Synchronization of Java Threads Using Rendezvous

James Shin Young - *jimy@eecs.berkeley.edu*

EE244/EE290N Project Report
December 9, 1996
Source code updated January 7, 1997

This document is still evolving. Any comments or suggestions are more than welcome.

ABSTRACT

Java is an attractive programming language due to its many features, including automatic memory management, lack of pointer arithmetic, and its support of concurrency, all of which help make it an arguably better language than C or C++, on which Java is loosely based. While Java's concurrency features, in the form of its threads package, are powerful, they can also be easily misused. Uneducated use of Java's threading features may lead to non-determinate programs, which arises from arbitrary ordering of interacting events in separate threads. Non-determinism may be eliminated by imposing an order on all such interacting events.

Java's synchronization primitives are sufficient to achieve a wide range of thread interaction behavior, but not well-suited in their raw form for use in imposing ordering on events in different threads. Approaches to this problem that directly use the synchronization primitives likely will be erroneous or inefficient if implemented by novices in concurrent programming. One solution to this problem is to provide higher level synchronization abstractions for the users.

Rendezvous, a mechanism for imposing an ordering on events in an arbitrary number of concurrent threads, was developed for Java. The rendezvous is shown to be useful in synchronous applications, where ordering of certain events between threads is necessary to ensure determinism. Relative to a comparable implementation using one of the standard Java synchronization methods, rendezvous is easier to use and gives better program performance.

INTRODUCTION

Java is an object-oriented programming language developed by Sun Microsystems, released to the public in May 1995. It was intended originally for use in embedded consumer applications, but has found the greatest amount of success as a language for networked applications. Although Java is very similar in syntax to C and C++, it contains many features lacking in these languages which make Java significantly different. The most important of these features are automatic memory management, lack of pointer arithmetic, and standardized support of concurrency. In this paper we address some issues raised by the last feature, concurrency.

Java's support for concurrency is in the form of its standard threads library. It gives the user the ability to easily fork new threads, join with other threads, as well as to stop or suspend them. In addition, Java has a built-in thread synchronization mechanism based on monitors. While the primitives provided are very powerful, in that they may used to implement a wide range of behaviors, they can also be easily misused and lead to incorrect, non-determinate programs.

Thus, it is critical that the threads package in Java be used intelligently and carefully. However, it is unreasonable to expect all Java programmers to be experts in concurrent programming. Perhaps this will become a reality in the future, but with the current generation of programmers trained in C and C++, other solutions should be found to encourage safe use of Java threads.

The approach espoused in this paper is that of layering abstractions onto Java threads to create "higher" order

concurrency mechanisms that can be used with lesser chance of error. In particular, we demonstrate how a seemingly simple use of Java's synchronization primitives can lead to non-determinacy and erroneous programs. We then show how a rendezvous mechanism can be of great general utility, decrease the likelihood of programming errors, and how it can increase program performance.

MOTIVATION

In single-threaded applications, program correctness is usually independent of the timing of its execution, and determined solely by the logical correctness of its functions. However, in multithreaded applications it is often the case that execution timing may have a great affect on the correctness of the program. Given one set of inputs, many different outputs might be produced, depending on how the threads are interleaved by the scheduler. Such programs with multiple possible behaviors are called *non-determinate*, and are in general undesirable*. Non-determinate programs can also be difficult to debug, as the offending behaviors may not be readily reproducible, and the use of a debugger may in fact modify its behavior.

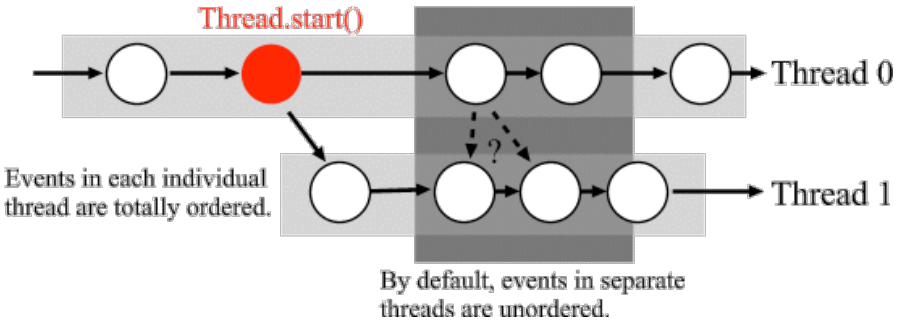

Figure 1. Ordering of events dictated by Java threads.

