An Empirical Study of Technologies to Implement Servers in Java

by

Brian Charles Demsky

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

© Brian Charles Demsky, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
May 11, 2001

Certified by . . . .
Martin Rinard
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

BARKER
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
JUL 11 2001
LIBRARIES

An Empirical Study of Technologies to Implement Servers in Java

by

Brian Charles Demsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2001, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science and Engineering

Abstract

Over the past few years, mainstream computing has shifted from isolated personal computers to networks of computational devices. As a result the client-server programming model has become increasingly important. These servers often have very strong performance and reliability criteria.

We did an empirical study of different server architectures, developed a novel thread-per-connection to event-driven transformation, and developed a performance model to predict server performance under various server architectures.

We considered four different server architectures: a thread-per-connection architecture built on a kernel-level thread implementation, a thread-per-connection architecture built on a user-level thread implementation, an automatically generated event-driven architecture, and a thread pooled architecture.

Our empirical study consisted of evaluating the different server architectures across a suite of benchmarks consisting of an echo server with different traffic patterns, a time server, a http server, a stock quote server, a game server, and a chat server.

We modeled the execution time of the servers as having an architecture independent component and an architecture dependent component. The architecture dependent component includes implementation dependent overheads such as thread creation, object inflation, synchronization, and select processing.

The performance model works well for all the servers we benchmarked. The event-driven transformation resulted in speedups of up to a factor of 2 and slowdowns of up to a factor of 3 relative to the thread-per-connection architecture using the kernel-level threads package.

From the empirical study, one finds that no single server architecture works best for all servers. In fact, the optimal server architecture for a given server depends on the usage pattern. We observed both the extreme speedup and slowdown mentioned for the event-driven server architecture for the echo server. These benchmarks differ only in their client traffic patterns.

Thesis Supervisor: Martin Rinard
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I owe a debt of gratitude to many people, for their assistance and guidance in my research. First, to my advisor, Martin Rinard, for without his guidance, patience, encouragement, wisdom, and help this project could not have been done.

I'd like to thank Scott Ananian for his part in developing the FLEX compiler, answering my endless stream of questions regarding FLEX, fixing my long list of bug reports, and for finally getting the relinker to work.

I'd like to thank Karen Zee, Ovidiu D Gheorghioiu, and Catalin A Francu for their assistance in developing the event-driven transformation and asynchronous I/O libraries.

I'd like to thank Alexandru Salcianu for his meta-callgraph implementation, support, and assistance.

I'd like to thank Patrick Lam, Viktor Kuncak, Darko Marinov, Maria Cristina Marinescu, and Radu Rugina for their support and assistance.

I'd like to thank my parents and my brother for their moral support. I'd like to thank all of the great friends I've made during my stay at MIT for making MIT enjoyable.

I was supported by a Fannie and John Hertz Foundation Fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As mainstream computing environments shift from isolated personal computers to networks of computers, server programs are becoming an increasingly important class of application. We see a wide range of server applications deployed on the Internet today including such examples as web servers, instant message servers, IRC servers, news servers, and game servers among many others.

The client-server model is a simple abstraction to use when developing network-based applications. The server typically accepts multiple incoming connections and provides some sort of service. The client contacts the server to use whatever services the server provides. A common example of this interaction occurs in the web. Web servers such as Apache[1] provide the service of delivering web pages. Web browsers such as Netscape contact a web server of interest to request web pages.

Reliability and performance are serious concerns for server software. Businesses often use servers in environments such as electronic commerce where failure translates into direct loss of income. Higher performance server software requires fewer machines for a given task, which simplifies maintenance and saves money. In this report, we evaluate a variety of architectures used to implement servers and their performance implications.

Industry uses the programming language Java[11] for many server side applications. For example, many websites use the Volano chat server[12], a Java server application. Server application developers have an interest in Java partly because it brings many modern language features such as a strong type system and garbage collection to mainstream use. These features allow Java to provide many safety guarantees that languages like C do not provide. Furthermore, Java's flexible security model aids in the development of secure server applications.

Features of the Java language and runtime such as memory management and safety guarantees are very attractive when developing server applications. One important class of errors that Java prevents is buffer overruns. In recent years, up to 50% of the CERT advisories were due to buffer overruns[13]. Buffer overruns occur when a programmer allocates a fixed size memory region in which to store information, and then the programmer allows code to write data beyond the end of the region. This typically occurs in C programs that use library routines. The problem is that a malicious attacker can use these bugs to execute arbitrary code. Since Java provides array bounds checks, buffer overruns are impossible.

When we design servers, we need to consider several factors. Although all servers accept multiple connections and provide some service, they have very different design concerns. Some servers have very short interactions with clients, and in these cases, the overhead of setting up a connection is critical. Other servers have long interactions with many clients, and in these cases the scaling behavior of the server with respect to the number of simultaneous connections is more important.

Programmers have used many different programming models for handling multiple connections in servers. We consider the following programming models in this report:

Thread-per-connection model In this model, there is a single main thread which accepts connections and a worker thread for each opened connection. For the worker thread, the programmer concerns himself or herself with the interactions for just one client. The simplicity of this model makes it very attractive. The thread pooled architecture uses this same programming model, the only difference is how the program assigns connections to threads.

Event-driven model In this model, the programmer thinks in terms of I/O events and responses the server makes to these events. This model can be more complicated to develop for, since the programmer must explicitly manage I/O events.

In this report, we evaluate several server architectures including the thread-per-connection architecture with either kernel-level (or native) threads or user-level (or green) threads, an automatically generated event-driven asynchronous I/O architecture, and a thread-pooled server architecture. We develop a performance model for the various architectures, and evaluate the performance model over a range of server benchmarks.

A second contribution of this report is the development of a novel compiler transformation from a thread-per-connection architecture to an event-driven architecture.

We did all of the implementations using the FLEX compiler framework[2]. FLEX is a full featured, research Java compiler developed by our group. Our group designed FLEX to facilitate full and partial program analysis and transformations.

Chapter 2

Server Architectures

Figure 2-1 presents a very simple echo server design. This server accepts one incoming connection at a time and echoes whatever the client sends back to the client. The problem with this server is that it is often desirable to allow the server to serve more than one client at a time.

```
class Echo {
  static public void main(String args[])
    throws IOException {
    ServerSocket s = new ServerSocket(1000);  //Open Socket
    byte buffer[] = new byte[100];

    while (true) {
      Socket clientSocket = s.accept();  //Wait for incoming connection
      try {
        //Get stream objects
        OutputStream out = clientSocket.getOutputStream();
        InputStream in = clientSocket.getInputStream();
        while (true) {
          int length = in.read(buffer, 0, buffer.length);  //Read input from client
          if (length == -1) break;
          out.write(buffer, 0, length);  //Send input back to client
        }
        clientSocket.close();  //Close socket when done
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Figure 2-1: Single Threaded version of an Echo Server

Programmers have developed many different approaches to handling multiple clients simultaneously with one server. The simplest approach used is similar to simply running many copies of the single threaded server. We refer to this approach as a thread-per-connection architecture. Whenever a new connection occurs, the server simply starts a new thread to handle interactions with this connection. The programming model for this approach consists of conceptual threads that each handles a single connection. If a server services thousands of clients, all of the operating system bookkeeping to keep track of the various threads can become quite expensive. Single threaded solutions can avoid some of this overhead.

Another approach taken to handling multiple clients with one server is to have one thread keep track of all of the connections. This thread continuously checks for new connections and other events from any clients that require attention and runs the appropriate code to handle the event that occurred. We refer to this server design as an event-driven architecture. The programming model for this approach consists of events and responses to these events that the server makes.

In this chapter, we present in detail the thread-per-connection architecture using either the kernel-level or the user-level threads packages, the thread pooled architecture, and the event-driven architecture. We discuss qualitative advantages and disadvantages of each of these server architectures. We present quantitative results in Chapter 4.

2.1 Thread-per-connection architectures

Thread-per-connection or process-per-connection architectures are perhaps the simplest to develop since one can use the threading mechanism to manage the control for the many connections servers may have. This server architecture allows the programmer to focus on the communications necessary for a single connection. The thread package then takes care of managing the multiple connections through the threading mechanism.

In Figure 2-2, we show a simple example of a server written using a thread-per-connection architecture. In this example, the main thread waits for an incoming connection. Whenever an incoming connection occurs, the main thread spawns off a worker thread and passes the incoming connection off to the worker thread. Note that the worker thread code only explicitly handles one connection.

Thread-per-connection architectures typically follow this simple design pattern. For each incoming connection, the server spawns a worker thread. The code for the worker thread only needs to concern itself with the one connection that it manages.

The simplicity of design that this architecture provides makes it very attractive for

```
class Echo {
  static public void main(String args[])
    throws IOException {
    ServerSocket s = new ServerSocket(1000);  //Open Socket
    while (true) {
      Socket c = s.accept();  //Wait for incoming connection
      Worker w = new Worker(c);
      w.start();  //Start thread to handle incoming connection
    }
  }
}

class Worker extends Thread {
  Socket clientSocket;
  OutputStream out;
  InputStream in;

