Joust: A Platform for Communication-Oriented Liquid Software

John H. Hartman, Larry L. Peterson, Andy Bavier
Peter A. Bigot, Patrick Bridges, Brady Montz
Rob Piltz, Todd A Proebsting, and Oliver Spatscheck

TR 97-16

Abstract

Joust is a software platform for liquid software—code that flows easily from machine to machine. Liquid software makes it easier to maintain, debug, update, and customize networked systems. One of the most interesting applications of liquid software is to interject it into the nodes of a network, allowing network functionality, such as routing, to be customized. Additional features, such as special-purpose congestion control and filtering algorithms, are also easily added. The challenge is to develop a communication-oriented platform for liquid software, one in which the focus is the efficient transfer of data, not high-performance computation. To this end we have designed and implemented Joust, which consists of a complete re-implementation of the Java virtual machine (including both the runtime system and a just-in-time compiler), running on the Scout operating system (a configurable, communication-oriented OS). The result is a configurable, high-performance platform for running communication-oriented liquid software. We present the results of implementing three different liquid software applications on Joust, including a prototype architecture for active networks.

December 3, 1997

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Liquid Software—code that easily flows from machine to machine—dramatically changes the way we think about and build networked systems. Systems built using liquid software will be easier to maintain, debug, and update, but perhaps more importantly, they will allow users and applications to customize the underlying services by interjecting code into the system's nodes.

Although most of the recent focus on liquid software considers how machine-independent code can be exploited at the end nodes (hosts) of a networked system—for example, by web browsers—one of the most intriguing possibilities is to interject mobile code into the intermediate nodes (routers) that forward packets through the network. A network that exploits liquid software in this way is often called an *active network* because the network's packet delivery service is programmable instead of fixed [13, 5].

For a system to support liquid software—that is, for a program to run on any node in the network—each node must export a common interface. This interface must be independent of both the node's machine architecture and its operating system. Java provides a good starting point for building such a system [6]. Java bytecode defines an architecture-independent representation of a program, and the Java Virtual Machine (JVM) defines an OS-independent interface for accessing OS/hardware resources.

While Java provides a good foundation for liquid software, the current JVM is limited in both the functionality it provides and the performance it delivers. This is especially true for systems that differ from traditional Java applets in the following two ways: (1) they consist of low-level systems code rather than high-level user applications, and (2) they implement communication-oriented services rather than computation-centric functions. Active network nodes, including application gateways such as firewalls and proxies, are just one example of low-level/communication-oriented systems that can benefit from liquid software; others include network-attached appliances (e.g., cameras, disks, displays) and specialized servers (e.g., file and web servers).

Consider two examples of the current JVM's limitations. First, liquid software running in an active network node or a network-attached player needs fine-grain control over system resources—e.g., CPU cycles and link bandwidth—so that it can schedule those resources to meet QoS guarantees and realtime deadlines. The current JVM does not support such low-level (systems-oriented) access. Second, active network nodes and network-attached appliances also need to support I/O mechanisms that are well-integrated with the underlying OS since their primary task is to move data from one device to another (e.g., from a network device to the frame buffer, from a video capture card to a network device, from an input port to an output port, and so on). It is ironic that Java is designed for networked systems, yet it provides no communication-oriented support beyond the socket interface found on Unix.

This paper describes the design and implementation of a Java-based system that addresses these limitations. The system, called Joust, is specifically designed to support liquid software on low-level, communication-oriented systems, of which active networks are a driving example. To a first approximation, Joust is an implementation of the JVM on top of the Scout operating system [8]. As illustrated in Figure 1, it consists of the underlying Scout OS, a runtime system for the JVM, and a Just-in-Time (JIT) compiler that translates Java bytecodes into native instructions. The system is unique in the way the JVM is integrated with Scout, which serves to address the functionality and performance limitations outlined above.

The three major components of Joust are described in the next three sections: Section 2 gives an overview of Scout, Section 3 describes the runtime system, and Section 4 describes the JIT compiler. Section 5 then describes three demonstration systems we have built using Joust. These systems serve to illustrate how the features of Joust—especially how it is integrated with Scout—can be exploited to support liquid software on low-level, communication-oriented systems, such as active networks.

2 Scout OS

Scout is a configurable OS that includes primitive abstractions to support communication. It is written in C and runs stand-alone on Intel Pentium and Digital Alpha processors. This section gives a brief overview of Scout, and discusses its role in supporting liquid software.

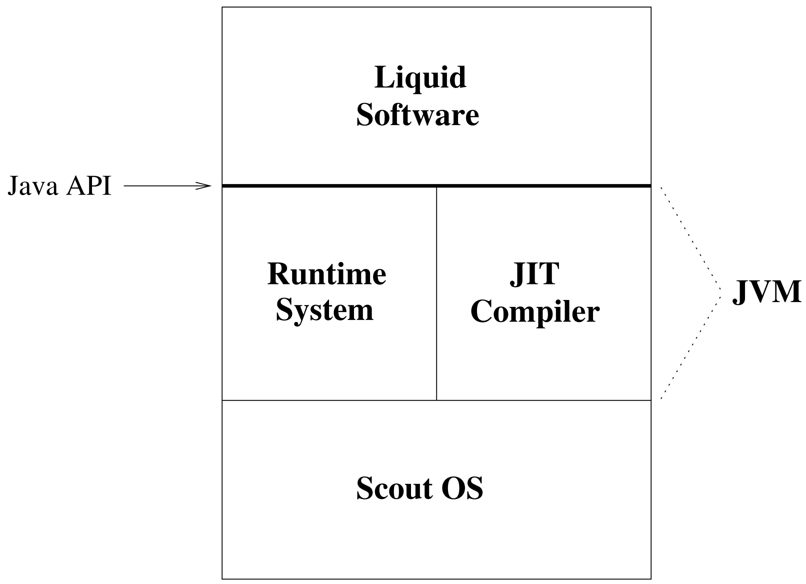

Figure 1: Overview of Joust Architecture

2.1 Configurability

Modules are the unit of program development and configurability in Scout. Each Scout module provides a well-defined and independent functionality. Well-defined means that there is usually either a standard interface specification, or some existing practice that defines the exact functionality of a module. Independent means that each single module provides a useful, self-contained service. That is, the module should not depend on there being other specific modules connected to it. Typical examples are modules that implement networking protocols, such as IP, UDP, or TCP; modules that implement storage system components, such as VFS, UFS, or SCSI; and modules that implement drivers for the various device types in the system.