To address the issue of non-determinacy in a multithreaded Java program, we must first establish a framework in which to discuss the environment created by the Java language. To begin, we take the semantics of a Java program to be simply a *partial order* on *events*. *Events* are the instructions that must be executed in order to carry out the functionality dictated by the text of the program. One can take the events to be the sequence of bytecodes executed on the Java virtual machine, but this detail is not critical to our discussion. The events in the program are initially totally ordered with respect to one another, until a call to the `Thread.start()` method is executed. At this point, the subsequent events in the initial thread are not ordered relative to the events dictated by the `run()` method of the new thread which was just started. This relation is shown graphically in Figure 1. In addition, we consider the program's *state* to be defined by the contents of all of its variables, and the "program counter," which keeps track of the next event to be executed within each thread.

In comparison with other concurrent languages, Java can be characterized as an asynchronous language. Thus, it is similar to languages such as Ada, Occam, and CSP [1], and quite different from synchronous languages, such as Esterel, Argos, and Lustre. In asynchronous languages an arbitrary amount of time may pass while one concurrent process (i.e., thread) waits to communicate with another, while in synchronous languages communications are thought of as "instantaneous" [2]. In practice, this means that the concurrent processes in asynchronous languages operate somewhat independently from one another, while in synchronous languages the processes operate together in lockstep. Practically, a fundamental difference between them is that asynchronous languages support non-determinism, while synchronous languages do not.

Non-determinism in Java arises fundamentally from unspecified ordering of interdependent events in concurrent threads. For example, in Figure 2, a path of communication between two threads exists through a shared variable `x`. Thread 0 reads the value from `x`, while thread 1 writes a value to `x`. However, unless measures are taken to fix the order in

which these threads access x, the resulting program might be non-determinate. Sometimes thread 0 might read x before thread 1 writes x, other times thread 1 will write x before thread 0 reads x. Both behaviors are acceptable interpretations of the Java program. Unless the program has been designed to take both these two possible orderings into account, the resulting program behavior will be non-determinate. Therefore, to eliminate non-determinacy, one must identify all possible inter-thread interactions, and impose an ordering on all pairs of interacting events.

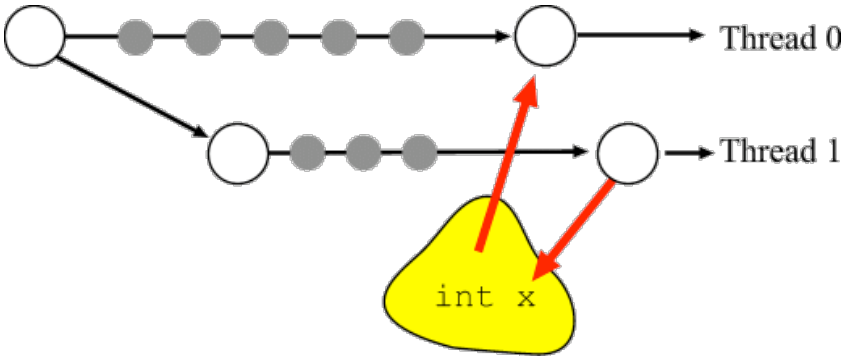

Figure 2. Non-determinacy arises from unspecified ordering of events.

We first investigate the synchronization primitives provided by Java, and how they may be used to impose ordering constraints between events in separate threads. Then a rendezvous mechanism for Java is introduced, and is shown to be easier to use, more scaleable, and faster than other techniques.

JAVA'S SYNCHRONIZATION PRIMITIVES

In order to allow the programmer to impose ordering on events between threads if so desired, Java provides a synchronization mechanism based on monitors. The features are invoked through the keyword `synchronized`, and the methods `wait()`, `notify()`, and `notifyAll()`.

