

# Improving Java Server Performance with Interruptlets

David Craig, Steven Carroll, Fabian Breg, Dimitrios S. Nikolopoulos, and  
Constantine Polychronopoulos

Center for Supercomputing Research and Development  
University of Illinois  
1308 W. Main St, Urbana IL 61801, USA  
phone: +1.217.244.4654  
{dcraig, scarroll, breg, cdp}@csrds.uiuc.edu

**Abstract.** With the widespread usage of the Internet, the need for high throughput servers has greatly increased. The Interruptlet system allows Java server application writers to register light-weight interrupt handling routines (written in C or Java). The underlying system architecture is designed to minimize redundant copies between protection domains and thread overhead involved in I/O handling in the JVM on Linux.

## 1 Introduction

The World Wide Web (WWW), was originally designed as a global document retrieval infrastructure built on top of a network of networks, called the Internet. Documents stored on any node of the WWW can be transferred to any requesting node, using the HyperText Transfer Protocol (HTTP). Documents written in the HyperText Mark-up Language (HTML) can form a system wide interrelated information system that can be retrieved by mouse clicks from a web browser running on a user's desktop.

With the introduction of the Java programming language [3] the web broadened its horizons by allowing more interactive content to be embedded in its documents. Small Java applications, called applets, can now be downloaded to and executed on the same web browser used to access other web documents.

As the Java language grew, so did its attraction as a more general purpose programming language. Many Internet server tasks are now being implemented in Java. Java Servlets provide a mechanism to extend the functionality of web document servers to provide complete e-commerce or virtual community services over the web. Servlets have the potential to replace the more traditional CGI applications, which are less portable and can be harder to program.

Java Servlets are most efficiently employed from within a Java based web server. The Java Virtual Machine (JVM)[4] running the web server can more seamlessly run Servlets than C coded web servers. However, Java based server applications still suffer from low performance<sup>1</sup>. Java based servers, that need to handle high loads, typically employ a large number of threads, introducing a

<sup>1</sup> <http://www.volano.com/report.html>

large overhead. This, combined with high overhead introduced by data transfer from the network device through the operating system to the application, yields less than adequate performance.

This paper presents two contributions. The first is a new form of asynchronous I/O for Linux called an *isocket* (Interruptlet socket) which eliminates much of the context switch and system call overhead involved in the reading and writing of data on I/O ports. The benefits isockets can be exploited by the standard Java I/O library. The second contribution is the concept of a Java *Interruptlet* which leverages the isocket to allow direct interrupt handling code to be registered and invoked for incoming I/O. The handler code interfaces directly with the Virtual Machine and allows the quick handling of requests in a server application.

In Section 2 we describe the design goals for our proposed architecture. Section 3 introduces the design of our architecture. Section 4 shows how our approach can be applied in a typical Java based server application. Section 6 compares our approach to other approaches to improve Java I/O performance. Section 7 presents some future work involving Interruptlet. Section 8 concludes this paper.

## 2 Design Goals

The current growth in size and popularity of the World Wide Web creates the need for ever more powerful servers to handle more client requests. Faster I/O support is needed to meet this demand, but ideally this faster I/O should be provided in an easily integrated fashion so that existing server software can be improved with little effort. The I/O overhead problem in Java is worse than a C environment because of the added layers of abstraction provided by the virtual machine increase the overhead for each transaction.

In monolithic operating systems, whenever data arrives at the networking device, a DMA transfer is setup to copy that data into a socket buffer in kernel space. Next, the kernel wakes up the process(es) waiting for data on that socket. After a process is woken, it has to transfer the data from kernel memory into its own, requiring expensive protection domain checks and possible page faults. Next, the virtual machine has to wake the thread that is waiting for that data, which requires expensive thread context switching and contending for the virtual machine scheduler lock. A handle to the data which is now in user space is given to that thread which can begin processing.

The motivation for our design was reducing the number of context switches and system calls necessary in a typical transaction for a server application written in Java. Secondly, we wanted to add fast interrupt handler-like routines to Java without exposing the details of the operating system or other non-portable details to the server application written in Java.

### 3 Interruptlets Design

In this section, we describe our proposed architecture for improved signal handling in Java programs. We will first give a high level overview of the complete process involved, and then explain each component of the architecture in more detail. We will also outline the changes to the JVM that were necessary to support Interruptlet execution. An overview of the architecture is shown in Figure 1.