To form a complete system, individual modules are connected into a *module graph*: the nodes of the graph correspond to the modules included in the system, and the edges denote the dependencies between these modules. Two modules can be connected by an edge if they support a common *service interface*. These interfaces are typed and enforced by Scout. By configuring Scout with different collections of modules, we can configure kernels for different purposes, including network-attached devices, web and file servers, firewalls and routers, and multimedia displays. For example, Figure 2 shows the module graph for a Scout kernel that receives and displays MPEG-compressed video. The configuration includes a device driver for the network card (ETH), two conventional network protocols (IP and UDP), a video-aware transport protocol (MFLOW), a decompression algorithm (MPEG), a window manager (WIMP), and a device driver for the graphics card (VGA).[1] Such a configuration is specified at build time, and a set of configuration tools assemble the corresponding modules into an executable kernel.

2.2 Path Abstraction

Scout adds a communication-oriented abstraction—the *path*—to the configurable system just described. Intuitively, a path can be viewed as a logical channel through a modular system over which I/O data flows. In this way, a path is analogous to a virtual circuit that cuts through the nodes of a packet-switched network; in fact, one can think of a path as a continuation of such a circuit through the host OS. In other words, the path abstraction encapsulates data as it moves through the system, for example, from input device to output device. Each path is an object that encapsulates

[1] KBD is the keyboard device driver, MOUSE is the mouse device driver, and ARP is ARP.

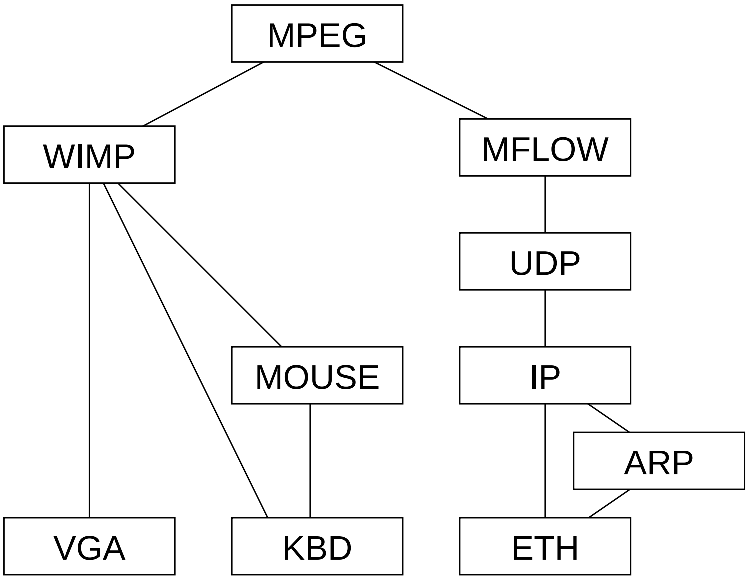

Figure 2: Example Scout Module Graph

two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

Although the module graph is defined at system build time, paths are created and destroyed at run time as I/O connections are opened and closed. Figure 3 schematically depicts a path that traverses the module graph shown in Figure 2; it has a source queue and a sink queue, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries. Operationally, paths come into play at two points in time; we use the video display example to illustrate.

First, at video setup time, a path object is constructed through the module graph. This is accomplished by calling Scout's pathCreate operation, specifying a starting module (in this case, MPEG) and a set of attributes for the new path. The attributes explicitly identify some of the modules the path should pass through (e.g., MFLOW), as well as the IP address of the video server. The MPEG module uses the attributes to choose which module adjoining it in the module graph the path will traverse next. Path creation is then forwarded to that next module. This process repeats until either the edge of the module graph is reached, or the attributes are not specific enough to allow the next module to be unambiguously chosen. In other words, the path is created incrementally, with each module deciding which adjacent module to visit next.

In our video example, MPEG is the starting module for pathCreate. However, the resulting path reaches from MPEG to ETH, but does not span the video device. Path completion is accomplished by extending the previously created path using Scout's pathExtend operation. This operation takes an existing path and a new set of attributes as arguments, and returns an extended version of the path. In our video example, pathExtend completes the path to the display device, as shown in Figure 3.

Second, network packets that arrive for a particular video stream are inserted into the source queue for the corresponding Scout path. Since there may be multiple video paths active in the system at a given time, Scout first *classifies* each incoming packet according to the path to which it belongs. Like path creation, packet classification is also done incrementally, with each module contributing a partial classification function; e.g., the IP module inspects the protnum field in its header. Once enqueued on a path, a thread is scheduled to shepherd this message along the path; this thread inherits its scheduling parameters from the path, as described in the next paragraph. When the thread runs, it executes the sequence of modules associated with the path, and deposits the message in the sink queue. The display device periodically removes frames from the sink queue and displays them.

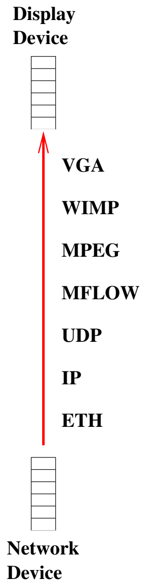

Figure 3: Example Path

Scout uses a two-level scheduling hierarchy: multiple simple schedulers each control their own thread pools, and Scout divides the available CPU cycles among them. Each path is assigned a thread scheduler at path creation time based on the path's attributes; different paths can use different schedulers. For example, a video frame traversing an MPEG path has a realtime deadline representing when it must be in the sink queue and ready to display. We use an Earliest Deadline First (EDF) scheduler for the MPEG path to handle this realtime constraint. Threads belonging to paths that do not have realtime deadlines are scheduled by a priority FIFO scheduler. The EDF scheduler receives sufficient CPU capacity from the system to run its realtime paths, with the remaining cycles going to the priority scheduler.

2.3 Discussion

Like the example shown in Figure 2, Joust is a particular configuration of Scout, one that includes a module that implements the Java Virtual Machine (JVM), along with a collection of modules that implement the underlying services upon which JVM depends (e.g., TCP/IP, NFS, WIMP). This configuration is defined more fully in the next section.