Each object in Java has a lock and a monitor to manage the lock. By naming one of the object's methods `synchronized`, this forces any thread wishing to invoke this method to acquire the lock for this object first. The lock is released by the thread once it returns from the method. Note that a thread can hold locks to many objects at once. The thread will block until it can get the lock if another thread already holds the lock to that object. In short, `synchronized` is useful to ensure mutual exclusion of groups of events. For example, Figure 3 shows two threads that both execute calls to synchronized methods in object A. While thread 1 executes `foo()`, thread 0 reaches a point where it wishes to call `bar()`. However, since both `foo()` and `bar()` are declared synchronized, thread 0 must block until thread 1 has exited from `foo()`. Subsequently, if thread 1 wishes to call method `fooBar()`, it must wait until thread 0 has exited from `bar()`. One important thing to note is that synchronized does not impose ordering constraints. In the example, another equally likely scenario is for thread 1 to execute both `foo()` and `fooBar()`, before thread 0 reaches `bar()`. Therefore, another mechanism is required to impose ordering between events in separate threads.

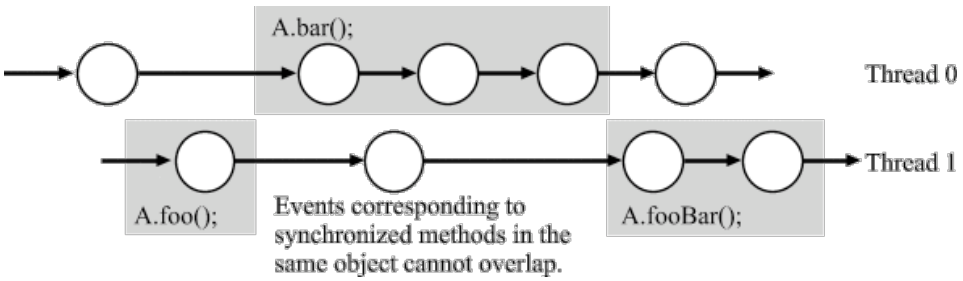

Figure 3. Synchronized ensures mutual exclusion of sequences of events.

Ordering on events in separate threads is imposed by the use of the `wait()`, `notify()`, and `notifyAll()` methods. While the thread has the lock of an object, it may call the object's `wait()` method. This atomically suspends the thread and releases the lock. The thread is added to the object's "wait set" [3], which contains a list of the threads waiting on the object. Subsequently, when another thread acquires that object's lock, that thread may elect to call either the `notify()` or `notifyAll()` methods. A call to `notify()` will choose one thread from the wait set in an unspecified manner (i.e., the wait set is not necessarily a FIFO), and resume its execution. A call to `notifyAll()` on the other hand resumes all the threads in the wait set. Thus, events which occur after a call to `wait()` in one thread must all happen after the call to the corresponding `notify()` or `notifyAll()` in another thread. This allows ordering constraints to be placed on events in separate threads.

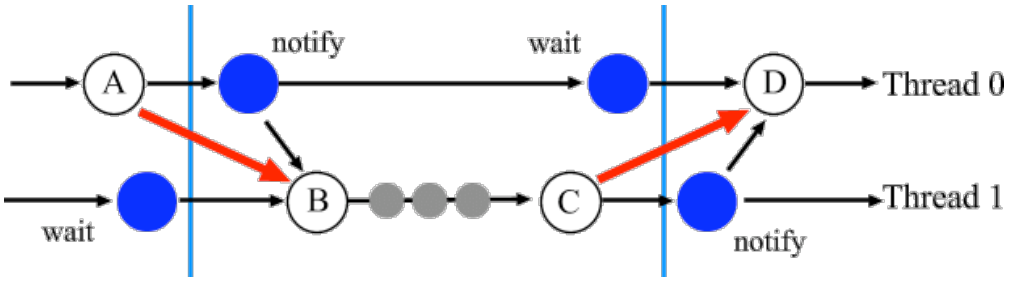

Figure 4. Naïve use of wait(), notify().

Although the behaviors of `wait()`, `notify()`, and `notifyAll()` are quite simple, using them correctly requires the programmer to be aware of some subtleties. Naïve use of these synchronization primitives can often result in non-deterministic programs. To illustrate an example of such a pitfall, consider the example shown in Figure 4.

