PicoThreads: Lightweight Threads in Java

Andrew Begel, Josh MacDonald, Michael Shilman
[abegel, jmacd, michaels]@cs.berkeley.edu

Department of Electrical Engineering & Computer Sciences
University of California, Berkeley, CA 94720

Abstract

High-performance, I/O-intensive applications often require complicated, split-phase, event-based implementations. Threads appear to be an attractive alternative because they allow the programmer to write a single sequence of operations and ignore the points at which the execution may be blocked. Unfortunately, the typical amount of memory required to support this technique prevents applications from scaling to large numbers of threads.

Rather than relying on event-based programming, we use a program transformation to provide a low-cost thread implementation. This transformation retains the advantages of user-scheduled, event-based programs, yet efficiently supports large numbers of threads. We replace the standard Java thread mechanism with lightweight PicoThreads and a cooperative, event-based, user-level scheduler.

This is accompanied by a PicoThread-aware, asynchronous network library. To evaluate our implementation, we measure the performance of a simple web crawler running thousands of threads. Our results show that PicoThreaded programs perform comparably or substantially better than threaded programs when using large numbers of threads.

1 Introduction

Threads evolved out of the operating system community as a general-purpose solution for implementing concurrent systems. Concurrency is often used by the operating system to efficiently overlap computation with concurrent, blocking requests. Threads, however, are not ideally suited for this purpose. Indeed, Ousterhout suggests that threads are seldom a good idea and proposes event-based programming as a better alternative [O96].

1.1 Event-Based Programming vs. Thread Programming

Event-driven or event-based programs are those which handle these situations by registering handlers to respond to various events. In this technique, code to handle a blocking call first initiates an asynchronous request, and then registers a callback method to complete the computation when the request has completed. This implementation strategy does not require threads or synchronization (on a uni-processor), and leads to efficient, user-schedulable programs with little support from the operating system. Event-based programs are popular for implementing graphical user interfaces and are also used to handle large-scale network services where events correspond to the completion of an I/O request or the arrival of a new connection.

While the advantages seem clear, event-based programming is not always the right answer. Threaded programming allows the programmer to write a logically sequential program, whereas event-based programs require the programmer to carefully divide each sequence of operations into non-blocking handlers that maintain persistent state. What began as a simple block of code degenerates into a much larger body of code which is often difficult to read, write and maintain. System threads also provide the illusion of concurrency on a uniprocessor[1].

However, the reason that event-based programming is popular is because it solves a large class of problems which do not require true concurrency. Using threads to handle overlapping, blocking requests is often more complicated and costly than necessary. In particular, synchronization, locking and preemptive scheduling are not always needed, but the thread programmer must accept all three and pay the cost in debugging difficulty as well. For a program running

1. Or actual concurrency when multiple processors are available

on a uniprocessor that expects to handle requests quickly (or yield cooperatively), these burdens are avoidable.

Assuming one is prepared to deal with synchronization, threads are still troublesome, as their memory footprint can be quite large. For instance, the default thread created by *pthread_create* in Solaris 2.5.1 reserves one megabyte of virtual address space for its stack and maps a minimum of one page (eight kilobytes) of virtual memory. An application which would like to use many threads suffers from the fact that memory is allocated for the worst case, even though a typical thread is not likely to use its full allotment.

Even if there is enough memory, the application's paging behavior is likely to suffer. An application with 5000 threads uses 40 megabytes just for stack space and requires more than 32 bits of address space! As a compromise, the programmer can use a smaller group of threads and map them onto a much larger pool of tasks to be completed. This approach is not ideal because it forces the programmer to compromise the simple programming model of one thread per task.

In this paper, we will examine the requirements of high-performance, network-intensive applications such as web crawlers or web servers. These programs attempt to execute large numbers of mostly unrelated tasks as quickly and as efficiently as possible.

Since a 100 Mb/s network is able to sustain 3000 simultaneous 28.8 Kb/s connections, we would like our application to be capable of running 3000 threads, one thread per connection. This one-thread-per-task approach is appealing because it allows each session to be modeled within a single flow of control.

This paper addresses the problems with large-scale threaded programming and the dichotomy between threaded and event-based techniques. Our solution is the *PicoThread*, a lightweight, user-level Java thread that can be cooperatively-scheduled, dispatched and suspended. PicoThreads are implemented in the Java bytecode language [AG97] via a Java class-to-class translation.