Building Joust on top of Scout has several important consequences. First, Scout's path abstraction is optimized for processing network I/O. It is by integrating this abstraction with JVM—as described in later sections—that we are able to introduce communication-oriented functionality into the Java API. Second, this careful integration between the JVM and Scout results in better performance than one can achieve with an off-the-shelf Java environment, such as JDK running on Solaris or Linux. Finally, unlike Sun's JavaOS—which is a fully integrated system, but one implemented entirely in Java—Joust implements performance-critical components (e.g., the kernel and the protocol stack) in C. These components are a standard part of the liquid software platform, and thus performance is more critical than code mobility.

3 Joust JVM Runtime

The Joust JVM runtime system provides the Java API routines needed by Java programs, and is implemented as a Scout module. The performance of the runtime is a concern because it is the heart of the Joust system. There are three aspects to improving the JVM runtime. First, the platform-independent API routines must be optimized in general.

Second, the JVM must be tailored to take advantage of any platform-specific functionality, in particular the specialized functionality provided by Scout. Third, the Java API itself must be extended so that Java applications can directly take advantage of Scout functionality.

3.1 General Optimizations

Joust's JVM was originally developed for the Toba system [10]. There are several platform-independent issues in the implementation of the JVM, primarily related to differences between the JVM's functionality and that commonly provided by operating systems. First, the JVM implements pre-emptive threads that use monitors and condition variables to synchronize. Java monitors are relatively unique in that they are recursive. A thread may enter the same monitor more than once, without deadlocking. This is in contrast to standard monitor implementations, and prevents Java monitors from being implemented directly using standard lock and unlock primitives. Instead, a Java monitor must be represented as a lock, a reference count, and the identity of the thread that holds the lock (if one exists). A thread enters a monitor by first checking the identity of the thread holding the lock, and doesn't acquire the lock if the entering thread already holds it. The reference count keeps track of how many times the thread has entered the monitor. On exiting a monitor the thread decrements the reference count and releases the lock only if the reference count reaches zero. Similarly, when waiting on a condition variable the thread must release the lock and save the reference count, and reacquire the lock and restore the reference count on re-entering the monitor.

The Joust JVM optimizes the monitor implementation for the single-thread case. We found that many Java applications are single-threaded, and we anticipate this will also be true for liquid software. Omitting the locks in the single-threaded case, and only using the reference counts, significantly improves the JVM performance. Locks are acquired, however, when a second thread is created.

Java exceptions are also a challenge to implement correctly and efficiently. Throwing an exception causes the flow of control to be returned to the closest handler for that exception, which causes the stack to unwind. Joust implements exceptions using setjmp and longjmp. Defining an exception handler invokes setjmp; throwing an exception causes a longjmp to the most recent setjmp. If an exception is caught by a different thread than the one throwing it (as allowed by the JVM), the Joust JVM allows the catching thread to exit any critical section it is in before catching the exception. This limited form of roll-forward prevents the system from deadlocking if the catching thread is inside a critical section when an exception is thrown.

Finally, the JVM provides garbage-collection facilities. The Joust garbage collector is adapted from the Boehm-Demers-Weiser (BDW) conservative garbage collector [1]. This collector considers every register and every word of allocated memory a potential pointer, and considers all memory reachable from these pointers to be in-use. An alternative would be to maintain type information for allocated memory so that pointers can be positively identified, but this requires that all Joust routines used by the JVM manipulate the type information correctly. Using a conservative collector increases the runtime overhead, but reduces the burden on the system developers.

3.2 Scout-Specific Optimizations

The second source of optimizations is to modify the JVM to take advantage of Scout-specific functionality. Figure 4 gives the module graph that implements Joust. The key thing to note is that the entire JVM is implemented in a single Scout module, and this module, in turn, depends on the TCP and UDP modules (to access the network), the NFS module (to access the file system), the DNS module (to resolve hostnames), and the WIMP module (to access the graphics device).

Implementing the Java API on Scout is complicated by the fact that the Java API is very similar to the POSIX API. Implementing the JVM on a POSIX-compliant operating system consists of little more than writing lots of wrapper functions. A Java application has a Process object which represents itself, complete with input, output, and error I/O streams. The java.lang.System class has properties that can be queried and look very much like environment variables. There is a socket interface that uses both reliable byte stream (TCP) and unreliable datagram (UDP) transports. The file system is hierarchical, File objects act much like filenames in the POSIX world, and file I/O is performed by creating a FileInputStream or FileOutputStream with an existing File object. Domain-style Internet hostnames can be resolved to 32-bit IP addresses. Finally, the Java Abstract Window Toolkit (AWT) provides

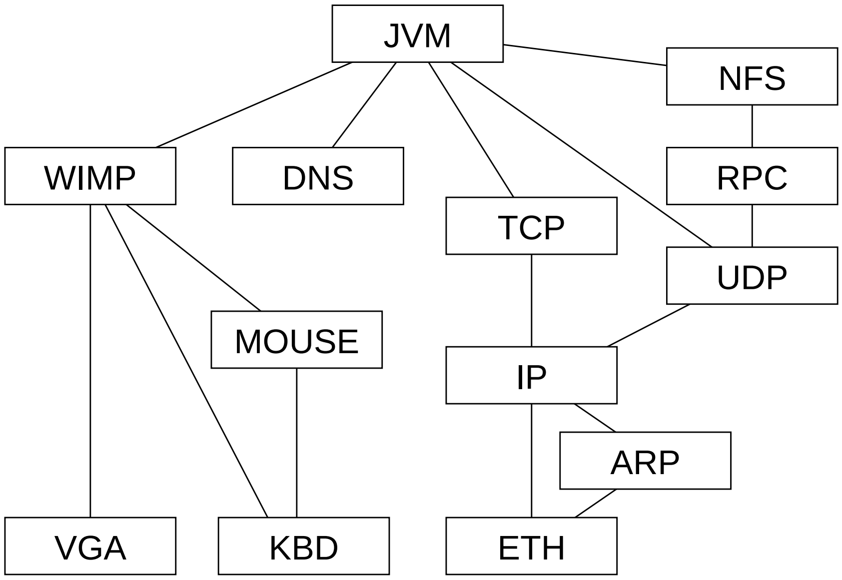

Figure 4: Scout Module Graph for Joust

all the widgets one expects from a graphical user interface: windows and dialogs, buttons, scrollbars, text areas, and canvases for drawing.