An Interruptlet is a Java class with a routine to handle a particular type of I/O request. The object is registered with the JVM which then redirects I/O requests to that handler. Currently, short network requests are the primary requests handled by Interruptlets. These are requests such as retrieving a static HTML page or updating counters for the number of bytes received and sent. The primary class of applications that will benefit from Interruptlets is server applications that frequently perform operations that complete in a 100's of cycles.

For example, in the web server application that we will describe in the next section, a commonly occurring set of static web pages is often served. These pages can be cached and served quickly as soon as a request is received by the Interruptlet. Any request that can be handled by the Interruptlet's handling routine is said to have taken the *fast path*. If the Interruptlet cannot handle the request, the request is queued and normal program operation for handling that type of request is resumed. This is referred to as the *slow path*.

To provide fast interrupt delivery from the operating system to the Interruptlet, we add a data structure called an isocket to the Linux kernel. An isocket allows a user process to read incoming data from an area in memory that we call the I/O *shared arena*. The I/O shared arena is a segment of memory shared between the kernel and a user process, like the JVM. The shared arena concept was first used as part of the nanothreads implementation [2]. Because the page is locked in memory and its location is known to both the kernel and JVM, it can be used to exchange information via reads and writes. This avoids the heavy overhead involved in a system call to read data from kernel memory.

The complete Interruptlet architecture consists of a modified Linux kernel, a modified version of the JVM, and a user level Interruptlet library. We will now describe each of these components.

#### 3.1 Linux Kernel Modifications

An isocket is derived from a traditional Linux socket. In a traditional socket, the data that is received is copied from kernel memory to user memory with a system call after the user process is awakened. In an isocket, the data is instead copied network interface and the I/O Shared arena as indicated at location (1) in Figure 1.

If the user process (JVM) is running when new data arrives, the process is sent a SIGIO interrupt (2). Upon receiving this interrupt, it checks the I/O shared arena for the data and receives a pointer to it if there is data available (data may not end up in the arena if it was full). This process is called a fast read