  Worker(Socket s) {
    clientSocket = s;
  }
  public void run() {
    try {
      //Get stream objects
      out = clientSocket.getOutputStream();
      in = clientSocket.getInputStream();
      byte buffer[] = new byte[100];
      while (true) {
        int length = doread(buffer);   //Read input from client
        if (length == -1) break;
        out.write(buffer, 0, length);  //Send input back to client
      }
      clientSocket.close();  //Close socket when done
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  int doread(byte[] buffer) throws java.io.IOException {
    return in.read(buffer, 0, buffer.length);
  }
}
```

Figure 2-2: Runtime Thread version of an Echo Server

quickly developing and debugging servers. Unfortunately, this architecture has to pay sometimes very expensive operating system overheads for thread creation and management.

The actual overheads that one pays depend greatly on the thread implementation. In this project, we consider two thread implementations: one using the LinuxThreads kernel-level pthreads implementation and another using a lightweight user-level threads package.

2.1.1 Thread Implementations

Thread packages provide programs with the means of running multiple computations conceptually at once. Thread packages also typically provide functionality for controlling the execution of multiple threads and functionality for sharing resources.

Programmers use two basic approaches to provide the capability of running multiple computations at once. One approach is to switch between threads. The kernel can do the thread switching as in kernel-level threads, or the program can use signals and/or explicit thread switches in library calls as is often the case in user-level thread packages. A second approach is to have multiple processors on a machine and to run the threads on different processors. However, if the system has more threads than processors, the thread package must share the processors between multiple threads. The threads package does this by having the processor switch between threads, which is a multiprocessor extension of the first approach. This switching has the downside of incurring additional operating system overheads and hurting cache performance.

Some means of coordinating threads is necessary. Thread packages often provide synchronization functionality. They also typically provide some sort of locking mechanism to guarantee exclusive access to the lock and some sort of wait and signal mechanism.

Threads typically live in the same address space, and they share resources such as file descriptors for open files and streams. These shared resources allow programmers to easily pass objects and open files between threads. This differs from the case of separate processes, which typically do not share the same address space or file descriptors.

Programmers commonly take two basic approaches to implement threads. One approach is to allow the operating system kernel to handle scheduling and I/O for the threads. We refer to this approach as kernel-level threads. A second approach is to allow libraries in user space to handle scheduling and I/O for the threads. We refer to this approach as user-level threads. We discuss these two approaches and the implementations in the following sections.

2.1.2 Kernel-level Threads

One thread system FLEX's runtime supports is LinuxThreads[14]. The LinuxThreads library provides kernel-level thread support. The linux kernel handles scheduling of threads and I/O.

Our runtime implements kernel-level threads using the LinuxThreads kernel-level clone based pthread implementation. This implementation allocates a pthread for each Java thread. The kernel handles the thread scheduling and I/O.

The scalability of LinuxThreads has many limitations. The Linux kernel has a limit on the number of running processes, LinuxThreads has a limit on the number of threads it can manage, virtual memory space places limits on the number of threads that can be created because each thread reserves virtual memory for its stack, and the Linux kernel has algorithms that scale with the number of processes[14].

Another disadvantage of kernel threads is that they are often quite expensive to allocate. We found this true for LinuxThreads. On our benchmark platform, a 275 Mhz StrongARM machine, a Java thread creation using LinuxThreads takes 2.2 milliseconds.

An advantage of using kernel-level threads is that they handle the Java I/O model very naturally. One simply calls the corresponding blocking I/O primitive and allows the thread to block. In this way, the program only incurs one system call overhead for each I/O request.

2.1.3 User-level Threads

The FLEX runtime also supports user-level threads. User-level threads provide threading support using only primitives available to user space programs. The user-level threads package provides thread multiplexing and I/O primitives. Part of the complication of developing a user-level threads package is that the user-level threads package cannot use blocking I/O calls, as doing so would cause all of the threads to halt. Instead, the lack of multiple operating system threads requires that we implement the blocking I/O model in Java using non-blocking primitives.

The user-level thread implementation handles the Java I/O model in a less efficient manner than kernel-level threads. It first tries the asynchronous version of the I/O request. If this fails, it performs a thread switch. After some number of thread switches, it uses the select call to check to see if there is any data ready. If so, it moves any threads that are now ready into the queue of threads that the scheduler will run. At this point, the thread package must repeat the I/O system call.

Our implementation allocates a stack for each thread, and performs thread switches on blocking I/O calls and synchronization calls. Our user-level thread implementation maps all of the Java threads to one kernel thread.

Some other examples of user-level thread packages include MIT Threads [10], PCR [8], and NSPR [3].

2.2 Thread Pooled Architecture

Thread pooled servers attempt to avoid thread creation overheads and scaling problems associated with hundreds of threads by simply not creating many threads. Instead, the server maintains a pool of threads, and as connections come in, the server farms the connections out to a pool of worker threads. Many servers, including Apache[1], use this sort of architecture.

Thread pooled servers are not without weaknesses. They can get worker threads tied up by slow or hostile clients. In this case, the thread pooled server either has to stop accepting new connections or spawn new threads and pay the thread creation overhead. A greater concern is that this approach does not generalize well to servers that require continuous connections such as chat servers or game servers.

We show an example of a thread pooled echo server in Figure 2-3. Note that the main thread begins by creating a pool of worker threads. It then starts accepting connections, and farming them out to the worker threads using a shared list. The worker threads remove incoming sockets out of the shared list and service them.

As previously mentioned, the thread pooled architecture eliminates thread creation overhead in most cases. Our implementation of the thread pooled servers use the same underlying kernel-level threads package as the thread-per-connection architecture. The rest of the overheads in the thread pooled architecture are very similar to the thread-per-connection architecture with a kernel-level threads implementation. The thread pooled architecture still pays largely the same synchronization and object inflation overheads and benefits from efficient handling of the Java I/O model from the underlying kernel-level thread implementation.

2.3 Event-driven Architecture

Event-driven architectures break down the server process into a set of I/O events and computations that the server does in response to these events. For example, an event-driven web server would receive an incoming event in the form of a client request and then would execute the code to request a file read. The server would then return to the event-driven loop. At some point in the loop, the server would check and see that the read finished. Once the file read event finished, the server would execute code to send the data read from the file to the client.

Event-driven architectures poll for I/O events. Upon receiving an I/O event related to one of the connections, the servers run the appropriate event processing code that takes in some state associated with the connection and the event. This architecture avoids some thread overheads associated with general-purpose thread implementations.

We present an example of an event-driven echo server in Figures 2-4, 2-5, and 2-6. Figure 2-4 shows the code that responds to accept events and initializes the connections. The code in Figure 2-5 responds to the server initializing a connection and checks for an incoming message. The code in Figure 2-6 responds to an incoming message and echos it back to the client. We omit the event-driven scheduler and I/O libraries for space reasons.

The reader might notice that these event-driven servers store their local variables in heap allocated data structures when blocking calls occur. These heap allocations incur some overhead for the original allocation and for increasing the garbage collection frequency.

Servers such as Zeus[15] and Flash[9] use event-driven architectures. The difficulty with this architecture is that event-driven servers are more difficult to write than their simpler thread-per-connection counterparts. The programmer has to explicitly manage all of the connections and write code in a complicated event-driven manner. To address these issues, we present a compiler transformation from the simple thread-per-connection architecture to the event-driven architecture.

We discuss this transformation in Chapter 3.

2.4 Qualitative Differences in Server Architectures

We explore a thread-per-connection architecture with both a kernel-level thread implementation and a user-level thread implementation, a thread pooled architecture, and a source transformation into an equivalent single threaded event-driven program.

These different server architectures have performance and programmability implications. The different versions incur different overheads for synchronization, thread creation, I/O, and procedure returns among other operations. Due to all of these different factors, it is difficult to predict the effect that the thread implementation has on a program's performance from first principles. Therefore, we chose to explore this space empirically with a set of server benchmarks.

2.4.1 Differences in Performance of the Server Architectures

The event-driven architecture has almost no synchronization cost, a very low thread creation overhead, and no per thread stack overhead. Because it uses heap allocated data structures, it incurs greater allocation and procedure return overheads. However, with more efficient garbage collectors than we considered, this overhead can be diminished[5]. The event-driven architecture may also incur a greater system call overhead due to its use of asynchronous I/O primitives. However, the effect of the system call overheads can also be greatly diminished by improving the operating system interface[6].

The thread-per-connection architecture with the user-level thread implementation is very similar to the event-driven version with the exception that it incurs a per thread stack overhead instead of any continuation overhead. It also incurs a greater synchronization overhead.

The thread-per-connection architecture with the kernel-level thread implementation has an extremely high overhead for thread creation and a more expensive synchronization overhead. It incurs a per thread stack overhead. However, it incurs a smaller system call overhead due to its use of blocking I/O primitives.

The thread pooled architecture does not create many threads, so it does not have a high thread creation overhead. With this exception, it behaves very similarly to the thread-per-connection architecture with kernel-level threads.

2.4.2 Differences in Programming the Server Architectures

The thread-per-connection architectures are the simplest to program. The developer can focus on the very simple connection oriented model. Each thread of execution only needs to concern itself with one connection.

The thread pooled architecture is only slightly more complicated. The worker threads are very similar to their counterparts in the thread-per-connection architecture, but after finishing a connection, the worker threads have to check for new connections waiting for service.

The event-driven architecture is perhaps the hardest to program. The programmer has to explicitly check for events, respond appropriately to events, and keep track of data structures for all of the threads. Notice that in the event-driven example in Figures 2-4, 2-5, and 2-6, that most of the code length comes from manipulating data structures to explicitly store and reload the connection's state. However, the transformation presented in the next chapter largely removes this disadvantage.