Unfortunately, the Scout API is not POSIX-compliant, making it more difficult to implement the Java API. Doing so involves keeping track of a great deal of state information. Some of this state—e.g, the user name and current working directory—are kept in the JVM module. Other state, such as the number of bytes currently available to be read from a socket, as well as the data itself, are maintained in Scout paths. Figure 5 provides a more detailed view of the JVM module and how it interfaces to other Scout modules. The inner boxes (e.g., Net and WinMGR) identify the service interface(s) exported by that module. Some modules—in particular JVM—support multiple interfaces.

Another mismatch between the Java API and the Scout API is that of thread preemption. Java threads are preemptive, whereas Scout threads are not. To rectify this situation the JVM compiler was modified to insert thread yields on backward edges in the control flow graph.

Typically, a Scout path corresponds to a network connection and I/O data flows over it. In general, however, modules need to communicate for other purposes; e.g., to implement control operations. This is done over "control" paths. For example, IP resolves addresses via a single control path to ARP; a separate path is not established for each resolution. These control paths are typically used for less performance critical operations, are created when a module is initialized, and are never destroyed. The JVM makes use of several such control paths.

3.2.1 Sockets

Scout does not support the socket API for transports like TCP and UDP. Both protocols simply export the same I/O interface—called Net—that every other I/O-oriented module in Scout uses. The Scout Net interface differs from the socket API mainly in how connections are established. In Scout, a single call to pathCreate, with the peer address given as an attribute, serves to create the path for the connection. The socket API requires multiple operations—e.g., socket, bind, and connect—to do the same thing. As a result, JVM contains a great deal of code to support the socket-style network connection setup required by the Java API.

Sending data over an existing path is similar to writing data to a socket. Incoming data is handled differently in the two systems, however. The socket interface is synchronous, meaning that an application blocks until data has arrived. In contrast, incoming data is delivered asynchronously to the module that created the path; it is the module's responsibility to buffer this data until it is ready to process it. A final wrinkle is that Scout networking uses a Message

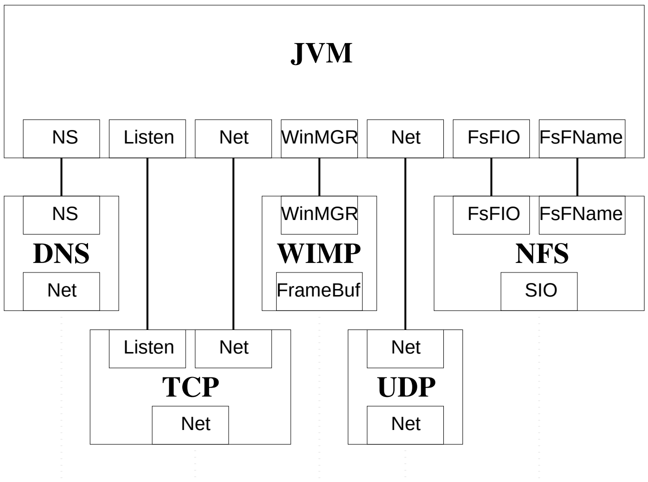

Figure 5: JVM Module Detail

object—a tree of buffers optimized for fragmenting and reassembling messages—while the Java API uses byte arrays as buffers. The JVM module buffers inbound data as Messages, and extracts data out of them as needed.

3.2.2 Name Resolution

Closely related to the issue of sockets is hostname resolution, as provided by the java.net.InetAddress class. Scout lacks gethostbyname and gethostbyaddr. However, it does have a DNS module that provides hostname resolution via the NS interface. To provide name resolution to Java applications, the JVM module creates a control path to DNS during module initialization. This path is then used to resolve hostnames as needed.

3.2.3 File System

The Scout file system functionality is a collection of modules that implements an NFS client. They export two interfaces: one for named operations on a given file (FsFNAME), and one for accessing an open file (FsFIO). The FsFNAME interface has methods for creating a new directory, reading the contents of a directory, getting the attributes of a file, and removing a file or directory. During initialization, the JVM module creates a single control path using this interface. Named file operations simply invoke the proper method on the interface to this path, and returns the results.

To read or write a file in Scout, JVM creates a path to the file via the FsFIO interface, specifying the filename as an attribute. Once the path is created, the JVM can read, write, and seek within a file by simply invoking the proper method. A file is closed by destroying the appropriate path. The JVM module adds little more than a wrapper. As with network connections, the Scout file system uses Message objects for the exchange of data, so the Joust runtime must perform the simple transformations between Java byte array buffers and Messages.

3.2.4 Window System

The Java Abstract Windowing Toolkit is implemented by the classes in the java.awt package. These classes provide Java applications with a platform-independent interface to various windowing toolkits. Unfortunately, WIMP is not a GUI toolkit; it is simply a window manager. It draws geometric shapes, but does not implement widgets, such as buttons. Instead of writing an entire GUI toolkit ourselves, we use the BISS GmbH [9] implementation of the AWT. BISS AWT can be used as a substitute for Java's AWT, or as the toolkit underneath it. Joust uses it as a toolkit so that applications require no modification; they use the standard Sun AWT classes. We therefore only needed to write some wrappers to link the BISS toolkit to WIMP's WinMGR interface.

The interaction between the JVM module and WIMP is via a single control path. This path is created at module initialization time, and is then used by the wrapper functions for all GUI related operations that Java's AWT requests.

3.2.5 Security

Our current experience with Joust is limited to running single Java applications in a single JVM module; this is the case for all the demonstration systems described in Section 5. In a more general scenario, one would like to be able to run multiple, mutually untrusted Java application within Joust. When these applications are independent—i.e., they do not interact within the scope of JVM—security can be enforced by multiply instantiating the JVM module. This can be done in Scout, which boths supports multiple instantiation of modules and the isolation of modules in hardware-enforced protection domains [11].

The more difficult question is how to enforce a security policy for multiple, mutually untrusted Java applications that interact with each other within the JVM. This is the subject of current research, as summarized in [12]. Netscape [3] and Microsoft [2] use object signing to identify the principal responsible for the program, and extended stack introspection to determine the rights of a current thread. These models can easily be used, not only to reveal the rights a current thread possesses, but also to determine the principal that requested those rights. Scout can use this information to determine which paths a particular principal has permission to use, thereby extending resource control and accountability from the JVM module all the way to a device.