Whereas the event-based program was forced to register a callback and yield control to the event loop, our translation produces threaded programs that yield control and a *continuation* sufficient to restart the thread where it left off. This allows the programmer to write in the convenient, sequential style offered by threads and retain the efficient, low-memory characteristics of event-based programs. In addition, we implemented an asynchronous, PicoThread-compatible network library and wrote a web crawler in order to evaluate our system.

1.2 The remainder of this paper

In the next section, we describe continuation and provide a context and mechanism for our Java class transformation. In the third section, we describe the PicoThread runtime system which is made up of the scheduler and the network library. Fourth, we introduce Piranha, our massively multithreaded web crawler and compare it to an alternative Java threaded version. In the fifth section, we evaluate our code transformation and speculate on possible future work. Finally, we present our conclusions.

2 Continuations and CPS Conversion

2.1 Continuations

A continuation is a return address for a procedure. This is commonly passed on the stack during function calls in order for procedures to know where to return. Most user programs do not directly manipulate continuations. Rather, they are used as a simple all-purpose mechanism for programming language compilers to implement standard iterative and exceptional constructs [FWH92]. In a program that has been converted to explicit continuation-passing style (CPS), each code block takes a continuation which says where to send the return result.

Continuations can also be used in more sophisticated ways to manage control flow. For example, multithreading can be implemented by substituting a scheduler continuation in every normal procedure call, allowing the run-time system to control the execution of a thread at a fine granularity. Instead of continuing normal control flow, the scheduler can save the continuation for rescheduling at a later time.

2.2 Continuations in Java

Java byte code can be modified to use explicit continuation-passing, which allows us to implement user-level multithreading. A thread's stack usage can be calculated statically at compile-time, so compiler-generated continuations need only save the active stack and local variables. This calculation of stack and variable usage is not available to Java threads because they are created at runtime. By CPS converting Java byte code, we are able to reduce the runtime memory requirement incurred by mulithreaded programs.

2.3 Implementation of CPS Conversion

2.3.1 Continuations

A PicoThread continuation is a Java object which contains a reference to the object and method in which it was created. Since Java's procedure call stacks do not have dynamic extent, continuations must also contain extra state to store a method's local variables. PicoThread continuations extend Java exceptions, so that we may take advantage of Java's zero-cost exception mechanism to pass continuations from method to method.

2.3.2 CPS Conversion

The traditional way to implement continuation-passing is to generate continuations on every procedure call [FHW92]. We tried this in an initial version but found that this style leads to an inefficient implementation in the JVM [LK97] [MD97]. On every procedure call, we had to package up the local state and restore it when the call returned.

As an optimization, continuations are not passed during normal execution. Instead, they are constructed lazily. In the event of a context switch, continuations are thrown up the call tree, saving the state in each frame. This method has a negligible effect on the execution of code where continuations are not used, leaving the common case of procedure invocation to be just as fast as Java[1].

In Figure 1, when method C wants to context switch, it creates a continuation (k0) and throws it back to its caller, method B. B wraps up its local state in a new continuation (k1) and throws it back to A. A wraps up

1. The overhead amounts to an extra argument, several local variables, two instructions at the beginning of the code segment, and one instruction to push the new argument at each call to a method

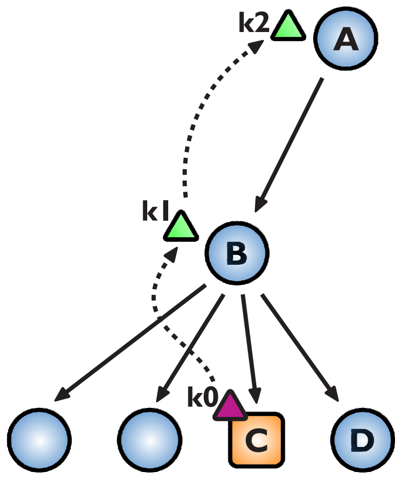

Figure 1. Continuation Passing in PicoThreads

its local state and throws the new continuation (k2) back to the scheduler.

When a continuation is rescheduled, each method in the control pathway is restarted and the local state restored. Upon reaching the last method call, the method that caused the context switch now returns without exception and normal execution of Java code resumes.

2.3.3 Deficiencies in the CPS Conversion Algorithm

The above translation has some deficiencies. Java's adherence to strict stack discipline causes several difficulties in our code translation. First, the algorithm does not save the stack during a context switch. Values left on the stack after the method invocation are not accessible from the exception region, and thus must be left behind. The cost of saving the stack in local variables has not been evaluated. However, this would slow down the execution of a method whether or not continuations were needed. In our web crawler application, this was not an issue due to its sequential code structure.