```
class Echo {
  static public void main(String args[])
    throws IOException {
    int ps = 4;
    ServerSocket s = new ServerSocket(1000);
    LinkedList ll=new LinkedList();

    for(int i=0;i<ps;i++) {
      //Create pool of worker threads
      Worker w=new Worker(ll);
      w.start();
    }
    while (true) {
      Socket c = s.accept();
      synchronized(ll) {
        //Farm out the incoming connection
        //to the pool of worker threads
        ll.add(c);
        ll.notify();
      }
    }
  }
}

class Worker extends Thread {
  LinkedList ll;
  Worker(LinkedList ll) {
    this.ll=ll;
  }
  public void run() {
    Socket clientSocket;
    while(true) {
      clientSocket=null;
      synchronized(ll) {
        do {
          try {
            //get an incoming connection
            clientSocket=(java.net.Socket)ll.removeFirst();
          } catch (Exception e) {
            try {
              //sleep if the linked list
              //is empty
              ll.wait();
            } catch (Exception ee) {
              ee.printStackTrace();
            }
          }
        } while(clientSocket==null);
      }
      try {
        OutputStream out = clientSocket.getOutputStream();
        InputStream in = clientSocket.getInputStream();
        byte buffer[] = new byte[100];
        while (true) {
          int length = in.read(buffer, 0, buffer.length);
       if (length == -1) break;
         out.write(buffer, 0, length);
        }
        clientSocket.close();
      } catch (IOException e) {
        System.err.println("IOException in Worker "+e);
      }
    }
  }
}
```

Figure 2-3: Thread Pooled version of an Echo Server

```
class Echo {
  static public VoidContinuation mainAsync
    (String args[]) throws IOException {
    ServerSocket s = new ServerSocket(1000);
    while (true) {
      ObjectContinuation oc = s.acceptAsync();
      if (oc.done==false) {
        Environment e=new mainEnvironment(s);
        mainContinuation mc=new mainContinuation(e);
        oc.setNext(mc);
        return mc;
      } else {
        Socket c=(Socket)oc.value;
        Worker w=new Worker(c);
        w.startAsync();
      }
    }
  }
}

class mainEnvironment {
  Object o1;

  public mainEnvironment(Object o1) {
    this.o1=o1;
  }
}

class mainContinuation implements
  ObjectResultContinuation {
  mainEnvironment env;

  public mainContinuation(mainEnvironment env) {
    this.env=env;
  }
  public void resume(Object c1) {
    Socket c=(Socket)c1;
    ServerSocket s=(Socket)env.o1;
    Worker w = new Worker(c);
    w.startAsync();
    while(true) {
      ObjectContinuation oc = s.acceptAsync();
      if (oc.done==false) {
        env.o1=s;
        c.setNext(this);
      } else {
        c=(Socket)oc.value;
        w=new Worker(c);
        w.startAsync();
      }
    }
  }
}
```

Figure 2-4: Event-Driven version of Connection Thread

```
class Worker extends Thread {
  Socket clientSocket;
  OutputStream out;
  InputStream in;

  Worker(Socket s) {
    clientSocket = s;
  }
  public VoidContinuation runAsync() {
    try {
      out = clientSocket.getOutputStream();
      in = clientSocket.getInputStream();
      byte buffer[] = new byte[100];
      while (true) {
        IntContinuation ic = doreadAsync(buffer);
        if (ic.done==false) {
          runEnvironment re=new runEnvironment(this,buffer);
          runContinuation rc=new runContinuation(re);
          ic.setNext(rc);
          return rc;
        } else {
          int length=ic.value;
          if (length == -1 )
             break;
          out.write(buffer,0,length);
        }
      }
      clientSocket.close();
    } catch(IOException e) {
      e.printStackTrace();
    }
    return VoidDoneContinuation();
  }
  IntContinuation doreadAsync(byte[] buffer)
    throws java.io.IOException {
    IntContinuation ic=in.readAsync(buffer, 0, buffer.length);
    if (ic.done==false) {
      readEnvironment re=new readEnvironment();
      readContinuation rc=new readContinuation(re);
      ic.setNext(rc);
      return rc;
    } else {
      return new IntDoneContinuation(ic.value);
    }
  }
}

class runEnvironment {
  Object o1,o2;

  public runEnvironment(Object o1, Object o2) {
    this.o1=o1;
    this.o2=o2;
  }
}
```

Figure 2-5: Event-Driven version of Worker Thread

```
class runContinuation implements
  VoidContinuation,IntResultContinuation {
  runEnvironment re;

  public runContinuation(runEnvironment re) {
    this.re=re;
  }
  public void resume(int length) {
    Worker this1=(Worker)re.o1;
    byte buffer[]=(buffer[])re.o2;
    try {
      if (length== -1) {
        clientSocket.close();
        if (next==null)
          return;
        else
          next.resume();
      }
      this1.out.write(buffer, 0, length);
      while(true) {
        IntContinuation ic = doreadAsync(buffer);
        if (ic.done==false) {
          re.o1=this1;
          re.o2=buffer;
          ic.setNext(this);
          return;
        } else {
          length=ic.value;
          if (length==-1) break;
          this1.out.write(buffer, 0, length);
        }
      }
      clientSocket.close();
      if (next==null)
        return;
      else
        next.resume();
    } catch (IOException e) {
      e.printStackTrace();
      if (next==null)
        return;
      else
        next.resume();
    }
  }
}

class readEnvironment {
}

class readContinuation implements IntResultContinuation {
  readEnvironment env;
  IntResultContinuation next;

  public readContinuatiuon(readEnvironment env) {
    this.env=env;
  }
  public setNext(IntResultContinuation next) {
    this.next=next;
  }
  public void resume(int length) {
    next.resume(length);
  }
}
```

Figure 2-6: Event-Driven version of Worker Thread

Chapter 3

Event-driven Transformation

3.1 Continuation Passing Style Conversion

The programming language community originally developed Continuation Passing Style or CPS Conversion to simplify control-flow to aid in the compilation of functional languages. The basic idea is that every procedure call takes in an additional argument, a function that expresses the remaining computation. The CPS conversion algorithm results in converted programs where every procedure call is in the tail call position, which is the last statement executed in a procedure, and where no procedure calls return.

In Figure 3-2, we show a simple Java example of CPS conversion. In Figure 3-1, we show the original program, and in Figure 3-2, we show the CPS converted program. We use classes to encapsulate the code and environments for continuations.

Every procedure call in the transformed version takes in a continuation in addition to its previous arguments. Moreover, every path through every procedure makes exactly one call, a tail call at the very end of the procedure. A more complex Java program that used local variables would use environments to store the local variables.

```
class Original {
  static public void main() {
    System.out.println(foo());
    System.exit();
  }
  int foo() {
    return 2;
  }
}
```

Figure 3-1: Original Sample Program

The relevant property of CPS conversion for our event-driven transformation is that the

```
class Converted {
  static public void main() {
    foo(new IntContinuation1());
  }
  void foo(IntContinuation a) {
    a.continuation(2);
  }
}

class IntContinuation1 implements IntContinuation {
  public continuation(int i) {
    System.out.println(i, new VoidContinuation1());
  }
}