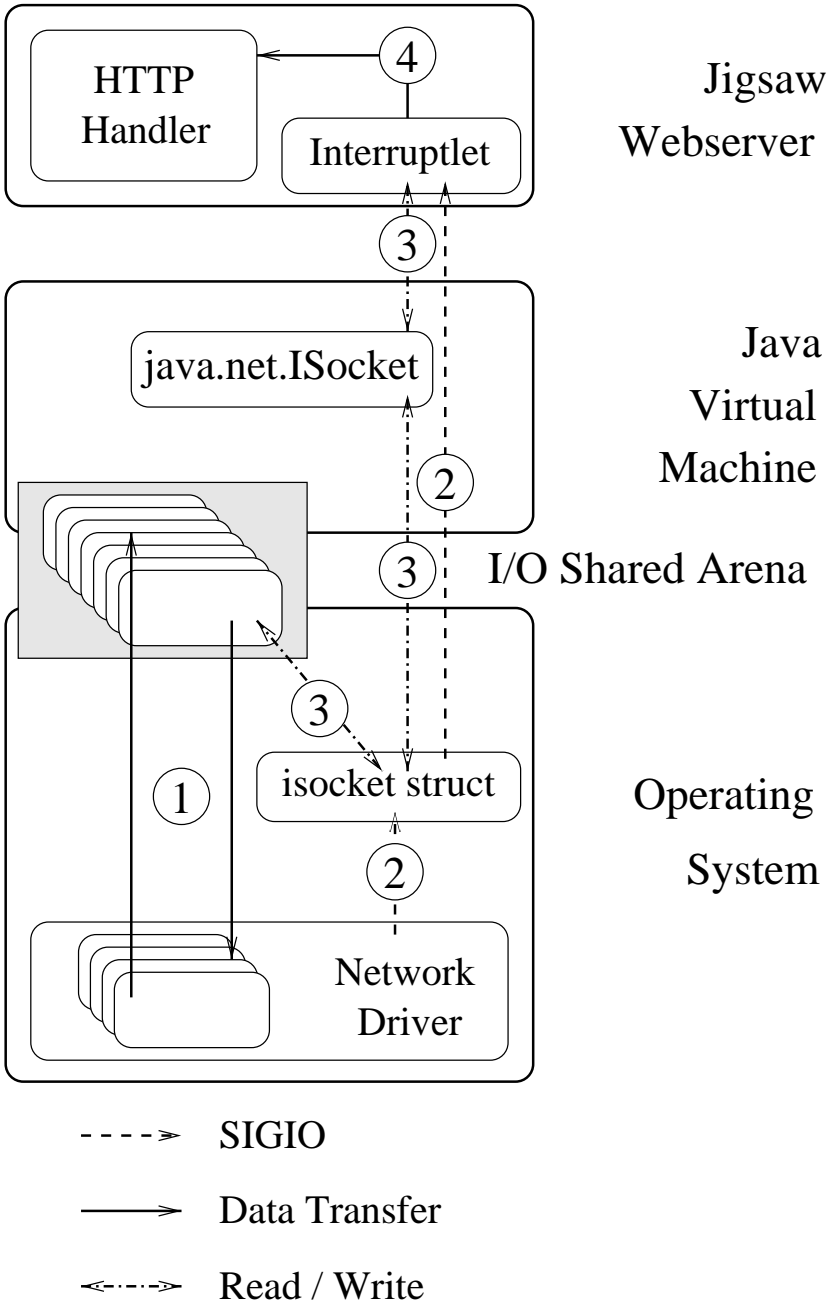


Fig. 1. Interruptlet Architecture

(3) and it saves a system call and the subsequent data copy that is implicit in a normal socket read operation. The data copy time can be hidden with multiple processors, since the copy is no longer coupled with the read call. In addition to reducing the copy overhead, we borrow an already running JVM thread to run the Interruptlet to save the overhead of awakening a thread to handle the I/O as done in the unmodified JVM. The JVM passes the pointer to the data in the I/O shared arena directly to the Interruptlet and invokes its handler routine with that data pointer as its argument. Note, that the Interruptlet code masks all other interrupts while it runs, which poses some restrictions on Interruptlet code, which we will describe in the next section.

Because the JVM can be interrupted by the OS at any time, special care has to be taken when writing the interrupt handler itself. The garbage collector may have been in the process of moving an object when the interrupt occurs, so accessing such objects requires special attention.

The isocket data structure has an identical interface to Linux sockets with two additions: `read_fast()`, and `write_fast()`. The standard socket `read()` and `write()` calls trigger the normal copy from the kernel buffer to the user buffer. In the isocket version, the data is stored in the shared arena on arrival and a ready bit is set. When the `read_fast()` function is called, it polls the ready bits for all of the buffers in the shared arena and selects the first buffer it finds that is ready. A pointer to this buffer is returned. A `write_fast()` operation reserves a buffer in the shared arena and then copies the data to be written there. The kernel then copies it directly to the network adapter from the shared arena (1).

### 3.2 JVM Modifications

The JVM needs to provide a mechanism to allow Java applications to create isocket structures. A copy of the existing socket classes (`java.net.Socket`, `java.net.ServerSocket`, etc.) was created whose native functions map the Java operations to isockets instead of sockets. In addition, isocket based implementations of the `java.net.SocketInputStream` and `SocketOutputStream` are provided. In addition, the JVM must be set up to invoke application provided Interruptlets whenever it receives a signal from the kernel to do so. This was accomplished by modifying the standard JVM I/O handler to check for registered Interruptlets before running the old handler. Finally, it is necessary to provide a mechanism for registering the Interruptlet itself. The Interruptlet class library provides a static (native) routine for registering an Interruptlet for a particular type of I/O.

### 3.3 User Level Interruptlets

The final component of the Interruptlets architecture is the language level Interruptlet classes that the user can subclass in order to write handler routines for I/O. The Java Interruptlet object must be of a type that extends the `Interruptlet` class:

```
public abstract class Interruptlet {
    protected Interruptlet(IServerSocket s);
    public abstract byte[] handleInterrupt(byte[] data);
}
```

The `Interruptlet` constructor takes care of registering the `Interruptlet` for the communication port associated with the supplied server socket. The JVM invokes the `handleInterrupt()` method whenever the networking device signals the arrival of data. The data parameter contains the incoming data on the networking device. The reply generated by the `Interruptlet` will be returned in a byte array by the `handleInterrupt()` method. When data is received and the `Interruptlet` is invoked, the data that is returned by the handler is then immediately sent to the requester.