Second, it cannot capture a continuation while control is inside the body of a constructor or between the *new* and *invokespecial* instructions used by the JVM to construct an object. This restriction means that no call whose value is used as the argument to a *new* expression may capture a continuation. Various techniques can be applied to compile out these untranslatable bytecode sequences.

Finally, the jump-to-subroutine (*jsr*) opcode pushes a ReturnAddress on the stack. The JVM provides a bytecode to store this value in a register, but no way to read it without the return (*ret*) instruction. Since this register cannot be read, our translation cannot capture a continuation inside a subroutine block. This restriction could be removed by replacing *jsr* and *ret* with equivalent *goto* and *tableswitch* instructions.

Yet another difficulty in CPS conversion is deciding which methods and procedure calls to translate. Our translation does not transform any method in the standard Java classes. Most of these methods are not related to I/O and do not block execution of the thread. However, the standard Thread and I/O classes must be rewritten to be PicoThread-aware in order to be used in a PicoThreaded program. If not, a blocking call will suspend the sole Java thread in which all of the PicoThreads are running.

3 PicoThreads Runtime System

The PicoThreads implementation includes three components: a thread class, a scheduler and a network library. Each piece is engineered to support continuation-passing and capture. The scheduler uses these continuations to manage control flow and poll for network events. The threads are used to manage groups of continuations and present a simple abstraction for user code. The last part of this section walks through a typical execution path for a networked application.

3.1 PicoThreads

PicoThread is a layer above continuations that presents a Java-like thread abstraction to the user. The user writes their code using the standard Java thread class. The first stage of the CPS conversion algorithm changes all references from java.lang.Thread to pico.PicoThread. All programs that wish to run threads implement Runnable (which is converted to PicoRunnable).

3.2 Scheduler

In our system, a round-robin scheduler processes continuations from a queue. When a PicoThread begins, it puts a continuation on the queue. When the scheduler dequeues it, it calls the run() method of the associated PicoRunnable object. The scheduler wraps this run() call in a top-level continuation exception handler. Whenever a method throws a continuation it will eventually be caught by this handler and enqueued.

3.3 Asynchronous Network Library

The PicoThread network library replaces the Socket, ServerSocket, SocketInputStream and SocketOutputStream classes from Java. There are four types of blocking calls in these classes: connect, accept, read and write. Each blocking call is replaced by a split-phase call. The first phase initiates an asynchronous request on the network and returns immediately by throwing a continuation to the caller. When the network is polled, if any of these asynchronous calls can be completed, the data is made available and the library enqueues the blocking PicoThread with the scheduler.

In Java, DNS lookup is an operation that blocks the calling thread. In order to make this asynchronous as well, we found a non-blocking DNS library on the net and incorporated it into our code [R92].

We had to modify the structure of the user code during our translation in order to avoid context-switching while values were left on the stack. In addition, the Java Socket constructor performs a connect to the port before it returns. Since we cannot capture continuations inside constructors, we rewrite the code to include a separate call to connect after the new Socket call is complete.

To achieve high performance, we buffer our Socket input stream. We have found that varying buffer sizes between 2-50 kilobytes does not seem to have any effect on network throughput. If there is no data in the buffer when a read call occurs, the read call throws a continuation. However, if the buffer has enough data in it to fulfill the request, then the data is returned immediately via normal function return.

3.4 Walkthrough of a Network Call

Figure 2 shows a walkthrough of a network call through our system. First, the scheduler starts a new PicoThread (1). This thread calls into user code which then makes a call into the network library. The library will take some time to read data from the network (2), so it throws a continuation signalling that it would like to yield. Code inserted by the CPS conversion handles the thrown continuation (3) from the library. It saves the procedure’s local state into a new continu-