Assume that in order to ensure determinacy, it is required that event A precede event B, and C precede D. To achieve this effect, the programmer synchronizes the threads on a common object. To ensure event B occurs after event A, a wait() is placed before A, and notify() placed after A. Therefore, as long as this same object is not used by other threads in the program for synchronization or at other times in threads 0 and 1, event B must occur after A. The same technique is then applied to order C and D. However, this implementation does not always achieve the desired result.

Figure 5 shows an example of how this implementation is non-deterministic and faulty. Assume that the first `wait()`/`notify()` pair has executed as desired, successfully ordering A and B. However, if thread 1 subsequently executes the call to `notify()` before thread 0 calls `wait()`, the `notify()` is "lost," and thread 0 will block forever, which was not the desired effect. Thus, ordering of events between threads is not accomplished by simple use of `wait()`, `notify()`, and `notifyAll()`.

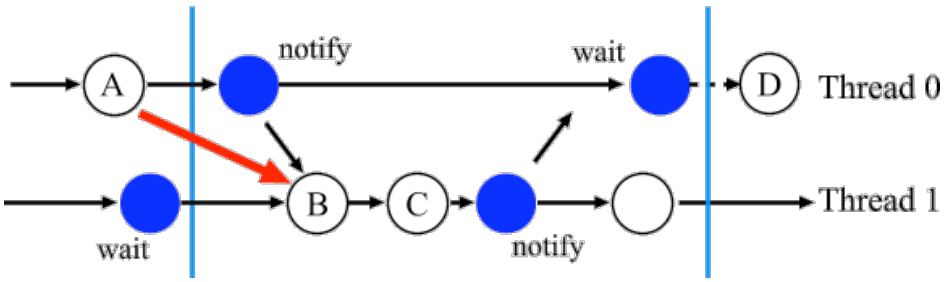

Figure 5. Sequence of events leading to deadlock.

One possible solution is to enclose each call to `notify()` within a loop, and implement some type of handshaking protocol between the threads when the `notify()` is received by the `wait()`. However, this style of implementation is quite suboptimal from several perspectives. First, an arbitrary number of `notify()` calls might be made before a response is received, thus expending extra system resources on

synchronization overhead. Second, this protocol does not easily extend to an arbitrary number of threads. Finally, it is burdensome to the programmer to implement by hand each time a synchronization is needed.

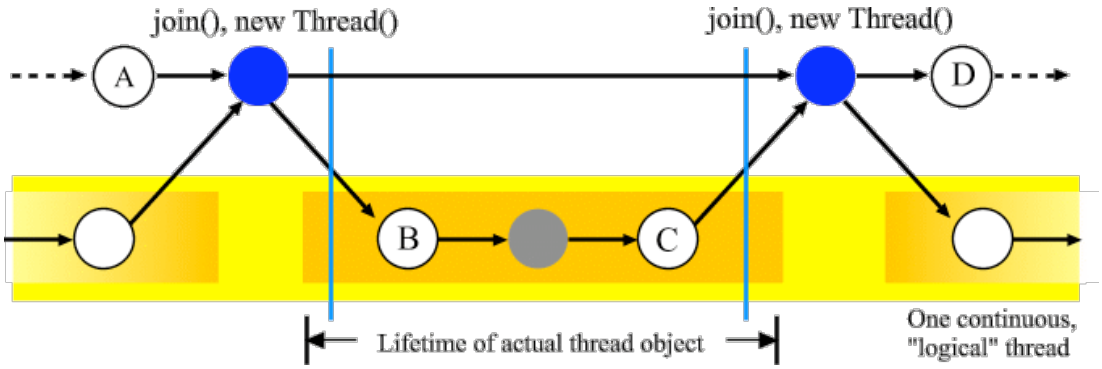

Figure 6. Synchronization using new Thread(), join().

The one standard method in Java which does impose ordering constraints in Java is `Thread.join()`. If one thread calls the `join()` method of a second thread, it ensures that all the events of the callee thread have been executed before the events following the `join()` call in the caller thread. To achieve the clock-type behavior we desired above, one would replace the `wait()`, `notify()` pairs with `new Thread()`, `join()` pairs. Each clock "tick" would correspond to a `new Thread()`, `join()` pair, as shown in Figure 6. This approach incurs the costly penalty of creating a new thread object at every clock tick, and on most Java runtime implementations, additional operating system overhead in starting new threads.