class VoidContinuation1 implements VoidContinuation {
  public continuation() {
    System.exit();
  }
}
```

Figure 3-2: Example of CPS Conversion for a Java Program

thread stores all of its remaining state in the continuation. We can use a modified form of CPS to return the continuation to a scheduler instead of having procedures directly invoke continuations. This provides a mechanism for suspending and resuming computations.

CPS conversion for suspended calls makes procedure calls much more expensive. Instead of simply storing local variables on a stack, CPS converted programs allocate a heap object, copy the local variables into the heap object, and then copy the local variables out of the heap object. To minimize this additional overhead, we use a selective CPS conversion. We discuss this conversion in more detail in the following section.

3.2 Basic Transformation

To convert programs from the multithreaded, blocking I/O model into the event-driven I/O model, we identify I/O operations that may potentially block. At these points, we need the ability to suspend the computation of one thread and to resume it later when the I/O is ready. To do this, we use a modified CPS conversion.

The first modification is that the program passes continuations back to a scheduler. The scheduler executes the continuation only when the I/O is ready. Instead of passing continuations into methods, we have methods return a continuation that represents the remaining computation for the called method. In this way, we enable optimizations that avoid the overhead of generating unnecessary continuations and invoking the scheduler.

The second modification is a performance concern. The overhead of generating continuations at every procedure could be quite substantial. Since the program requires the capability to suspend and resume the computation only when the program executes blocking I/O primitives, we only need to generate continuations for methods in call chains that end in a potentially blocking I/O primitive.

We identify blocking I/O primitives by hand auditing the native calls in the Java class library to determine the set of native I/O methods that may block. We refer to these as directly blocking methods. We provide replacement asynchronous versions for directly blocking methods that return continuations.

At compile time, the compiler identifies all methods that may be part of a call chain that ends with the program calling a directly blocking method. We do this by generating a call graph and identifying all methods that can directly or indirectly call one of the identified blocking I/O primitives. We refer to these methods as indirectly blocking methods. Our compiler transforms indirectly blocking methods into methods that return continuations.

For efficiency reasons, it is important to have a precise call graph. This minimizes the unnecessary use of the more expensive continuation passing style calling convention. Our implementation uses a call graph algorithm similar to Ole Agesen's Cartesian product algorithm [4].

For an indirectly blocking method, the compiler identifies any call sites that may call directly or indirectly blocking methods. The compiler replaces each of these call sites with a call to the replacement version of the method, which returns a continuation. After each one of these call sites, the compiler generates a continuation object for the current method. The continuation includes an environment for storing the values of live local variables, a normal resume method containing code for the computer to execute upon the normal return of the call site's callee, and a exceptional resume method containing code for the computer to execute if the call site's callee throws an exception.

The callee method needs a pointer to the caller, so that upon completion it can run the caller's continuation. In order to do this, the transformed code returns the continuation it built in the current method to the caller. After the caller builds its continuation object, the caller sets a link from the callee's continuation object to the caller's continuation object.

The transform also needs to handle the case where a potentially blocking method does not block. To do this, the transformed code must wrap the return value in an identity continuation for any return or throw statements reachable in a method without calling a possibly blocking method.

The generated resume methods start by unpacking the local variables from the environment. The resume method must include all the code in the original method that is reachable from the return of the original call site. For any potentially blocking call site, the resume method takes the returned continuation, links it to the previously stored next continuation, and returns to the scheduler. For return or throw instructions, the resume method simply calls the resume method of the calling method with the appropriate value in its linked next continuation.

For any non-blocking method that may be called at the same site as a blocking method due to virtual calls, the compiler must generate a separate version with the same calling semantics as the blocking methods.

An event-driven scheduler manages the continuations, polls select for pending I/O, and provides thread functionality.

3.3 Optimizations to the Basic Transformation

The basic transformation provides the opportunity for many optimizations. The obvious one is that when no blocking I/O operation has occurred or the I/O is immediately ready, it is not necessary to incur the overhead of generating continuations and returning to the scheduler. We implement an optimization where I/O operations may optimistically return a value if possible. Continuation objects contain a flag indicating whether the caller can immediately recover the return value of the callee, or whether it must return to the scheduler.

Another optimization used is that in loops with blocking I/O, it is often the case that the same call site blocks in every iteration. In this case, we can simply reuse the data structures from the previous iteration. By recycling the continuations, we avoid the overhead of requesting new memory and garbage collecting the old continuations.

Our transformation as implemented and the example below use both of these optimizations.

3.4 Example

For our example, we consider the simple echo server shown in Figure 2-2. A call graph tells us that the methods main, run, and doread can potentially directly or indirectly call the blocking I/O primitives, read and accept. Therefore, the compiler needs to transform all of these methods.

In the method main, the analysis identifies a potentially blocking I/O primitive, accept. At this point, the compiler inserts code that saves the state of the thread if the accept call can not immediately return a socket.

The compiler builds the class mainEnvironment to store the live local register values at the accept call, and the class mainContinuation to store the code for the remaining computation in the main method. The transformed resume method in the class mainContinuation first restores the live local variables and then continues with the remaining computation for the method. The transformed method uses the recycling optimization. If this method blocks at the accept call, it simply reuses the old continuation objects. We reproduce the transformed code for the Echo class in Figure 3-3 from Chapter 2 for the reader's convenience.

If a blocking method has a potentially non-blocking execution path, the method's type still requires it to return an object of a continuation type to the caller. The runAsync method shows an example of this using the "return VoidDoneContinuation()" statement to generate an identity continuation.

We show the transformed code for the worker class in Figure 3-4 and 3-5.

3.5 Extensions

We optimized our implementation of the event-driven transformation and scheduler for what we believe to be common paradigms in server applications. Our implementation assumes that each thread will execute a finite number of non-blocking instructions between each blocking I/O primitive and that no program holds a lock while calling a blocking I/O primitive. Our implementation only supports uni-processors. We can trivially remove all of these limitations, but in doing so, the transformed servers incur some additional overhead. We describe these extensions below.

We can develop further optimizations to reduce the overheads incurred in the event-driven transformations. One such optimization is explicitly managing the memory used for the continuations. We discuss this optimization in detail later in this section.

The event-driven transformation is useful for architectures other than just using our asynchronous I/O library to generate a single-process event-driven server. We discuss these other possibilities later in this section.

Synchronization and other Threading Constructs Support for synchronization can be added using a two step process. The first step is to determine which synchronization operations need to remain. For an inefficient implementation, we can skip this step and leave all the synchronization operations in the program. Otherwise, this can be done using a fixed point algorithm and the results from a pointer analysis algorithm. We also need to add the wait methods to our list of primitive blocking methods.

By synchronization, we mean simple mutual exclusion locks. Java bytecode encodes the operation to acquire a lock as a monitorenter instruction and the operation to release a lock as a monitorexit instruction. The Java language constructs that can result in the generation of monitorenter and monitorexit instructions result in the generation of structured pairs. Unfortunately, arbitrary sequences of monitorenter and monitorexit instructions are legal in bytecode. For example, Java can only generate nested sequences of locks like monitorenter A, monitorenter B, monitorexit B, monitorexit A. However, one can legally express monitorenter A, monitorenter B, monitorexit A, monitorexit B in bytecode.

Therefore, the analysis needs to be correct for arbitrary locking sequences, but precise only for the paired locking constructs that the Java language generates.

The first stage of such an analysis would consist of recognizing the paired monitorenter and monitorexit constructs that exists in the original Java language. To do this, the analysis would begin with a dataflow analysis on each callable method. The dataflow analysis would determine at each program point which locks the method obtained and in which order. This type of analysis is possible because the monitorenter and monitorexit statements obtain the object from the same local pointer. At merge statements, the analysis would check that the incoming points have acquired the same locks in the same order. If the incoming program points had not, the analysis would conservatively declare all of the locks that any of the incoming program points to the merge had acquired as necessary. If at any monitorexit statement the pairing property does not hold, the analysis declares all the locks that the offending statement holds as necessary. When the analysis declares a monitorenter statement as necessary, the analysis must also declare all other monitorenter statements that may lock on the same objects as necessary. We use the results of the pointer analysis to conservatively mark these other monitorenter sites as necessary.

The second part of the analysis is a fixed point algorithm. The algorithm would begin with a set of methods S that contain locks that the analysis declared necessary or that indirectly or directly blocking. The algorithm would cycle through all of the callable methods in the program, looking for call sites that may call any method in S. When the analysis finds such a call site, the analysis declares all locks that the first stage of the analysis indicates as being held at this point as necessary. The analysis declares any monitorenter site that may operate on the same object as these locks as necessary, and the analysis adds the appropriate set of methods to S. The analysis continues cycling through all of the callable methods until the analysis reaches a fixed point.

Upon termination, the algorithm guarantees that any monitorenter/monitorexit pair surrounding any possible thread switch point can generate a continuation at the monitorenter call. The algorithm terminates because there are only a finite number of synchronization operations that the analysis may declare necessary.

The second part of the process is to treat the monitorenter statements that we left in as potentially blocking for the event-driven transformation. We build optimistic continuations at these statements. Support for synchronization and methods such as notify/wait would then be implemented in the event-driven scheduler.

Fairness The current implementation makes no fairness guarantees. If some thread executes an infinite loop that does not contain a blocking I/O operation, all other threads will starve. We could fix this by looking for back edges in the graph representation of the method. By simply treating back edges as we would treat blocking calls and returning to the scheduler, we can remove this limitation.

Multiprocessor Support We can support multiprocessors by allowing the scheduler to run multiple continuations at once. Because of the additional concurrency allowed in this system, all synchronization statements that the standard multithreaded version requires must remain in the multiprocessor event-driven version.

Multiprocessor support requires that we modify the event-driven scheduler to allow more the schedule to execute more than one continuation at once.

Explicit continuation/environment management One source of overhead in our implementation of the event-driven transformation is that the continuations are heap allocated. At every single blocking function call site, the transformed program generates a continuation object. These continuation objects have lifetimes that mirror the original stack implementation. Therefore, the program could allocate fixed areas of memory for each of the original threads to store the continuations.

Other I/O libraries The event-driven transformation can be easily adapted to use other asynchronous I/O libraries. One could also use event-driven I/O models such as the asymmetric multi-process event-driven (AMPED) architecture that the Flash webserver[9] uses. The AMPED architecture addresses the issue that in some cases for some versions of UNIX, non-blocking reads may actually block on disk files. The AMPED architecture works around this problem by using a pool of worker threads to handle file reads. By simply dropping in a replacement I/O library, our transformation can automatically generate the corresponding event-driven servers from servers written in the thread-per-connection architecture.

```
class Echo {
  static public VoidContinuation mainAsync
    (String args[]) throws IOException {
    ServerSocket s = new ServerSocket(1000);
    while (true) {
      ObjectContinuation oc = s.acceptAsync();
      if (oc.done==false) {
        Environment e=new mainEnvironment(s);
        mainContinuation mc=new mainContinuation(e);
        oc.setNext(mc);
        return mc;
      } else {
        Socket c=(Socket)oc.value;
        Worker w=new Worker(c);
        w.startAsync();
      }
    }
  }
}

class mainEnvironment {
  Object o1;

  public mainEnvironment(Object o1) {
    this.o1=o1;
  }
}

class mainContinuation implements
  ObjectResultContinuation {
  mainEnvironment env;

  public mainContinuation(mainEnvironment env) {
    this.env=env;
  }
  public void resume(Object c1) {
    Socket c=(Socket)c1;
    ServerSocket s=(Socket)env.o1;
    Worker w = new Worker(c);
    w.startAsync();
    while(true) {
      ObjectContinuation oc = s.acceptAsync();
      if (oc.done==false) {
        env.o1=s;
        c.setNext(this);
      } else {
        c=(Socket)oc.value;
        w=new Worker(c);
        w.startAsync();
      }
    }
  }
}
```

Figure 3-3: Event-Driven version of Connection Thread

```
class Worker extends Thread {
  Socket clientSocket;
  OutputStream out;
  InputStream in;