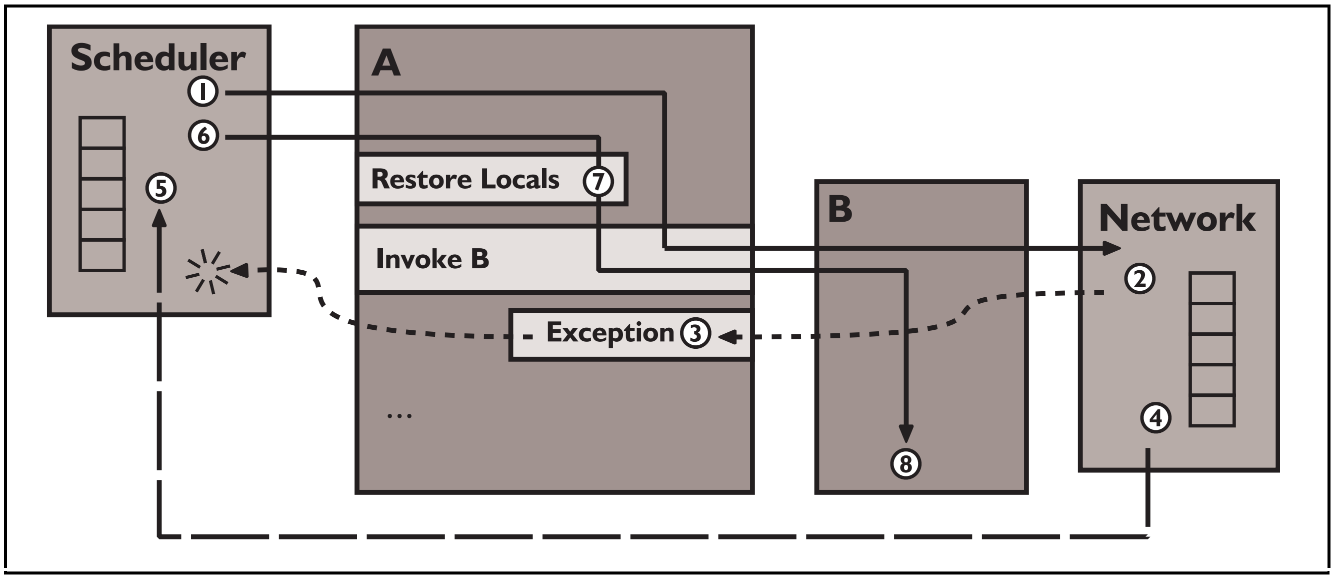

Figure 2. A Walkthrough of a PicoThreaded Network Call

ation, which is linked to the network's continuation and is thrown up to the procedure's caller. Eventually a continuation is caught and stored by the scheduler. When the network library has received its data, it reschedules the continuation with the scheduler (4-5). The continuation is restarted and reinvokes the original procedure (6). More code inserted by the CPS conversion uses the continuation to unpack the method's local state (7) and reinvoke the suspended network call. Since the data is now available, the library can return its results immediately to the calling procedure. Finally, control is returned to the user code (8).

4 Piranha: A PicoThreaded Web Crawler

Piranha is a web crawler which was written to illustrate the programming and performance advantage of PicoThreads over Java system threads. A web crawler is conceptually a simple program. Starting from a root web page, the program reads all the links and recurses on the new pages. Coding this with threads leads to a simple implementation.

This implementation spawns a new thread for every URL to be processed, traversing web pages as fast as it can. Given the tree-like nature of the web, this program quickly creates a large number of threads. In fact, starting from Yahoo, one can reach around 30,000 links in no time at all.

```
import java.net.*;

public class Piranha implements
Runnable {

  URL url;

  public Piranha(URL u) {
    url = u;
  }

  public void run() {
    for(Enumeration e = getLinks(url);
          e.hasMoreElements(); ) {
      URL u = (URL) e.nextElement();
      Piranha p = new Piranha(u);
      new Thread(p).start();
    }
  }
}
```

4.1 Thread Pools

Any experienced Java programmer will look at this program and declare it infeasible. Given the current performance of Java threads, they are right. Threads are slow to create and use too much memory. Thread creation and context-switching are known to be slow in many implementations of the JVM [B97]. Creating a new thread per URL causes not just one object to be created, but all related local data structures as well. Since Java threads use both a Java stack and native C thread stack, each would contain at least one page of stack space. Crawling through 30,000 links, with one thread per link, would run through 240 MB of memory. Java, a garbage-collected language, would be forced to GC constantly. In our tests, we found that the garbage collector was forced to recover 500 KB five times every second.

A careful Java programmer has a keen eye for reusing objects and threads. Object reuse is popular in other garbage-collected languages, when programmers need to make sure that no memory is consed during execution. This leads to a cramped, ad-hoc style of program that we call thread pools. In order to evaluate this technique, we wrote a web crawler which preallocates all memory buffers and creates a constant number of reusable threads. If the URLs in the queue are consumed more slowly than new ones are generated, extra ones are discarded.