3.2.6 Exec Considered Harmful

We have been successful in mapping the POSIX-like Java API to the Scout world and its paths, with one notable exception. The java.lang.Runtime class has exec methods which create new Process objects. Scout does not have the notion of a process, let alone exec'ing a new one. Hence, we can not support this in the Joust runtime. However, this is less of a problem than it first appears. To exec a process, the application needs to know what can be exec'ed on the system. Java provides this information via the queryable properties in java.lang.System. A Java application can ask what operating system and version is underlying the Java VM it is currently running on. It can then use this information to exec another process. However, this makes exec so platform-dependent as to violate our notion of what liquid software is and does, hence we do not consider it a problem that it is not supported in Joust.

3.3 API Extensions

The final optimization to the Joust JVM is to extend the Java API to support Scout-specific functionality. In particular, we extended the Java API to include operations on Scout paths. The pathCreate and pathExtend routines were added to the API, allowing Java applications to create and extend paths between Scout modules. Java applications can thereby move data between Scout modules without actually touching the data. Java applications are also able to modify attributes on Scout paths, thereby changing their behavior.

Although these extensions allow Java applications direct access to Scout functionality, the access is limited. We plan to better integrate Scout paths by further extending the Java API. Currently paths only span Scout modules. Java applications can create and modify these paths, but cannot participate in them directly. Ideally, paths would be able to extend into Java applications, allowing modules on the path to be implemented in Java. Among other things, this requires adding Scout's Message object to the Joust JVM, so that Java applications can directly manipulate Scout messages.

One complication of allowing Scout paths to span Java applications is the interaction between the JVM and the Scout scheduler. Currently, paths that span the JVM module are scheduled by a priority scheduler. This ensures that computation in the JVM does not interfere with any realtime paths. This is particularly important because the JVM includes a garbage collector, whose impact on performance is variable and difficult to predict. If a realtime path is extended into a Java application it becomes difficult to meet the realtime constraints. This is an open research issue.

The current method of manipulating existing Scout paths also needs improvement. A Java application may want to modify the behavior of a path, for example to change the frame rate on a video path. This requires adding a module-specific class to the Java API, so that Java applications may interact with the module in an object-oriented fashion. Currently the Scout infrastructure supports paths composed of Scout modules; it will be extended to support paths that span both Scout modules and Java applications.

3.4 Discussion

The Joust JVM performance must be optimized at several levels. First, the core JVM functionality must be optimized so that potentially slow functions, such as exception handling and synchronization, are made as efficient as possible. Second, the JVM must be modified to make efficient use of the Scout path abstraction. The Java API is POSIX-like, making its implementation on Scout nontrivial. Third, the Java API must be extended to allow Java applications direct access to Scout functionality. This involves exposing the path and message abstraction to the application, so that it can use them effectively. Together, these optimizations result in a high-performance JVM that is specialized for use in a communication-oriented system.

4 Bytecode Compiler

The Joust execution framework is based on the Toba translation system [10]. Toba originally supported a *way-ahead-of-time* (WAT) Java translator, and ran on various Unix systems. We have since ported it to Scout and added a *just-in-time* (JIT) compiler.

The driving concept of Toba is that a large amount of frequently executed Java code is taken from class hierarchies that are used by a variety of programs and are not modified by most programmers. As a result, much can be gained by spending off-line time to translate those classes into a format that can be executed as efficiently as possible. For example, the AWT component of JVM, which is coded in Java, is WAT compiled and linked into Joust.

Toba translates Java class files into C code, which is then compiled with a standard optimizing C compiler to build fast versions of these classes. The translation code is written in Java. The generated C code is a straightforward conversion of the Java bytecode, with each Java method translated into a C function, and C structures used to represent Java classes and instance variables. Toba uses translated versions of the Sun java.* classes to provide most of the standard Java execution environment. The underlying native methods that make up the rest of the execution environment are described in the previous section.

4.1 Just-in-Time Compilation

One of the main limitations of the Toba system has been its main feature: that translation takes place far in advance of execution time. This allows for aggressive optimization, but keeps programmers from using Java's support for dynamic loading and linking of classes at runtime. Since Joust is intended to serve as a network node that handles liquid software, it must be able to receive and integrate code that implements new or improved services, long past the time the kernel is built. We have augmented the basic Toba system with a JIT compiler that translates Java bytecode to the native instruction set of the host machine. Currently, Intel Pentium and SPARC architectures are supported.[2]

Dynamic loading in Java is generally performed by obtaining a byte image of a Java class file from some source, such as a disk file or a network connection. The standard Java interface provides a function, java.lang.Class. defineClass, which translates the byte array into a Java class. The first stage of translation, called *loading*, locates the raw class data and performs minimal sanity checks. The second stage, *linking* or *resolving* the class, must verify that the bytecode is structurally sound (e.g., has no illegal branches), and obtain basic information about referenced classes and methods so that the methods of the class being linked can be executed. This information is stored in a Java java.lang.Class structure, which is made available for use by the running program through routines that look up and create instances of classes by a dynamically specified name. In Toba, this lookup is performed through internal hash tables and use of a dynamic linking loader to get the address of existing class structures in the currently running

[2] The Toba JIT runs in Scout/Pentium (Joust), Linux/Pentium, and Solaris/SPARC.

program. Scout does not support dynamic linking in this form, so the initial kernel includes a list of the addresses of all classes that are implemented within it.

The JIT uses much of the infrastructure of Toba to convert the class file to an internal format more suitable for translation: symbolic references are resolved, instructions are annotated with execution state information, and so on. We begin code generation during the load phase, with a straightforward translation of the Java instructions to native object code. Since we have not performed linking, certain references—for example, to methods in other classes that may not be loaded—are left empty, and a backpatch structure is maintained so the missing addresses can be filled in later. After loading, we have the final physical address of the code for each method, and can use these addresses in the link stage of other classes without running into problems with code that has circular dependencies.

Joust generates the code for all methods of a given class, and its dependents if not already available, at the time the class is defined, rather than waiting until each method is invoked as a true "just-in-time" compiler would. This simplifies the generated code by eliminating runtime checks, and also makes it feasible to project execution time, information that helps in scheduling decisions.

The current implementation of the JIT does a straightforward translation of each bytecode into a sequence of one or more native instructions. We retain the JVM stack-based execution model, except we replace stack operations with references to fixed memory locations. We can do this because Java guarantees that the stack state at any instruction will be the same regardless of the path taken to that instruction. The small number of general-use registers in the Intel architecture limits the value of keeping execution stack elements in registers; a review of the bytecode for several benchmarks indicates that registers would be more effectively used for directly caching local method and instance variables, thereby eliminating the need to place these in the execution stack at all. We plan to add the analyses required to do this, and expect it will decrease both code size and execution time significantly.

