = localArrayOfObject[0];
```

Uncached Integer Field Read

```
for(i = 0; i < 100; i++)
{
  j = localArrayOfObject[i].var1;
}
```

Setup Code

```
flushCache();
localArrayOfObject[0] = localObject;
```

Uncached Integer Field Write

```
for(i = 0; i < 100; i++)
{
  localArrayOfObject[i].var1 = j;
}
```

Cached Ref Field Write

```
for(i = 0; i < 100; i++)
{
  localArrayOfObject[i].o = localObject;
}
```

Cached Ref Field Read

```
for(i = 0; i < 100; i++)
{
  localObject = localArrayOfObject[i].o;
}
```

Setup Code

```
flushCache();
localArrayOfObject[0] = localObject;
```

Uncached Ref Field Write

```
for(i = 0; i < 100; i++)
{
  localArrayOfObject[i].o = localObject;
}
```

Setup Code

```
flushCache();
localArrayOfObject[0] = localObject;
```

Uncached Ref Field Read

```
for(i = 0; i < 100; i++)
{
  localObject = localArrayOfObject[i].o;
}
```

References

[1] Anderson, Thomas E., David E. Culler, David A. Patterson, and the NOW team. "A Case for Networks of Workstations: NOW." *IEEE Micro.* February, 1995

[2] Bertelsen, Peter. "Semantics of Java Byte Code." Department of Information Technology, Technical University of Denmark. March, 1997. Available at `ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/jvm-semantics.ps.gz`.

[3] Christansen, Bernd O., et al."Javelin: Internet-Based Parallel Computation Using Java." ACM 1997 Workshop on Java for Science and Engineering Computation.

[4] De Couto, Douglas S. J., Brandon W. Porter and Michael B. Taylor. "A Survey of Distributed Java Computing and Software Shard Memory Systems." 6.853 Mock Area Exam, October, 1997. Available at `http://graphics.lcs.mit.edu/~decouto/6.853/project/area-exam.ps`.

[5] Gosling, James, Bill Joy, and Guy Steele. *The Java anguage Specification.* Addison Wesley, 1996.

[6] "Remote Method Invocation Specification." JavaSoft. Available at `http://www.javasoft.com`.

[7] Kehler, Pete, et al. "TreadMarks: Distributed Shared Memory on Standard workstations and Operating Systems." 1994 Winter USENIX.

[8] Lindholm, Tim, and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[9] Philipsen, Michael and Matthias Zenger. "Javaparty – Transparent Remote Objects in Java." ACM 1997 Workshop on Java for Science and Engineering Computation.

[10] Scales, Daniel J. and Monica S. Lam. "The Design and Evaluation of a Shared Object System for Distributed Memory Machines." First Symposium on OPerating Systems Design and Implementation (OSDI), 1994.

[11] Scales, Daniel J., Kouroush Gharachorloo, and Chandramohan A. Thekkath. "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory." Proceedings of the Seventh International Conference on Architectural Support for Porgramming Languages and Operating Systems. October, 1996.

[12] *The Magic Cookie Reference Manual, Volume 1.* SOB Technologies Inc, 1997.

[13] Yu, Weimin and Alan Cox. "Java/DSM: A Platform for Heterogeneous Computing." ACM 1997 Workshop on Java for Science and Engineering Computation.