We ran this version with varying numbers of Java threads in Java 1.1.5 under Solaris 2.5.1on an UltraSPARC 1 to test its performance.

We can see here in Figure 3 that our throughput maxes out with three Java threads. We get better network bandwidth for larger files, which is to be expected because we incur less system overhead. We only ran up to 32 threads because our performance dropped considerably with more than 8 threads.

4.2 One Thread per URL

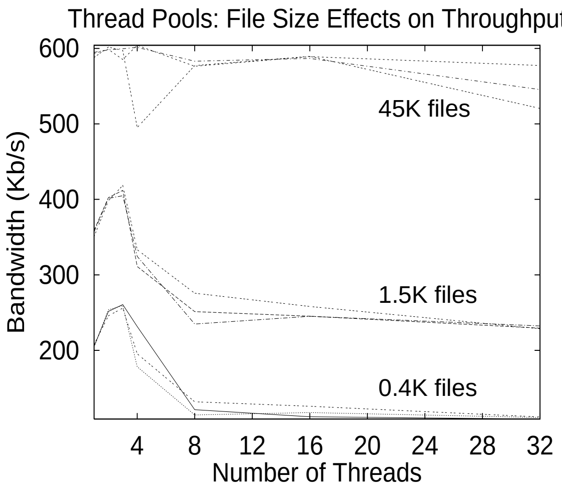

Figure 3. This graph shows the performance of the thread pools web crawler.

We ran the simple web crawler with Java threads and PicoThreads in Java 1.1.5 under Solaris 2.5.1 on an UltraSPARC 1. We ran two experiments, one on a controlled set of web servers in a local cluster, and the other on Yahoo.

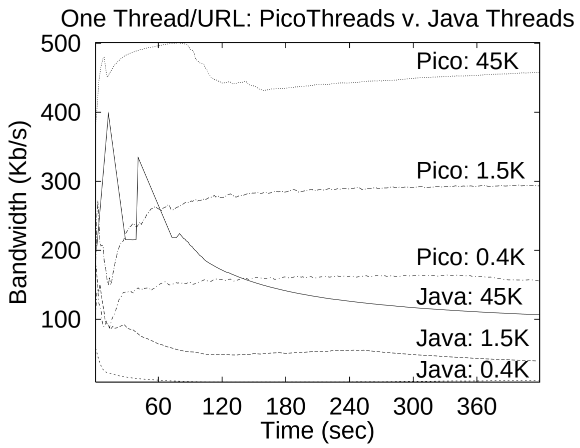

Figure 4. This graph shows the performance difference beteween a PicoThreaded web crawler and a Java threaded version.

This graph is much more illuminating. We see how a PicoThread implementation of the simple multi-threaded web crawler does against an equivalent implementation that uses Java threads. For each size file, PicoThreads get three to fifteen times the bandwidth of the Java threads. We speculate this is mostly due to our cooperative, user-level context-switching.

Since a PicoThread only context switches when a network call needs to block, once the read buffer has been filled, the PicoThread gets to process a large amount of data before yielding control. Java uses preemptive scheduling and more evenly distributes the execution time of each thread during parsing of the HTML files. This leads to slower throughput overall. In addition, the Java threaded version uses more memory per thread than the PicoThread version, thus exhibiting bad paging and cache behavior.

PicoThread performance only achieved a maximum of 500 Kb/sec, while our optimized thread pool version using Java threads achieved 600 Kb/s. We analyzed the execution status of our PicoThreads whenever they did a state change and found that our scheduler and network read/write/connect time was only using 5-10% of the total time. This means that 90% of the time was spent in Java user code parsing URLs. Under a JIT compiler or faster VM, our CPU-bound code could easily perform better. We also do not worry about minimizing memory allocation. We speculate that our web crawler performance would improve if we took better care to reuse memory structures in between thread lifetimes.

5 Evaluation

5.1 CPS Conversion

Traditional CPS conversion algorithms use heap-allocated stacks when they know that a block of code will be returning a continuation. Since Java never heap-allocates its stack, our code transformation had to insert bytecode to save the stack during a context switch. We had to perform data flow analysis to determine the types of the objects on the stack and in the local variables in order to meet the JVM's type-safety constraints. In other words, this made code conversion more difficult in Java than in other languages. This difficulty has also been noted by Odersky and Wadler [OW97].