Hence, this approach is only practical for very large-grained concurrency, where concurrent threads have very few interdependencies. If finer grained concurrency is desired, where interactions between threads are rich are relatively common and opportunities for non-deterministic behavior are more frequent, the extra overhead incurred by using `join()` is unacceptable. This approach also is quite inelegant from a programming perspective, as events which are part of one logical thread are segregated into separate thread objects.

Java's synchronization primitives are inherently best at describing asynchronous communications between threads, where some amount of non-determinacy is expected and in fact desired. Examples of applications for which the primitives are well-suited include user interfaces, multimedia, and other interactive systems, where an occasional lost button click or dropped video frame is not detrimental to the system's overall correctness. However, to achieve the functionality required by synchronous* applications, where non-determinacy must be in general be avoided, abstractions designed specifically for synchronous applications are needed.

A RENDEZVOUS MECHANISM FOR JAVA

It is clear that a more efficient and easier mechanism is needed in Java to impose ordering on events in different threads. Without such a mechanism, only experts in Java and concurrent programming would be able to write non-deterministic, reliable multithreaded programs. To this end, a mechanism based on rendezvous [1] is written to help simplify multiprogramming tasks . The resulting implementation provides an abstraction that is powerful, easy to use, and efficient.

The rendezvous mechanism is implemented as a single class, and provides a set of functions useful for a wide range of applications. In short, it allows one to synchronize an arbitrary number of concurrent threads very easily. To use the rendezvous class, the methods used are `add()`, `remove()`, and `rendezvous()`. First, a new rendezvous object and the new thread objects to be synchronized are created. Then, the threads are placed in the rendezvous, using the `add()` method, usually before the new threads are started. Within the body of each thread, the `rendezvous()` method of the rendezvous object is called. This causes that thread to block

until all threads in that particular rendezvous have called `rendezvous()`. When all threads reach their rendezvous points, all the threads in the rendezvous resume execution. The effect of the rendezvous mechanism is shown in Figure 7.

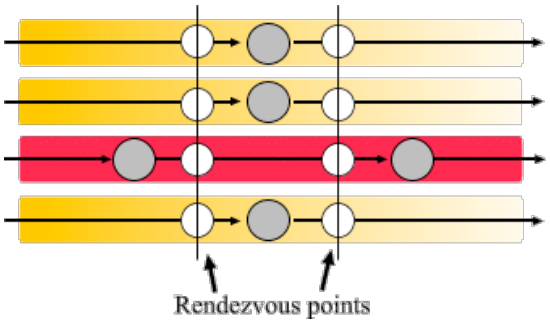

Figure 7. Effect of rendezvous mechanism

Additionally, at any point during its execution, a thread may elect to remove itself or another thread from the rendezvous. This feature supports the capability to implement conditional synchronization, which can be useful in many applications. Also, several rendezvous objects may exist within a program at once, and a single thread may participate in many rendezvous. This allows many different "clocks" to coexist within the program, and is also quite powerful in allowing a wide range of expression of behaviors. Taken together, the capabilities of the rendezvous class make it ideal for use in applications where inter-thread interaction is rich and complex, and where determinate behavior is desired.

The implementation of the rendezvous mechanism necessarily relies on the primitives provided by Java, but uses them very efficiently. The `notifyAll()` method is called only once per rendezvous "tick", and the `wait()` method is invoked once per tick per thread. As compared to the handshaking implementation, rendezvous does not make extraneous calls to `notify()` or `notifyAll()`. Compared to the `new Thread()`, `join()` approach, rendezvous does not incur the object creation and operating system overhead. It can also be argued that the programs using rendezvous are much cleaner and elegant than programs using the other two approaches.