4.2 Benchmarks

Even without the optimizations just described, the overall performance of the JIT on a set of microbenchmarks is 2.3 times faster than Sun JDK 1.1.3, and more than half the speed of optimized Toba WAT code. Table 1 shows the performance of several Java implementations on a set of microbenchmarks adapted from the UCSD Java Microbenchmarks [4]. The times are in microseconds, obtained by measuring a given number of repetitions and averaging. All tests were performed on a 200MHz Pentium Pro system; the Unix tests used Linux 2.0.31. We tested two versions of Sun's Java Developer's Kit. Version 1.0.2 is the one with which Toba (and Joust) is compatible, but significant performance improvements in later JDK releases make it appropriate to base our comparisons on JDK 1.1.3. We also compare with the Kaffe system [14], another system which implements just-in-time compilation of Java bytecode. The final three columns compare the Toba/Linux JIT, the Toba/Linux WAT, and Joust.

The microbenchmark names are for the most part descriptive. The value of `setjmp` and `longjmp` for exception handling is indicated by the three-times speedup Toba enjoys over JDK; the Kaffe system has a particularly high overhead for exception handling. Toba's optimization for single-threaded execution is shown in the results for synchronized block and method entry, which are noticeably faster when there is only one thread in the system. The optimization is unfortunately bypassed in the Joust system, where code received over the network is invoked in a separate thread. A somewhat biased inter-system comparison is given in the geometric means of the microbenchmark times when normalized to JDK 1.1.3, which shows Joust performing over two times faster than the Sun implementation.

It should be noted that these microbenchmarks emphasis execution of JIT'ed code. For more complex cases that use standard API class support for AWT and advanced data structures, an even greater speedup is to be expected because that code is way-ahead-of-time compiled by Toba and linked into the Joust kernel where it may called by any JIT'ed method.

4.3 Discussion

The division of labor between Toba's WAT compiler and Joust's JIT compiler improves the performance of Joust kernels by optimizing the fixed components that are needed in a particular system, while still allowing updates and extensions in on-line operation. For example, AWT, and other Java classes that make up the JVM, are WAT-compiled and linked into Joust. The JIT is used to compile only liquid software that is dynamically loaded into Joust.

Benchmark	Reps	jdk102	jdk113	kfe113	jit102	wat102	joust102
null-loop	10^8	0.808	0.191	0.026	0.040	0.015	0.034
add-int	10^7	2.051	0.362	0.042	0.174	0.035	0.171
multiply-int	10^7	2.071	0.352	0.047	0.177	0.042	0.186
add-double	10^7	2.438	0.593	0.136	0.413	0.050	0.277
multiply-double	10^7	2.980	1.140	0.689	0.910	0.602	0.882
array-assign	10^7	1.760	0.334	0.068	0.264	0.057	0.237
instance-var	10^7	1.664	0.418	0.035	0.160	0.035	0.164
method-local	10^7	1.825	0.560	0.124	0.159	0.065	0.156
method-remote	10^7	2.096	0.597	0.123	0.250	0.066	0.248
method-interface	10^7	2.201	0.846	1.906	0.367	0.212	0.350
exception-local	10^6	3.031	1.628	115.511	0.941	0.736	0.681
exception-caller	10^6	3.959	2.353	165.499	1.068	0.784	0.885
exception-remote	10^6	6.909	5.492	165.058	1.942	1.694	1.458
exception-bypass	10^6	23.729	9.481	440.863	2.613	1.528	2.551
sync-block-single	10^6	5.817	1.403	1.052	0.736	0.658	1.141
sync-method-single	10^6	5.398	1.627	1.079	1.021	0.910	1.371
sync-block-multi	10^6	5.819	1.406	1.050	1.136	1.065	1.141
sync-method-multi	10^6	5.398	1.629	1.078	1.406	1.288	1.365
thread-yield	10^5	48.640	42.430	6.470	35.180	36.290	3.500
gmean-norm-excl-excp		0.28	1.00	3.27	1.87	4.36	2.18
gmean-norm		0.32	1.00	1.11	1.99	4.14	2.34

ID	Platform Description
jdk102	Sun JDK version 1.0.2, Linux 2.0.31, 200MHz Pentium Pro
jdk113	Sun JDK version 1.1.3, Linux 2.0.31, 200MHz Pentium Pro
kfe113	Kaffe 0.9.2 on JDK-1.1.3 class files, Linux 2.0.31, 200MHz Pentium Pro
jit102	JIT JDK-1.0.2 class files, Linux 2.0.31, 200MHz Pentium Pro
wat102	Toba -O on JDK-1.0.2 class files, Linux 2.0.31, 200MHz Pentium Pro
joust102	JIT on JDK-1.0.2 class files, Joust, 200MHz Pentium Pro

Table 1: MicroBenchmark Timings (microseconds per repetition)

5 Demonstration Applications

This section describes three applications of liquid software that we have implemented in Joust, and reports measurements of their performance. The demonstrations both serve to illustrate the value of liquid software and to highlight the advantages of Joust's design.

5.1 Active Networks

An active network is an application of liquid software that interjects code into the nodes within a network, thereby giving programmers the ability to customize network services. In an active network, routers not only forward packets, but they may also execute code on behalf of those packets. For example, this code could do sophisticated routing, congestion control management, or packet filtering.

ANTS is an experimental active network architecture written in Java [13]. All ANTS packets, called *capsules*, carry or refer to executable code. This code is installed in ANTS routers in the network as they are traversed by capsules. A capsule's code is responsible for performing almost all computation in the network on that capsule, including routing it through the network to its final destination. This means that the router execution environment needs to be carefully structured and tuned to obtain good performance.

We have constructed an active network node using Joust and the ANTS framework. In our system, the core ANTS architecture—the base system that every ANTS router must run—is translated from Java to C code using the Toba WAT compiler. This C code is then statically linked into the Scout kernel. The Joust JIT compiles only the protocols carried by capsules.