The design of the CPS conversion was also an issue. Traditionally, each CPS-converted method should receive a continuation and pass one at every function call. However, given that the stack and local variables must be saved and restored at every method call boundary, this leads to extra work for each procedure call. This is contrary to the intent that continuation-passing be a lightweight operation [SS76] [S77].

Since this package is only intended to increase performance for split-phase, I/O-intensive applications, it is only necessary to context switch during the slow, but infrequent, network operations. Therefore, we altered the traditional conversion technique to create continuations not on every method call, but lazily, after a context switch has been initiated.

5.2 Scheduling Policy

There has been much discussion in recent years about exposing scheduling policy to the user [ABLL92]. We experimented with three different schedulers for our web-crawler: FIFO, LIFO, and random-distribution. Since the birth-rate of threads greatly exceeded the death-rate, FIFO and LIFO scheduling policies produced approximately breadth-first and depth-first searches of the web, respectively. Due to I/O time-outs, the real problem with large numbers of threads is the scheduling interval, not the scheduling policy. We observed that the scheduling interval for 1000 threads[1] was as high as 10 seconds. Clearly, Java's performance must improve before larger numbers of threads become feasible. Our goal of saturating a 100 Mb/s ethernet cannot be realized with today's Java implementations on contemporary processors.

5.3 Java Security Implications

One important concern is whether the translation compromises the Java security model. The translated code cannot allow unprivileged access to private methods or local state. Since our translation does not alter the access flags of any method, field, or class, we only have to insure that it is impossible for an unprivileged agent to read, write, copy, or replay a continuation. We can guard against read and write attacks by making continuations private inner classes. We can prevent copy and replay attacks with suitable protection of the runtime PicoThread library. These security claims assume that our translation is trusted, and that no further alterations are made to the class file.

5.4 Future Work

We were unable to find a Java implementation fast enough to properly exercise our web crawler. Current implementations leave it CPU-bound, regardless of the threading model. Perhaps a better test would be to have many threads read files over the network without parsing them.

6 Conclusion

This project shows that it is possible to write highly-parallel threaded programs that are as efficient as custom, split-phase, event-based programs. However, Java's threading model and overall performance must improve before this approach can be used effectively. Cooperative multithreading with PicoThreads is like event-based programming because it does not require synchronization or locking, and leads to efficient implementations without operating system support. It is better than event-based programming because it gives the programmer a convenient sequential programming model. Operating system threads should only be used when true concurrency is required. Thus, cooperative multithreading is the best solution

1. Solaris 2.5.1 imposes a hard-limit of 1024 file descriptors per process.

when the principle requirement is to overlap computation with blocking I/O requests.

Acknowledgments

We would like to thank the authors of the various software we incorporated in this project. In particular, we thank Arthur Do for the HTML Tokenizer, Darren Reed for the AR Library for asynchronous name resolution, and Jon Meyer for the Jasmine Java bytecode assembler.

References

[ABLL92] T. Anderson, B. Bershad, E. Lazowska, H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. ACM Transactions on Computer Systems, 10(1):53-79, February 1992.

[AG97] K. Arnold and J. Gosling. The Java Programming Language. Addison Wesley. 1997.

[B97] D. Bell. Make Java Fast: Optimize! JavaWorld, April 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html

[FHW92] D. Friedman, C. Haynes, M. Wand. Essentials of Programming Languages. MIT Press/McGraw Hill, 1992.

[LK97] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison Wesley. 1997.

[MD97] J. Meyer and T. Downing. Java Virtual Machine. O'Reilley and Associates. March 1997.

[O96] J. K. Ousterhout. Why Threads are a Bad Idea (For Most Purposes). Invited Talk, USENIX Technical Conference. 1996. Slides at http://www.sunlabs.com/~ouster/threads.ps.

[OW97] M. Odesky and P. Wadler. Pizza into Java: Translating theory into practice. POPL. 1997.

[R92] D. Reed. BIND Contributed Code Asynchronous Resolver Library. ftp://ftp.isc.org/isc/bind/src/cur/bind-8/bind-contrib.tar.gz.

[SS76] G. Steele Jr. and G. Sussman. LAMBDA: The Ultimate Imperative. AI Lab Memo #353. MIT. Cambridge, 1976.

[S77] G. Steele Jr. LAMBDA: The Ultimate GOTO. AI Lab Memo #443. MIT, Cambridge, 1977.