EXPERIMENTAL RESULTS

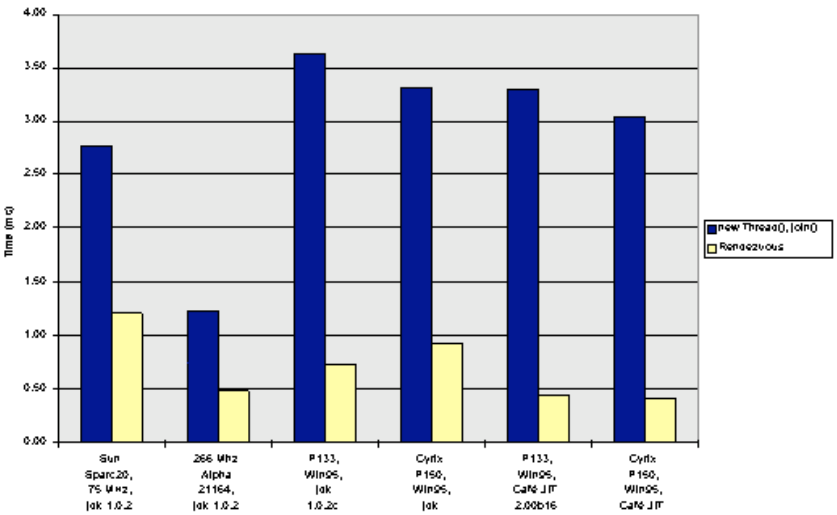

Figure 8. Time per new Thread(), join() pair vs. time per rendezvous "tick".

Performance measurements were taken comparing the `new Thread()`, `join()` approach with the rendezvous mechanism. Experiments were not performed comparing rendezvous to a handshaking protocol, as the latter is not easily scaleable to an arbitrary number of threads.

The platforms tested were a 75 MHz Sun Sparc20, running jdk

(Java Development Kit) 1.0.2; a 266 MHz DEC Alpha 21164 running jdk 1.0.2; a 133 MHz Pentium and 120 MHz Cyrix P150 running Win95, jdk 1.0.2, Symantec's jdk 1.0.2c and Café Just-In-Time compiler 2.00b16.

In almost all cases tested, the rendezvous implementation is faster than using `new Thread()`, `join()`, by a factor of from two to eight times faster. These results are graphed in Figure 8. The only exception was with Win95, running Sun's jdk 1.0.2, as shown in Figure 9. In these cases, the rendezvous implementation is much slower. Judging by the results obtained from the other configurations, this suggests that this is an anomaly resulting from a faulty implementation of `notifyAll()` in the x86 version of jdk 1.0.2.

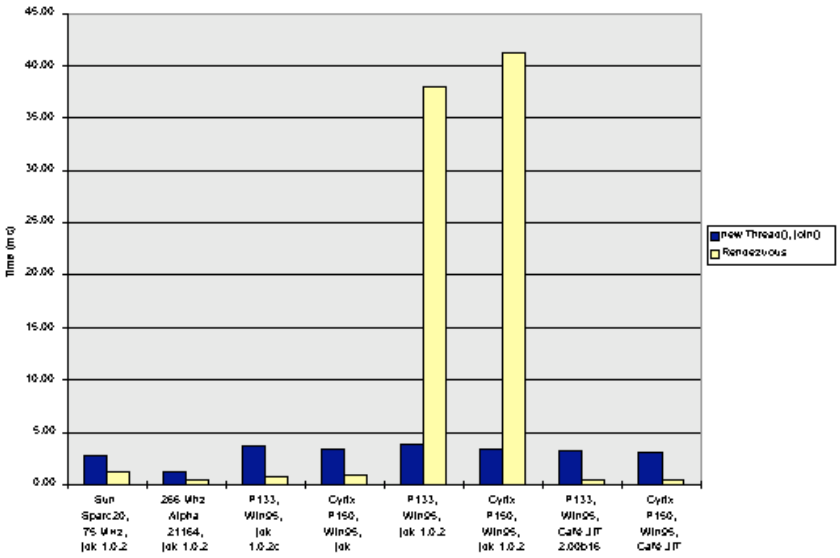

Figure 9. Inconsistent results obtained with jdk 1.0.2 for x86.

CONCLUSION

One of the most attractive features of Java, its built-in threads capabilities, is also one of its most dangerous. If not used intelligently, it is quite simple to create non-deterministic programs that are exceptionally difficult to debug. The source of non-determinism in a Java program lies in the arbitrary order of execution of events in different threads which may affect one another. To eliminate non-determinism, one must identify all events in a thread that may affect the behavior of events in other threads, and impose an ordering on these events.