We have measured the performance of our ANTS active network node using two simple test cases. We compared our implementation on Joust to ANTS running on Sun's JDK interpreter, and to ANTS running on UNIX after it had been compiled with Toba. Specifically, we measured two different ANTS protocols (the programs referenced by ANTS capsules):

- **PingPong** bounces a capsule carrying one byte of data between two active nodes. This protocol does minimal computation, and so measures the overhead of sending and receiving capsules. Note that only the first capsule causes the **PingPong** application to be loaded; all subsequent capsules simply carry a reference to that code.
- **RouteQuery** queries a capsule-defined routing table on a remote active node. This computation is more representative of the work a capsule might do in practice.

	JDK/Linux(μs)	Toba/Linux(μs)	Joust(μs)
PingPong	1770	830	518.4
RouteQuery	2510	872	539.2

Table 2: Performance of ANTS on Different Platforms

For both test protocols, measurements were taken on a pair of 200MHz PentiumPro processors connected by a 10Mbps ethernet. The first software platform is Joust, as just described; the second was the same Toba WAT and JIT compilers that Joust uses, but running on Redhat Linux 2.0.23; and the third was Sun JDK 1.0.2 running on the same Linux OS. The results are presented in Table 2. Each individual test involved 10,000 round-trips, and each test was run 10 times.

Both Joust and Toba/Linux are clearly faster than JDK/Linux. Joust takes full advantage of WAT compilation and an underlying C core to achieve high-performance. To determine in more detail the Joust advantage over Toba/Linux, and to determine the incremental costs of the JVM and ANTS, we ran several additional tests. Specifically, we measured simple UDP round-trip latencies on the underlying OS (Scout and Linux), and then UDP latencies from within a Java program. These numbers are then compared to the ANTS **PingPong** results from the previous experiments. The results are summarized in Table 3.

	Joust(μs)	Toba/Linux(μs)	Difference(μs)
ANTS/PingPong	518.4	830	312
Java/UDP	368.2	596	228
OS/UDP	305.4	350	45

Table 3: Performance of Networking Subsystems on Different Platforms

We observe two things from these results. First, the Java/UDP round trip latencies are much smaller on Joust than on Linux/Toba, even though the OS/UDP latencies in Scout are only moderately faster than in Linux. This is due to the difference in how well the JVM networking is integrated with the underlying OS. Second, focusing on the Joust numbers, we see that the JVM adds 122μs to the round trip latency, and the ANTS infrastructure adds another 219μs to the round-trip times.

5.2 NetTV

NetTV is a Java application that decodes and displays MPEG-compressed video streams sent over a network. It runs on an end system, but it is similar to the ANTS example in that it is both low-level and communication-oriented. In particular, NetTV requires realtime scheduling and the ability to efficiently move data from the network device to the display device.

NetTV is based on the module graph used as an example in Section 2, but also involves a Java application that implements the user interface and controls the Scout path that receives, decodes, and displays the video. In other words, the Scout configuration required by NetTV corresponds to the union of the module graphs given in Figures 2 and 5, with NetTV dynamically loaded into the JVM module.

To use NetTV, the user selects a video from a menu. NetTV then establishes a Scout path to receive, process, and display the video stream. NetTV includes a graphical user interface that allows the user to interactively control the video, for example, to adjust its frame rate and quality. NetTV does not, however, directly process the video frames; this processing is done by the MPEG path that traverses modules contained in the underlying Scout kernel. NetTV accesses this underlying Scout facility through a Scout class extension to the standard Java API. The following outlines in more detail how NetTV interacts with Scout.

- NetTV first uses the Java AWT to create a hierarchy of nested windows, including a parent window, a subwindow that displays the video, and additional subwindows that implement various control buttons. AWT invokes WIMP, which returns a unique identifier for each window it creates.

- NetTV creates a Scout path starting at the MPEG module and exending to the display device. This is done via the extended API, which in turn invokes the native pathCreate primitive. The identifier of the display subwindow returned in the first step is included as a parameter to this call; this tells Scout that the path should terminate at this window.

- NetTV calls the Scout API again to extend this path from MPEG to the network device; this corresponds to a call to the native pathExtend routine. As parameters to this call, NetTV supplies the video server's IP address and a well-known UDP port number from which the server sends videos. After path extension completes, NetTV retrieves the local UDP port number assigned to the path.

- NetTV sends a REQUEST message to the video server containing the name of the video and the local UDP port. This message is sent using Java's standard socket method. The server sends back a REPLY message acknowledging that the video exists, and then begins transmitting frames.

- As the video arrives, Scout recognizes the frames as belonging to the path it just created for NetTV, and enqueues them in that path's source queue. The path executes as described in Section 2, causing the frames to traverse the path and eventually be displayed.

The user sees the display illustrated in Figure 6, where video shown in the inner display window (the swimming turtle) is managed by the Scout MPEG path, while the outer window and the buttons are managed by the Java NetTV application. WIMP supports hierarchical windows, which gives the user the impression of a single graphical application.