  Worker(Socket s) {
    clientSocket = s;
  }
  public VoidContinuation runAsync() {
    try {
      out = clientSocket.getOutputStream();
      in = clientSocket.getInputStream();
      byte buffer[] = new byte[100];
      while (true) {
        IntContinuation ic = doreadAsync(buffer);
        if (ic.done==false) {
          runEnvironment re=new runEnvironment(this,buffer);
          runContinuation rc=new runContinuation(re);
          ic.setNext(rc);
          return rc;
        } else {
          int length=ic.value;
          if (length == -1 )
             break;
          out.write(buffer,0,length);
        }
      }
      clientSocket.close();
    } catch(IOException e) {
      e.printStackTrace();
    }
    return VoidDoneContinuation();
  }
  IntContinuation doreadAsync(byte[] buffer)
    throws java.io.IOException {
    IntContinuation ic=in.readAsync(buffer, 0, buffer.length);
    if (ic.done==false) {
      readEnvironment re=new readEnvironment();
      readContinuation rc=new readContinuation(re);
      ic.setNext(rc);
      return rc;
    } else {
      return new IntDoneContinuation(ic.value);
    }
  }
}

class runEnvironment {
  Object o1,o2;

  public runEnvironment(Object o1, Object o2) {
    this.o1=o1;
    this.o2=o2;
  }
}
```

Figure 3-4: Event-Driven version of Worker Thread

```
class runContinuation implements
  VoidContinuation,IntResultContinuation {
  runEnvironment re;

  public runContinuation(runEnvironment re) {
    this.re=re;
  }
  public void resume(int length) {
    Worker this1=(Worker)re.o1;
    byte buffer[]=(buffer[])re.o2;
    try {
      if (length== -1) {
        clientSocket.close();
        if (next==null)
          return;
        else
          next.resume();
      }
      this1.out.write(buffer, 0, length);
      while(true) {
        IntContinuation ic = doreadAsync(buffer);
        if (ic.done==false) {
          re.o1=this1;
          re.o2=buffer;
          ic.setNext(this);
          return;
        } else {
          length=ic.value;
          if (length==-1) break;
          this1.out.write(buffer, 0, length);
        }
      }
      clientSocket.close();
      if (next==null)
        return;
      else
        next.resume();
    } catch (IOException e) {
      e.printStackTrace();
      if (next==null)
        return;
      else
        next.resume();
    }
  }
}

class readEnvironment {
}

class readContinuation implements IntResultContinuation {
  readEnvironment env;
  IntResultContinuation next;