If the `Interruptlet` cannot handle the request in a sufficiently short time span, the request is deferred to the slow path by returning a null pointer instead of a byte array handle. The slow path (4) must be made runnable by the `Interruptlet` by notifying a new thread of control to handle the request. In other words, the slow path is equivalent to how the request would have been handled in the absence of the `Interruptlet`. Since all interrupts are masked during the completion of an `Interruptlet`, it is key to quickly return from the `Interruptlet` handler as soon as possible if the request cannot be handled.

## 4 Application: Web Server

One of the most important servers found on the Internet is the web server. A web server uses the HyperText Transfer Protocol (HTTP) to serve a wide variety of documents from the server's host to requesting clients. For heavily visited sites, it is important for a web server to maintain a high rate of serviced documents and to provide clients their expected response times.

The typical mode of operation of a web server, like that of many other server applications, is to have a main routine listen and accept connections from client applications and then to spawn a new thread to handle this request. Handling a typical request involves reading and interpreting the request, finding or constructing the requested document, and finally generating a response containing the document and return it to the client.

Jigsaw<sup>2</sup> is a fully customizable web server developed by the World Wide Web Consortium<sup>3</sup>. Jigsaw includes support for CGI and Servlets and is completely written in Java. It largely follows the mode of operation described above.

We plan to add an `Interruptlet` based cache facility to the Jigsaw web server to allow more efficient handling of static HTTP requests. Based on observations of real world web server statistics, the designers of the SpecWeb99<sup>4</sup> benchmark suite estimate that 70% of all HTTP requests are static HTTP requests.

<sup>2</sup> <http://www.w3.org/Jigsaw/>

<sup>3</sup> <http://www.w3c.org>

<sup>4</sup> <http://www.specbench.org/osg/web99/>

Requests that can be satisfied by this cache constitute an Interruptlet fast path as described in the previous section. The complete architecture of our Interruptlet enhanced Jigsaw web server is shown in Figure 1. The slow path, using the default Jigsaw request handling routines, is taken if the requested document is currently not in the cache.

The Interruptlet cache is a simple Least Recently Used (LRU) cache implemented in native C code that is linked with Jigsaw using the Java Native Interface (JNI). It stores document contents associated with simple HTTP GET request strings. A Java interface to the cache is provided to allow the Interruptlet slow path to update the cache, in addition to the updates from the fast path. We will discuss fully Java written Interruptlet routines when discussing future work in Section 7.

The main loop (accept connection and spawn new handler thread) has to be modified to make our Interruptlet mechanism work. Instead of creating a standard Java `ServerSocket`, we create an `IServerSocket` and register our Interruptlet with it. In addition, we create a pool of handler threads that stand by to execute slow path requests. It is important to note that the modifications necessary to adopt the Jigsaw web server to work with Interruptlets were minimal. The slow path of the server simply reuses the code that was already written for normal handling. The ability to reuse the core of the code is one of the strengths of our approach.

## 5 Preliminary Results

The Interruptlets system is still in the implementation phase. However, we have conducted experiments to characterize the performance of the isockets part of the design. The test application was a simple echo server that accepts messages of varying lengths and then echoes the same message back to the sender. Two versions of the echo server were created: one version with a standard Linux socket implementation (Linux 2.2.19) and one version with isockets. The isocket version does not currently use the Interruptlets ability to run on the currently executing thread.

In the first experiment, the client and server were run on the same machine, a 4 processor Dell server with Pentium Pro 200Mhz processors. We recorded the time between the client sending the request to the server and the client receiving the reply. The isocket version was about 5% (20–50  $\mu$ s) faster than the version with normal sockets. Next, the client and server were run on separate machines connected by a cross Ethernet cable. The isocket version was 3% (0–30  $\mu$ s) faster.

The performance gain is due to the removal of the buffer copies from the critical path. In the normal sockets version, data arrives at the NIC triggering an interrupt. The kernel copies the data from the NIC to kernel space and then notifies the user application that the data is available. The user application then issues a read system call which copies the data to user memory. In the isockets version of the echo server, the interrupt arrives and the kernel copies the data to the shared arena. The kernel notifies the user application that the data is

available and the user application reads it directly from the shared arena. In short, there is one less copy and one less system call.

We expect significant gains in throughput (in terms of number of client requests) from the increased level of concurrency available from the I/O shared arena. The I/O shared arena enables the overlapping of the copying of incoming data to user space with the processing of data in the server. Additional gains are expected from the use of Interruptlets, which reduce the number of context switches required to handle data and reply to the clients.