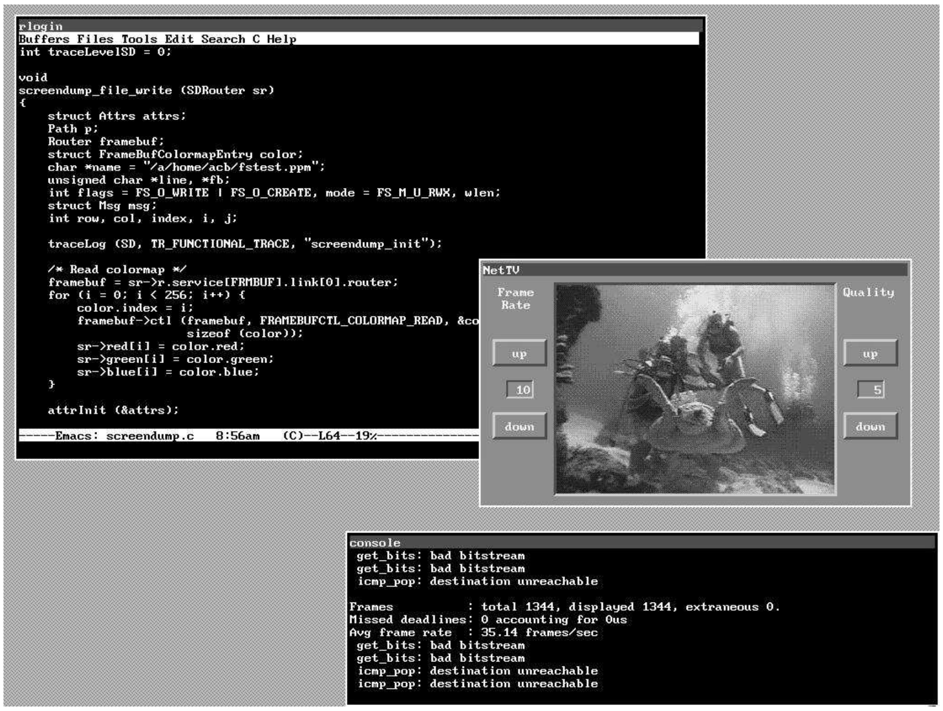

Figure 6: Example NetTV Display

As the video plays, the user can interact with NetTV's GUI to change visual aspects of the video. NetTV uses the Scout extended API to communicate the user's input to the path that carries the video. Changing a video's attributes affects how resources are allocated to the video path, for example, the path may require more CPU cycles if the user wishes to increase the frame rate. However, it is up to the path to directly negotiate with the system for additional resources; NetTV simply sets the parameters used by the path.

Video	# of frames	max. rate [fps]	
		Scout	Linux
Flower	150	44.7	37.1
Neptune	1345	49.9	39.2
RedsNightmare	1210	67.1	55.5
Canyon	1758	245.9	183.3

Table 4: Coarse Grain Comparison of Scout and Linux

NetTV illustrates two advantages of being able to establish a data channel outside of the Java VM. First, NetTV is able to achieve excellent MPEG performance. Table 4 compares the maximum frame rates for videos played on Scout and Linux/X11 systems; we observe that Scout consistently outperforms a Linux-based MPEG decoder by 20-34% [7]. We know of no comparable numbers for an MPEG decoder written in Java. Second, separating the MPEG path from the JVM makes it possible to schedule the two entities separately, and in fact, allows different schedulers to be used for each. The MPEG path, because it has realtime constraints, is scheduled in Scout using the EDF scheduler. The JVM, however, is scheduled by a priority scheduler. Scout ensures that the EDF scheduler is given sufficient CPU capacity to run the realtime paths, with any excess time given to the priority scheduler. This also means that Java's garbage collector does not interfere with the timely execution of the MPEG path.

One can imagine wide applicability of a mechanism that allows Java programs to establish and control data channels outside the JVM. On an Active Network node, for example, this facility might be used to do cut through: a dynamically loadable protocol is involved in establishing a flow—e.g., negotiating its QoS contract—but is otherwise not involved in forwarding packets from input port to output port. This strategy allows one to implement each service in the most efficient way—in the case of NetTV, by C code, but in general, perhaps by specialized hardware (e.g., a high-performance switching fabric). The key is to provide a general enough interface to support such data channels without sacrificing the platform independence of Java. In Joust, this interface allows the Java program to specify high-level attributes (invariants) for the path.

5.3 MPEG Filter

The final application is a filter that one might inject into the middle of a network connection. This application could run on a router that connects a high-speed backbone link to a low-speed last-hop link. While this filter could have been written for the ANTS architecture, for this demonstration we wrote the application in Linux and dropped it unchanged into Joust.

The purpose of the filter is to intelligently massage an MPEG-encoded video stream so it can be transmitted over a low-speed link. This can be done in a variety of different ways, depending on circumstances; e.g., difference in link speeds, the frame types used by the video, and so on. This is precisely why this filter is a good candidate for liquid software—a video application can choose the most appropriate strategy for the situation.

In our case, we wrote a simple filter that removes **B** frames from the video stream, and forwards only **I** and **P** frames. This filter, running on Joust, was able to satisfy the realtime requirements for the video streams we measured. The filter was also effective: it reduced the bandwidth required by one sample video from 223KBps to 152KBps.

6 Concluding Remarks

Joust is a platform for liquid software targeted at low-level, communication-oriented systems like active networks. It achieves good performance and rich functionality by applying two general design principles. First, the fixed components of Joust are either written in C (e.g., Scout kernel and modules), or written in Java but WAT compiled (e.g., ANTS and AWT). The JIT is applied only to portions of the system that must be dynamically loaded. Second, the JVM is carefully integrated with the underlying OS facilities. This includes extending the JVM to allow liquid software to take advantage of Scout's communication-oriented path abstraction.

The impact of these two design principles is demonstrated by three applications of liquid software: the ANTS active network architecture, a network appliance that decodes and displays MPEG video, and an MPEG filter. There are two results of particular note. First, Joust is two to three times faster than an off-the-shelf Java system (Sun's JDK running on Linux) for all the benchmarks and applications we tried. Second, allowing liquid software to establish and control external data channels—e.g., Scout paths—seems like a very powerful idea that has wide applicability.

Acknowledgments

We would also like to thank the other members of the Scout and Sumatra groups. This work supported in part by DARPA contracts DABT63-95-C-0075 and N66001-96-8518, and NSF grant NCR-9204393.

References

[1] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practive & Experience*, pages 807–820, Sept. 1988.

[2] M. Corporation. Microsoft security management architecture white paper. URL: `http://www.microsoft.com/security/ie4security.htm`, 1997.

[3] N. C. Corporation. Netscape devedge online documentation. URL: `http://developer.netscape.com/library/documentation/`, 1997.

[4] W. G. Griswold and P. S. Phillips. Microbenchmarks for Java. URL: `http://www-cse.uscd.edu/wgg/JavaProf/javaprof.html`, 1996.

[5] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A programming language for active networks. URL: `http://www.cis.upenn.edu/~switchware/papers/plan.ps`.

[6] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[7] D. Mosberger. *Scout: A path-based operating system*. PhD thesis, University of Arizona, 1997.

[8] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 73153–168, Oct. 1996.

[9] B. G. H. Page. Biss GmbH. URL: `http://www.biss-net.com/`, 1997.

[10] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications—a way ahead of time (WAT) compiler. Technical Report 97–01, Department of Computer Science, The University of Arizona, 1997.

[11] O. Spatscheck and L. Peterson. Escort: Scout security architecture. Technical Report TR97-17, The Department of Computer Science, University of Arizona, Tucson, Arizona, Dec. 1997.

[12] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 116–128, Saint Malo, France, Oct. 1997.

[13] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH 98*, San Francisco, CA, Apr. 1998.

[14] T. Wilkinson and Associates. KAFFE: A free virtual machine to run Java code. URL: http://www.kaffe.org, 1997.