  public readContinuatiuon(readEnvironment env) {
    this.env=env;
  }
  public setNext(IntResultContinuation next) {
    this.next=next;
  }
  public void resume(int length) {
    next.resume(length);
  }
}
```

Figure 3-5: Event-Driven version of Worker Thread

Chapter 4

Empirical Evaluation

In this section, we quantify the performance differences of the various server architectures across a variety of server applications. Note that we continue to use the term architecture to refer to software design and not to refer to the various hardware platforms introduced in this chapter. The factors in the performance differences between the various server architectures can be quite complicated, and they can include factors such as how the server handles concurrency, the caching system, overheads of building continuations, and differences in GC performance among many other things. We examine the performance of these servers in a much simpler model. We believe that for many applications, the most significant differences between the server architectures are the thread creation overhead, the synchronization overhead, and the object inflation overhead.

4.1 Performance Model

We use a simplified model for execution time of the servers for a given workload as follows:

$$t_{total} = t_{architecture\ independent} + t_{architecture\ dependent}$$

$$t_{architecture\ dependent} = t_{thread\ creation} + t_{object\ inflation} + t_{synchronization}$$

This model separates the architecture independent cost $t_{architecture\ independent}$ from the implementation dependent cost. The architecture independent time consists of the time the program spends executing or waiting on operations that we do not model as being different between the architectures.

There are other architecture dependent factors that we omit in this description of server performance. These omitted factors are typically small relative to the modelled ones. This is not true for all possible server behaviors, and an example of this shows up in the serial echo benchmark and we discuss this example in later sections.

The thread creation overhead is how much time the processor uses per thread creation operation. Some implementations of threads have rather costly thread creation operations. The synchronization overhead is how much time the server uses per synchronization operation on an object. The object inflation is how much time the server uses per object inflation operation. Object inflation occurs because the runtime does not allocate memory by default for some structures, instead the runtime allocates memory on demand for the first operation requiring one of these structures. One such operation is object synchronization. We discuss the causes of these overheads in more depth in the next section.

We begin by measuring thread creation, synchronization, and object inflation overheads with microbenchmarks. We use these measured overheads with runtime instrumentation counts to gain insight into the performance of the various server architectures.

4.2 Discussion of Overheads

In this section, we discuss the thread creation, synchronization, and object inflation overheads in more detail.

4.2.1 Thread Creation

When a program creates a thread, the runtime must do some amount of work to set up the appropriate bookkeeping structures and resources. For example, in order to create a new thread in Linux's kernel-level threads implementation, the thread package reserves virtual memory for the stack, adds a process entry to the kernel tables, and sets up structures inside the thread library. For user-level threads, the runtime must allocate a stack, the runtime must allocate and initialize thread specific data structures, and the runtime must add the thread to the ready queue. These actions take some amount of time that depends on the thread implementation. We quantify this overhead as the thread creation overhead.

4.2.2 Object Inflation and Synchronization

Synchronization and object inflation overheads are closely related. The Java language provides locking primitives that allow threads to lock, suspend, or wake suspended threads. Moreover, Java associates a full-featured lock structure with every object created in the program.

To implement locks, thread packages need some amount of memory for keeping track of the lock state. To avoid having to allocate this amount of memory for every object created by a Java program, many Java implementations do not allocate the memory required for the locking data structure for a given object until a locking operation actually occurs on that object. At this point, an operation known as object inflation occurs.

In Java, each object has a hashcode associated with it. In our compiler, for inflated objects we reuse this space as a pointer to an inflated object data structure, and move the hashcode to the inflated object data structure. In the case of our compiler, the program allocates memory for the lock data structure for the given object, moves the object's hashcode to the newly allocated memory, flips an unused bit in the hashcode to flag the change, and uses the remaining bits of the hash code to point to the newly allocated memory. Effectively, the process of inflating objects gives an additional overhead to the first synchronization operation to occur on an object. This overhead is the object inflation overhead.

To guarantee exclusive access to a lock, the program must execute some algorithm each time it calls a locking primitive. The time this algorithm takes to execute is the synchronization overhead.

4.3 Experimental Setup

Our experimental setup consists of a client simulator running on a 866 Mhz Pentium III processor with 512 MB of memory, and a server running on a Netwinder with a 275 Mhz StrongARM processor with 128 MB of memory. We connected the two computers via 100 megabit Ethernet through a LinkSYS 5 port 10/100 baseT Ethernet switch. We isolated the test network to prevent outside traffic from influencing the results.

We chose this benchmark platform to insure that the server's performance was the bottleneck. We chose a relatively fast client machine and network to ensure that the server is always busy. We wrote the clients in either C or Java. We compiled the Java clients for the Pentium III using our compiler's C backend with the event-driven transformation enabled.

4.4 Measuring Overheads

In this section, we discuss how we measure the thread creation, synchronization, and object inflation overheads with microbenchmarks. Knowledge of these overheads along with counts of the given operation allows us to calculate the architecture dependent times discussed in the previous section.

Our thread creation microbenchmark simply executes 10,000 thread creations, runs, and joins for threads that execute the empty method under each architecture. For each architecture, we present the average cost of this in Table 4.1. This benchmark is not applicable to the thread pooled server since thread pooled servers do not create threads except for a static pool at the beginning of execution.

Our synchronization microbenchmark captures the synchronization cost and the object inflation cost. The microbenchmark does this by synchronizing on the same object many times in a loop. This gives us the time taken for a synchronization operation.

A separate loop synchronizes on newly created objects many times in a loop. This gives us the time taken for a synchronization operation, an object inflation operation, and an object allocation. To separate the object inflation cost, we measure the allocation cost in a separate loop that simply allocates new objects. With all of the costs known, we can simply subtract to calculate the cost of object inflation.

We present the results of the object synchronization and inflation benchmarks in Table 4.1. Note that the synchronization cost shown for the event-driven version of the server is simply the cost of a native call to an empty routine. These calls remain, because the transformation does not attempt to remove them. The event-driven version performs no actual synchronization.

4.5 Benchmarks

Our set of benchmarks includes: Chat, Quote, HTTP, Echo, Game, Phone, and Time. We intend these benchmarks to simulate many real world server applications.

Server Architecture	Event-Driven	Thread Pooled	Thread-Per-Connection w/ Native Threads	Thread-Per-Connection w/ User Space Threads
Thread Creation Cost (ms)	0.21	N/A	2.21	0.65
Object Inflation Cost (ms)	N/A	0.04333	0.04333	0.03212
Synchronization Cost (ms)	0.0014	0.0044	0.0044	0.0025

Table 4.1: Microbenchmark Results for Server Architectures

Chat Server The chat server is a simple multiperson chat room server. A user connects to it, and the server rebroadcasts any messages inputted to all other users. Our client simulation opens a set of connections to the server, each connection sends a number of messages, and then waits for the server to deliver all the messages. The chat server is a modified version of a chat server written by Lou Schiano and it is available on the web at http://www.geocities.com/SiliconValley/Bay/6879/chat.html.

Quote Server The quote server is a simple stock quote server. A client connects to the server and requests the prices of various stocks. Our client simulation starts off a number of threads, each thread repeatedly requests a stock price and waits for a response from the server before going to the next request. The quote server is a modified version of StockQuoteServer written by David W. Baker, and it is available in Que's Special Edition Using Java, 2nd Edition.

HTTP Server The HTTP server is a simple web server. Our client simulation attempts to simultaneously request web pages from the servers. The HTTP server is a modified version of JhttpServer written by Duane M. Gran.

Echo Server The echo server simply echoes any input back to the client. We have four different clients for this server. The serial echo benchmark opens a set of connections, does a request and a response on each connection in order, and then cycles through the request and response sequence for a number of times. In this benchmark, only one connection at a time ever has data sent across it.

The limited parallelism echo benchmark opens a set of connections, sends a request down each connection, reads the response and sends another request for each connection in order, and finally repeats the last step for a number of times. In this benchmark, many connections can be active at once.