It was shown that while Java provides primitives for synchronization, they are not appropriate for use as is to impose such orderings on events. A rendezvous mechanism was implemented for the Java language which is powerful, easy to use, and efficient. Experiments were run comparing the rendezvous to a comparable implementation using standard Java library calls, and it was found that rendezvous is superior in virtually all cases.

In summary, a rendezvous mechanism in Java was found to be quite valuable for use in a wide variety of applications, where it is necessary to synchronize events in multiple threads to ensure determinacy.

REFERENCES

[1] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, vol. 21, no. 8, pp. 666-677, 1978.

[2] G. Berry and S. Ramesh, "Communicating Reactive Processes," Proc. 20th ACM Conference on Programming Languages, pp. 85-98, 1993.

[3] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Reading, MA: Addison-Wesley, 1996.

Appendix A - Additional experimental results

Additional data was obtained from the experimental measurements, which are not directly relevant to the results obtained, but are useful for providing further background on the test environments, and the current state of Java technology. Figure 10 shows the approximate relative integer performance of all the configurations tested. The Just-In-Time compiler clearly boosts the performance significantly. Figure 11 shows just the performance of the interpreted Java engines.

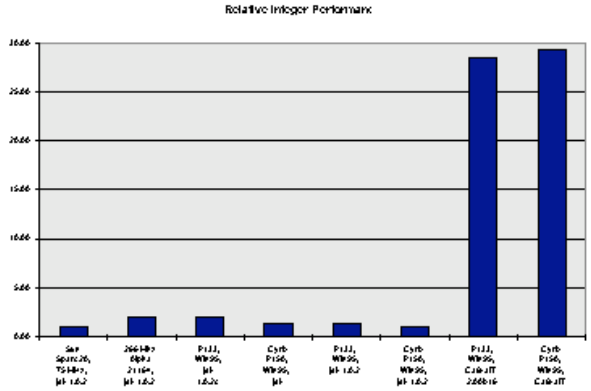

Figure 10. Relative integer performance of tested configurations.

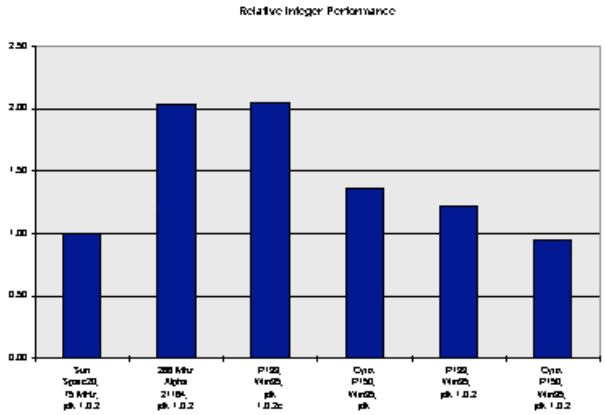

Figure 11. Relative integer performance, JIT's omitted.

APPENDIX B - SOURCE CODE FOR RENDEZVOUS CLASS

```
/*
 |
 | Rendezvous.java
 |
 | James Shin Young
 |
 | Copyright (c) 1996 by James Shin Young and the Regents
 | of the University of California.  All rights reserved.
 |
 | Permission to use, copy, modify, and distribute this software
 | and its documentation for NON-COMMERCIAL purposes and without
 | fee is hereby granted provided that this copyright notice
 | appears in all copies.
 |
 | Updated January 7, 1997
 |
 */

/**
 *        @author James Shin Young
 */

public class Rendezvous
```

```
{

////////////////////////////////////////////////////////////////////////////
//* Variables */

        /* Private variables */

        // Array to store all the threads involved in the rendezvous.
        private Thread[] target;
        // Keeps track of the number of threads in target
        private int targetIndex;
        // Array to set the state of all the threads.
        private boolean[] targState;

////////////////////////////////////////////////////////////////////////////
//* Constructors */

        public Rendezvous() {
                this(2);
        }

        public Rendezvous(int initSize) {
                target = new Thread[initSize];
                targState = new boolean[initSize];
                targetIndex = 0;
        }

        public Rendezvous(Thread[] initMembers) {
                target = initMembers;
                targState = new boolean[target.length];
                for (int i = 0; i
```