## 6 Related Work

The POSIX standard provides its own functions that implement asynchronous I/O. An application calls `aio_read()` to read from a file descriptor, but does not want to be blocked. The application can choose to be signaled when the read operation can be completed or to wait for incoming data with a blocking call at a later time. In standard Linux, this functionality is implemented using separate threads to handle the request. SGI's KAIO<sup>5</sup> uses a more efficient split-phase I/O, where the request is queued at the I/O device.

The Non Blocking I/O (NPIO) library<sup>6</sup>, part of the Sandstorm project [8], provides non blocking I/O facilities to Java applications. NPIO is implemented as a JNI wrapper around the native non blocking I/O facilities `select()` and `poll()`.

Interruptlets also provides a mechanism to asynchronously handle incoming data within a Java application. In addition, `isockets` allow a more efficient propagation of the incoming data through the kernel to the application, by using the I/O shared arena to reduce context switching and protection domain checking overhead.

IO-Lite [6] has a similar approach to our shared arena to minimize unnecessary and redundant buffering and copying. They have a single copy of each I/O buffer that does not need to be copied from kernel space to user space, but they use a read only buffer system instead of a locked page specifically for communication. However, IO-Lite does not reduce context switching overhead.

U-Net provides a zero-copy interface and was the basis for the Virtual Interface Architecture (VIA). It achieves this by providing a user-level interface to the network adapter that allows applications to communicate without operating system intervention. The main distinction between U-Net and Interruptlets is that Interruptlets also tries to eliminate context switching overhead by hijacking the currently running thread to service the interrupt. Also, the combination of the reply with the return from the Interruptlet is unique to our design.

The Flash [5] web server combines a single threaded event driven architecture for cached workloads with a multithreaded architecture for disk-bound requests. Exploiting IO-Lite in their Flash web server improved its performance with

<sup>5</sup> <http://oss.sgi.com/projects/kaio/>

<sup>6</sup> <http://www.cs.berkeley.edu/~mdw/proj/java-npio/>

40-65%. Redhat TUX web server<sup>7</sup> obtains high performance by moving HTTP handling into the Linux kernel. Our approach is more general in that isockets and Interruptlets are concepts that could be exploited in a wide range of server applications.

## 7 Future Work

A prototype Interruptlet system is currently being developed and performance characterization is the next logical step. The most important work will be characterizing the length of time the interrupt handler routine can be allowed to execute before the masking of interrupts degrades server performance and reliability. Because performance of the handler routine is critical, the Interruptlet handler code should be statically compiled in advance and dynamically linked at registration time. At present, this is accomplished by writing the cache and handler routine in C using JNI. In the future, these functions will ideally be written in Java and aggressively compiled by a static Java compiler (such as GCJ [1] or the PROMIS compiler system [7]).

## 8 Conclusion

The Interruptlet system provides Java programs with the ability to register interrupt handler routines (written in C or Java) and link them seamlessly with their Java server applications. The isocket subsystem provides improved I/O for the Java Virtual Machine by eliminating the redundant copying of data between the network adapter, kernel space, and user space. The system eliminates thread wake-up overhead by borrowing the current running JVM thread to run the handler routine. By writing server applications to make use of the Interruptlet user library, the programmer is, in effect, writing completely portable interrupt handlers.

## References

1. P. Bothner. A Gcc-based Java Implementation. In *IEEE Comcon*, February 1997.
2. D. Craig and C. Polychronopoulos. Flexible User-Level Scheduling. In *Proceedings of the ISCA 13th International Conference on Parallel and Distributed Computing Systems*, August 2000.
3. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley Developers Press, 1996.
4. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley Developers Press, 1996.
5. V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. of the 1999 Annual Usenix Technical Conference*, Monterey, CA, June 1999.

---

<sup>7</sup> <http://www.redhat.com/products/software/ecommerce/tux/>

6. V.S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proc. of 3rd Usenix Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.
7. H. Saito, N. Stavrakos, S. Carroll, C. Polychronopoulos, and A. Nicolau. The Design of the PROMIS Compiler. In *Proceedings of the International Conference on Compiler Construction*, March 1999. Also available in “Lecture Notes in Computer Science No. 1575” (Springer-Verlag).
8. M. Welsh, S.D. Gribble, E.A. Brewer, and D. Culler. A Design Framework for Highly Concurrent Systems. Technical report, Computer Science Division, University of California, Berkeley, April 2000.