The long connection echo benchmark opens a set of connections, sends a message down all of the connections, and then waits for responses. When any response comes in, the client simulator immediately sends another request until it has sent the desired number of messages across each connection. In this benchmark, the client simulator keeps many connections active at once.

The short connection echo benchmark opens a pool of connections and sends messages down them. Whenever the client simulator receives a response, it closes the corresponding connection, opens a new one, and repeats until the client simulator has sent the desired number of requests and received the desired number of responses. In this benchmark, the client simulator keeps many connections active at once.

Game Server We designed the game server to simulate a server that pairs online game clients. The client simulator simply opens pairs of connections and pings messages back and forth over these connections.

Phone Server The phone server provides phone look up and phone entry services to incoming connections. The client simulator simply connects to the server, and each connection adds one entry to the database.

Time Server The time server simply tells any incoming connections the current time. Our simulation attempts to keep open a certain number of connections querying the server for the time.

4.6 Benchmark Methodology

We executed the benchmarks on an isolated network of two machines. Each machine only ran the benchmark processes and necessary system processes. We hardcoded the mapping of hostnames to IP addresses using the /etc/hosts name resolution mechanism in Linux. We repeated each benchmark 30 times and averaged the results.

The compiler instrumented the binaries to count synchronization and inflation operations. We calculated the number of thread creations from the clients' connection pattern.

The clients attempted to maintain a load of 50 active client connections on the server. In some cases, such as the short connection echo benchmark, although the client may attempt to create a load of 50 clients, the server does not actually see 50 simultaneous active threads. This is because the 50 active connections the client sees includes connections the server has already served but are waiting in operating system buffers and connections that the server hasn't accepted yet but are waiting in operating system buffers.

We configured the thread pooled servers to use pools of 4 worker threads for the short echo, the time, the http, and the phone benchmarks. The rest of the benchmarks required more than 4 connections open at once, so we set the pool size to the number of simulated clients. We empirically determined the number 4 to yield good results for our setup.

4.7 Benchmark Results

In this section, we present and discuss the benchmark results. We evaluate the performance model in qualitative terms, with a quantitative evaluation of the performance model appearing in a later section.

Echo and Time Servers In Figure 4-1, we present the benchmark results for the echo server using the limited parallelism echo benchmark, the long connection echo benchmark, and the short connection echo benchmark; and the time server. These servers are the simplest in our benchmark suite. Modeling the performance of these benchmarks in terms of an architecture independent execution time plus a thread creation overhead, an object inflation overhead, and a synchronization overhead works extremely well.

For comparison to the results in Figure 4-1, we benchmarked an event-driven C version of the server. The C version takes 1.60 seconds for parallel echo benchmark, 1.64 seconds for the long connection echo benchmark, and 3.25 seconds for the short connection echo benchmark. The C echo server takes roughly a third less time for the benchmarks. This extra time the Java server takes can likely be attributed to additional safety checks such as array bounds checks that are present in the Java version, a more generalized event-driven support system, native call overheads, and optimization differences between our research Java compiler and a production C compiler. The Java version of the echo server does not make extensive use of the class library functionality, so this should not play much of a role.

The parallel echo and long echo benchmarks perform about the same for all the architectures. This is because they do not create threads during the execution and execute minimal synchronization operations.

Server architectures with low thread creation overheads like the event-driven or thread-pooled architectures do extremely well on the short echo and time benchmarks. This is due to the large number of threads created by these benchmarks. Note that the architecture independent time is similar for all of the benchmarks, indicating that the performance model works well for these benchmarks.

In Figure 4-2, we present the benchmark results for the game, web, chat, phone and quote servers. Our simple performance model works well on most of these servers, but it does break down somewhat on the quote and game server.

Chat Server The chat server creates very few threads during its execution. A more important factor in the performance of the chat server is the synchronization overhead. The performance model appears to work reasonably well for the chat server with some imperfection for the user-level threads.

Quote Server The quote server benchmark creates very few threads during its execution because the client simulator only opens 50 connections at the very beginning. Because of this connection pattern, the thread creation overhead is not very important in this benchmark. The event-driven and user-level thread architectures do better on this benchmark than the performance model predicts.

HTTP Server For comparison to the http server results in Figure 4-2, Apache takes 4.30 seconds to complete the same task. The large performance difference between Apache and our web servers can likely be attributed to extensive use of inefficient class library functionality, additional safety checks such as array bounds checks that are present in the Java version, native call overhead, and optimization differences between our research Java compiler and a production C compiler. The http server benchmark benefits from architectures with low thread creation overheads and from low object inflation overheads. Note that the architecture independent time is similar for all of the architectures, indicating that the performance model works well for this benchmark.

Game Server The game server benchmark creates very few threads during its execution, so thread creation overhead is not very important. The event-driven and user-level thread architectures do better on this benchmark than the performance model predicts.

Phone Server The phone server benchmark benefits from low thread creation overheads and from low object inflation overheads. Note that the architecture independent time for the phone server benchmark is similar for all of architectures, indicating that the performance model works well for this benchmark.

Serial Echo Benchmark In Figure 4-3, we present a benchmark for which the model breaks down. For comparison to these results, the C version of the echo server takes 3.16 seconds to execute the serial echo benchmark. Note that this is slower than the thread-per-connection benchmark with the kernel-level threads implementation. This is because the event-driven C server suffers from the same problem as the event-driven Java version and the user-level threads version.

This benchmark demonstrates a problematic behavior region for the event-driven and user-level thread implementations. After every incoming connection, these servers must call select, because there are no other continuations or threads ready for execution. Since the client only interacts with one connection at a time, the server can never amortize the overhead of calling select and processing the returned results over multiple threads. This benchmark calls select 5,000 times, whereas the parallel version only calls select 100 times. But both of these benchmarks send the same total volume of traffic over the same number of connections. Eliminating the select overhead is a topic of active research and we discuss it later in this chapter.

4.8 The Serial Echo Benchmark

To explain the results of the serial echo benchmark, it is necessary to extend our previous performance model. We do this by adding a term to represent the time it takes to process the select call to get:

$$t_{architecture\ dependent} = t_{thread\ creation} + t_{object\ inflation} + t_{synchronization} + t_{select\ processing}$$

We measured the select processing overhead for the event-driven and the user threads packages with a microbenchmark. The microbenchmark called the select processing code with 49 threads blocked waiting to read serial I/O and 1 thread waiting to read "/dev/zero" I/O that was ready. We give the overheads in Table 4.2. This gives a total select processing time of 4.46 seconds for the event-driven server and 1.13 seconds for the user-level threads server. Note that we cannot simply time the select processing code for the server benchmark,

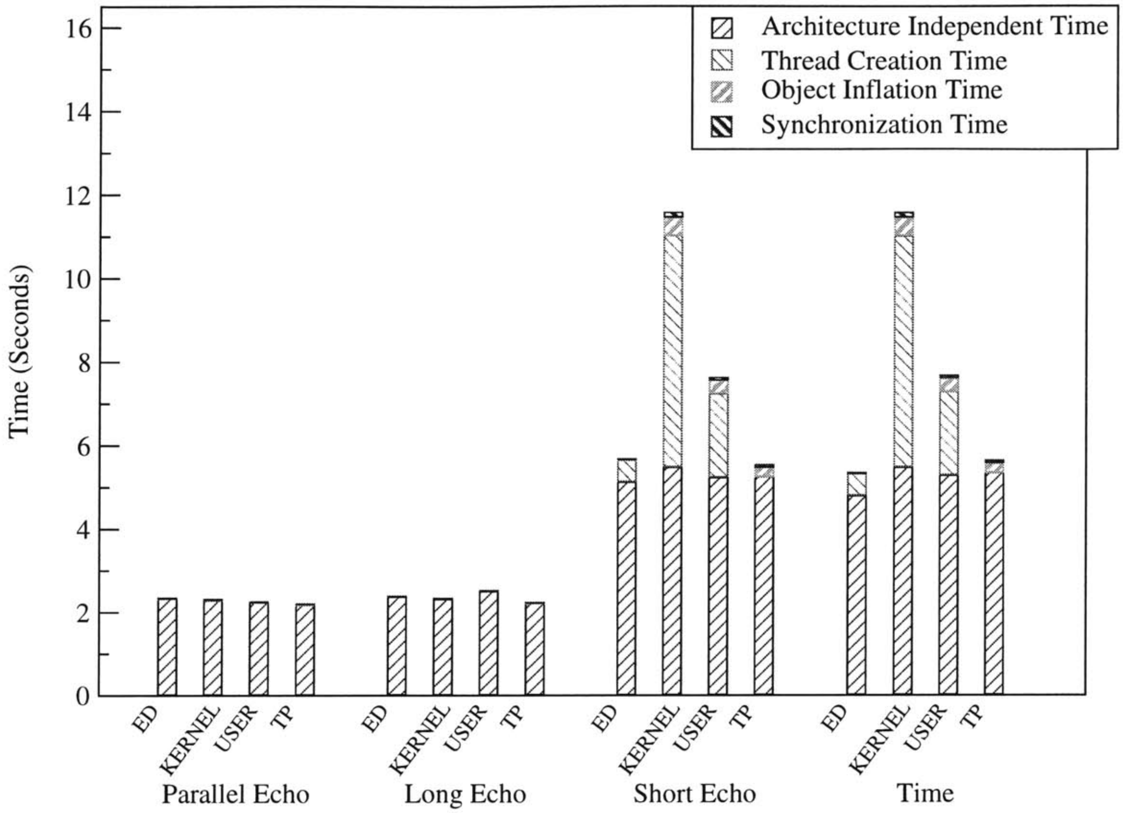

Figure 4-1: Echo and Time Servers

because the network latency appears in the amount of time select blocks for. If we simply timed the select processing code, we would effectively count the network latency in the select processing overhead. Since the network latency is a server architecture independent time and the kernel-level threads do not have a select call to measure, this is not the correct approach.

Server Architecture	Event-Driven	User Space Threads
Select Processing Cost (ms)	0.891	0.225

Table 4.2: Microbenchmark Results for Select Evaluation

We present these results in graphical form in Figure 4-4.

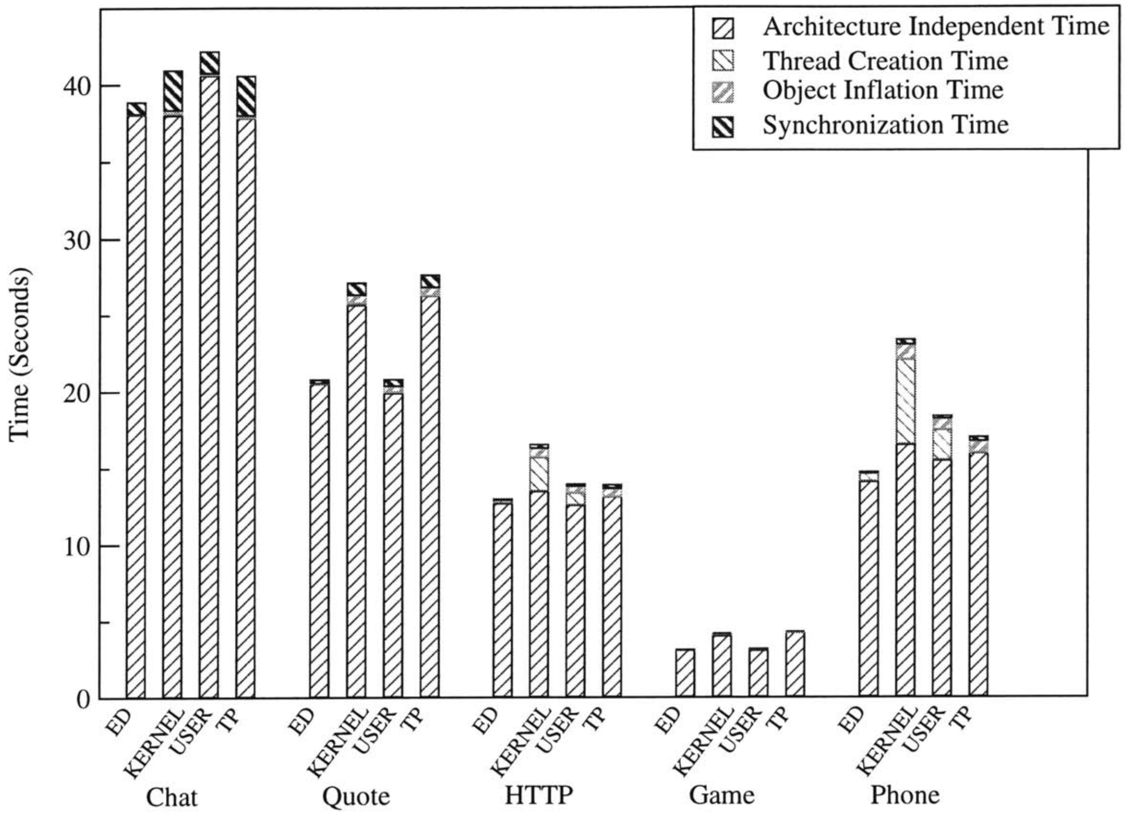

Figure 4-2: Game, Web, Chat, Phone and Quote Servers

4.9 Quantitative Evaluation of the Performance Model

The performance model attempts to split the execution time of the servers into an architecture dependent portion and an architecture independent portion. For a given server, the architecture independent time should remain constant across all architectures. A meaningful evaluation of the performance model should numerically express how invariant the architecture independent time is across the various architectures.

The standard deviation of the architecture independent times gives a measure of how close the architecture independent times are. However, this number is tied to the actual execution time of the benchmark. To remove this dependence, we can divide the standard deviation by the average architecture independent time for the given benchmark. This number is called the coefficient of variation in statistics.

To help understand which components of the performance model are important for the benchmarks we ran, we present results for five different models ranging from no model to

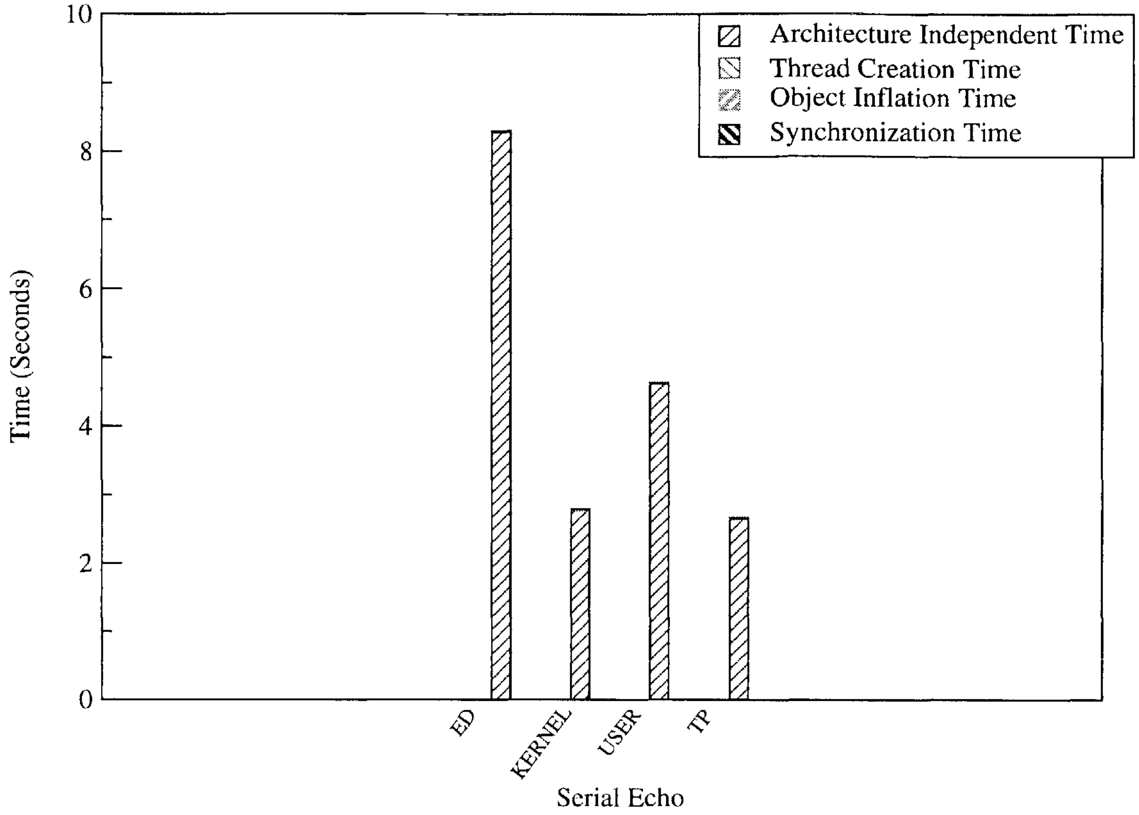

Figure 4-3: Serial Echo Benchmark

the select compensated model used for the serial echo benchmark.

The *no model* performance numbers are for a model with an architecture dependent overhead described by the equation $t_{architecture\ dependent} = 0$. This model assumes that all the architectures are essentially the same.

The *thread creation model* performance numbers are for a model described by the equation $t_{architecture\ dependent} = t_{thread\ creation}$. This model assumes that the only significant difference between the architectures is in the thread creation time.

The *object inflation model* performance numbers are for a model described by equation $t_{architecture\ dependent} = t_{thread\ creation} + t_{object\ inflation}$. This model assumes that thread creation and object inflation are the only significant differences in the architectures.

The *complete model* performance numbers are for a model described by the equation $t_{architecture\ dependent} = t_{thread\ creation} + t_{object\ inflation} + t_{synchronization}$. This model assumes that thread creations, object inflations, and synchronizations are the only significant differ-

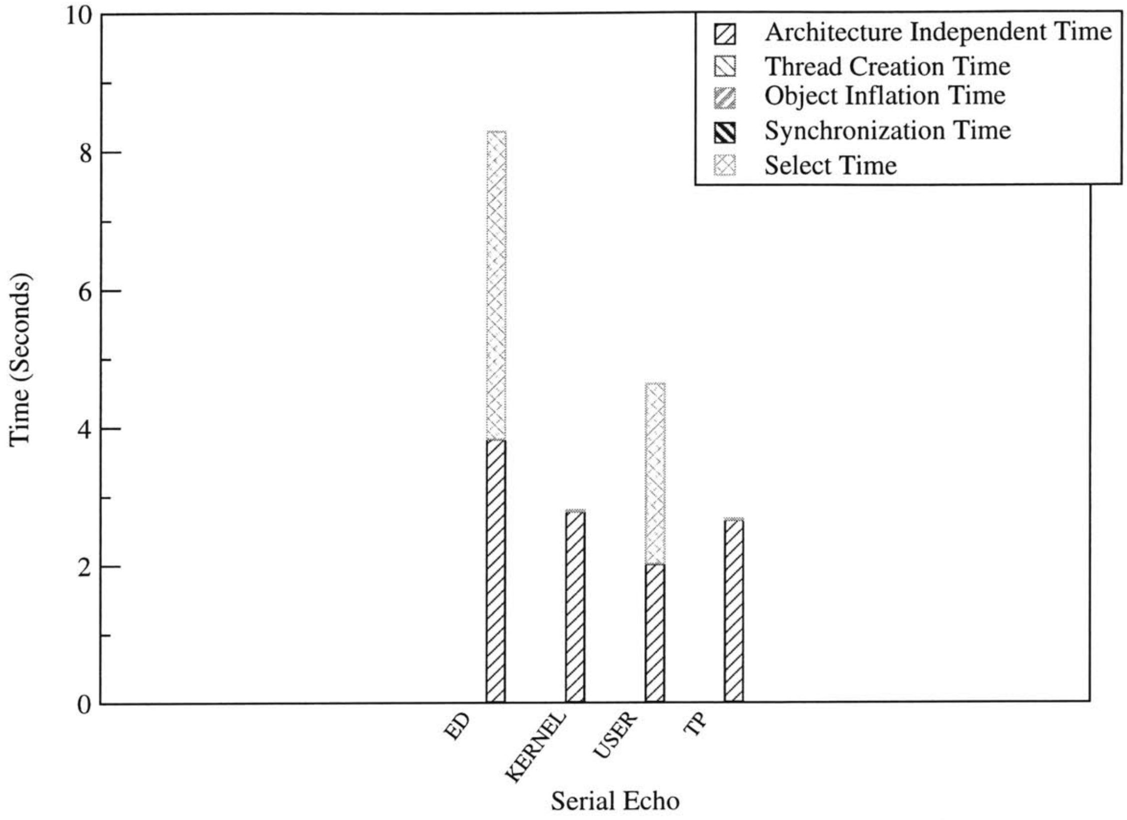

Figure 4-4: Serial Echo Benchmark with Select Overheads

ences between the different architectures.

The *complete model with select* performance numbers are for a model described by $t_{architecture\ dependent} = t_{thread\ creation} + t_{object\ inflation} + t_{synchronization} + t_{select\ processing}$. This model adds the overhead of select processing into the complete model performance numbers. We only use this model for the serial echo benchmark.

We present the coefficients of variation in Table 4.3. The reader may note that increasing the sophistication of the model does not always yield better results. This is because some overheads are not significant for some benchmarks. For example, the long connection echo benchmark opens its connections before the benchmark starts and does very little synchronization. Adding more details to the model when they are not significant for the benchmark is unlikely to yield better results. Some of the results shown in Table 4.3 reflect this.

The model works very well for the short echo benchmark, the time benchmark, the chat

Server	No Model	Thread Creation Model	Object Inflation Model	Complete Model	Complete Model w/ select
Limited Parallelism Echo	0.026	N/A	0.029	0.030	N/A
Long Echo	0.050	N/A	0.052	0.052	N/A
Short Echo	0.370	0.065	0.035	0.028	N/A
Time	0.380	0.091	0.062	0.057	N/A
Chat	0.034	0.033	0.032	0.034	N/A
Quote	0.158	0.157	0.152	0.144	N/A
HTTP	0.107	0.052	0.038	0.032	N/A
Game	0.177	0.176	0.172	0.170	N/A
Phone	0.199	0.095	0.072	0.067	N/A
Serial	0.571	N/A	0.573	0.574	0.268

Table 4.3: Performance Model Evaluation

benchmark, the http benchmark, and the phone benchmark. It works reasonably well for the game and the quote benchmark. Even after corrections for select processing, the model still works poorly for the serial echo benchmark.

The complete model predicts that the limited parallelism echo benchmark and the long echo benchmark have little dependence on server architecture, and the nearly identical execution times across the various server architectures confirms this.

4.10 Discussion of Results

From our benchmark results, it is clear that no single architecture is always better than the others. However, for various classes of applications, some architectures have advantages.

For applications that establish short connections such as web servers or other request and response based information servers, architectures with very small thread creation overheads are optimal. Event-driven and thread pooled servers appear to be the best choice due to their low or non-existent thread creation cost.

For applications with persistent connections that require little synchronization such as our long connection echo benchmarks, it appears that no server architecture has a clear advantage.

For applications with persistent connections with lots of synchronization, such as a chat server, architectures with low synchronization overheads are optimal. The event-driven or user-level thread implementation works best for this class of applications.

Applications with very few active connections do not perform well with architectures that use select polling. They end up running some very small portion of application code, they run out of computation to do, and then run some relatively expensive select processing routines. For this reason, native thread implementations are the best choice in this class of applications.

The other important lesson to keep in mind is that the relative importance of various overheads depends on client behavior. It is likely that thread creation overheads would become much more important in the chat, quote, and game server if the clients made short lived connections to the servers. And it is reasonable to think that these different connection patterns could occur when people deploy the servers.

We can fault the UNIX operating system for part of the poor performance of the event-driven servers in cases like the serial echo benchmark. The next section discusses problems with operating system support for event-driven servers.

4.11 Operating System Support for Event-driven Servers

UNIX falls short in providing efficient support for event-driven servers. One problem is that non-blocking I/O calls may actually block for file reads. The AMPED[9] architecture attempts to work around this flaw.

Another flaw is that the only reasonably scaleable way for event-driven servers to check for events in UNIX is polling select. The time select takes to run is proportional to the number of file descriptors scanned, and not the number of ready file descriptors returned. Some work has been done in providing scaleable implementations of select[7]. However, the interface that select uses inherently scales poorly[6]. Banga, Mogul, and Druschel reduce select/event delivery overhead for a proxy web server from 33.51% of the execution time to less than 1% by switching to an explicit event delivery mechanism. Using an explicit event delivery mechanisms like the one developed by Banga, Mogul, and Druschel would likely result in greatly improved performance for the event-driven servers.

Chapter 5

Conclusion

This project considers an event-driven server architecture, a thread-per-connection architecture with kernel-level threads, a thread-per-connection architecture with user-level threads, and a thread-pooled server architecture. We presented an empirical model for the performance of these architectures derived from thread creation, synchronization, object inflation, and select processing overheads.

We presented a novel thread-per-connection to event-driven transformation utilizing a modified partial continuation passing style conversion in this report and empirically evaluated it. We compared this server architecture to a more traditional thread-per-connection server architecture and a thread-pooled server architecture.

The event-driven architecture very efficiently handles thread creation and enables the elimination of synchronization and the corresponding object inflation in many cases. However, the event-driven transformation introduces a large select processing overhead. I/O patterns that do not allow the server to amortize this cost over many connections result in very poor performance from the event-driven architecture. The transformation results in performance increases up to a factor of 2 and performances decreases up to a factor of 3.

The factor of 3 slowdown is partly due to poor operating system support for event-driven servers. Others do research on more efficient I/O primitives for event-driven servers. An event-driven server using an improved operating system interface would probably not see this severe slowdown.

The performance model considered in this report is a simplistic model, only accounting for the differing costs of various actions under the different architectures. It does not account for second order effects such as differences due to memory management. But it explains the benchmarks results quite well.

Further research remains in explicitly handling continuation memory management, examining different event-driven architectures, doing a more complete performance model, examining different operating system interfaces, and using SMP continuation schedulers.

Bibliography

[1] Apache web server. http://www.apache.org.

[2] Flex compiler. Available from http://flex-compiler.lcs.mit.edu/.

[3] Netscape portable runtime (nspr). http://www.mozilla.org/projects/nspr/index.html.

[4] Ole Agesen. The cartesian product algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[5] Andrew Appel. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, pages 47–74, January 1996.

[6] G. Banga, J. Mogul, and P. Druschel. A scalable and explicit event delivery system for UNIX. In *Proceedings of the Usenix 1999 Annual Technical Conference*, June 1999.

[7] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[8] Alan Demers Mark Weiser and Carl Hauser. The portable common runtime approach to interoperability. In *ACM Symposium on Operating Systems Principles*, December 1989.

[9] V. Pai, P. Druschel, and W. Zwaenepol. Flash: An efficient and portable web server. In *Proceedings of the Usenix 1999 Annual Technical Conference*, June 1999.

[10] Chris Provenzano. Mit pthreads. http://www.humanfactor.com/pthreads/mit-pthreads.htm

[11] Sun. Java. http://www.javasoft.com.

[12] Volano LLC. Volano chat server. http://www.volano.com.

[13] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2000.

[14] Xavier Leroy. The linuxthreads library. `http://pauillac.inria.fr/ xleroy/linuxthreads/index.html`.

[15] Zeus Technology Limited. Zeus web server. `http://www.zeus.co.uk`.