Communicating Process Architectures 2000
P. H. Welch and A. W. P. Bakkers (Eds.)
IOS Press, 2000

Native JCSP – the *CSP for Java* Library with a Low-Overhead CSP Kernel

James MOORES (jm40@ukc.ac.uk)
Computing Laboratory, University of Kent at Canterbury, CT2 7NF

Abstract. The JCSP library provides a superior framework for building concurrent Java applications. Currently, JCSP is a collection of classes that uses the standard Java Threads mechanism to provide low-level facilities such a process scheduling and synchronization. The overheads of using Java Threads can be quite large though, especially for synchronization and context switching.

This paper begins by describing various options for increasing performance, and then how the standard Java Threads work. The integration of the low-overhead CCSP run-time system into a Linux-based Sun JDK 1.2.1 Java Virtual Machine is then described. This integration provides the low-level support required to dramatically increase the performance of the JCSP library's model of concurrency. The paper then looks at the problem of maintaining backward compatibility by preserving the functionality of the existing threads mechanism on which much legacy code depends.

The paper finishes by looking at the performance displayed by the current prototype JVM and contrasting it with the performance of both Green (co-operatively scheduled) and Native (operating-system scheduled) Java Threads.

1 Introduction

The Java programming language has been widely praised for more tightly integrating support for concurrency directly into the core of the language. Concurrency-related problems remain though: race hazards, deadlock, and starvation are all still major causes of bugs in concurrent systems written in Java. These problems are aggravated by the lack of transparency in the combinational semantics of Java Threads. Large projects, such as the *Swing* library, have simply abandoned threading wherever possible and actively discourage the use of threads by developers.

The JCSP library [1] brings the improved security and scalability of CSP-based languages like occam to concurrent programming in Java. The current JCSP implementation is as a set of classes built on top of the standard Java Threads model, and carries the overheads associated with Java Threads. It allows the development of more scalable, more highly concurrent Java applications and components.

Although Java provides two different underlying threads mechanisms, both are still between one and two orders of magnitude slower than CSP-based platforms like occam and CCSP. The high overheads mean that Java developers are discouraged from using too many threads and from synchronizing those threads too often. This in turn leads to a reluctance to use concurrency as a structural tool to express the natural parallelism in a problem. Instead, threading is often used just as a tool for enhancing system response time and in utilizing multiprocessors.

Native JCSP solves these problems by bypassing the layers of software that JCSP must go through. Under Native JCSP, instead of using Java Threads, processes run directly on a modified version of the CCSP system that is integrated into the JDK1.2.1 JVM.

The CCSP system is a low-overhead run-time kernel originally designed to support occam and C. The resulting increase in performance removes the need for developers to worry about the costs associated with a greater use of concurrency.

2 Possible approaches to implementation

The concept of integrating radically different support for concurrency into the Java platform could be implemented in several different ways. Looking at how other languages support CSP constructs should provide some direction.

occam is the original CSP-based language. It was designed to run on the Inmos Transputer architecture. The Transputer provided all the necessary concurrency services as a part of the instruction set of the machine itself. Microcode was used to perform complex primitives such as operations to start and stop processes, perform inter-process communication, and so on. No other architecture has since provided such complex concurrency facilities at such a low level.

The KRoC project [2] achieved the same result on more conventional and more modern architectures. By moving the most complex functionality into a software kernel (or 'run-time system'), and in-lining more simple operations it was able to replace the microcode functionality of the transputer with a thin layer of software. The SPOC [3] project was similar, but generated C code that included its own scheduler (basically a large `switch()` statement) [4].

CCSP [5] brought a KRoC-style kernel to the C language (and, incidentally, it also supports KRoC occam). It interfaces the language with the kernel using a function library and macros to provide an API for process management and IPC very similar to that of the Inmos C compiler for the Transputer (which used the Transputer's microcoded instructions much like occam). The kernel was based on a SPARC assembler KRoC kernel, but was rewritten in a mixture of C and embedded assembler. Since then the CCSP kernel has evolved to include numerous extra facilities to the C programmer. These include priorities, an external communications interface, 'native' timer support, and a flexible API.

3 A CSP runtime system under Java

There are several possible approaches to making the use of CSP under Java more efficient. The existing Java Threads[1] implementation could be rewritten. This would bring the added advantage of speeding up all existing Java applications too. A disadvantage of this approach is that CSP operations may needlessly be translated to multiple Java Threads operations.

The Java Threads model is full of subtleties, and some operations, such as changing another Thread's priority and dequeuing multiple threads from a monitor, can be quite inefficient to implement. Beating the current implementations to the same degree as CCSP may therefore not be possible.

As an aside, it is also worth noting that although Java Threads, or more accurately Hoare Monitors, are generally seen as a lower-level scheme than CSP, the opposite could also be argued. Welch and Martin [6] have shown that Java Threads can be expressed in terms of CSP, so it may even be that at some time in the future the existing Java Threads mechanism could be built on top of a CSP-based kernel.

Having decided to implement a separate CSP-based run-time system rather than rework the existing one, there are several different levels at which the kernel might be implemented.

[1] 'Threads' is capitalized here because it refers specifically to the implementation of the class `java.lang.Thread`

Could a kernel be hand-written at the byte-code level? The main advantage of this approach is that it should work on all existing JVM implementations so porting to different architectures would not be necessary. One disadvantage would be that it would only execute at a speed of the JVM it was running on. This would be less of a problem with modern JITs and dynamic compilers like Sun's *Hotspot*. However, the Java language does not support the necessary low-level operations necessary to manage multiple stacks.

So unfortunately, while an attractive option, this does not appear to be possible. Calling the JVM [7] a 'virtual machine' is somewhat misleading. The instruction set of the JVM is quite specific to Java's object model. It is not possible, for example, compile an arbitrary C program into byte-code. There are several reasons for this. Byte-code is heavily type-checked for security reasons to ensure that, for example, only the correct type of pointers are dereferenced. Byte-code can also not perform arbitrary absolute jumps. It has only method calls and relative jump instructions for flow control, and the relative jumps are restricted to 16-bit offsets in most JVM's (32-bit offsets are supported as an optional extension in the specification). Thirdly it may be impossible to manipulate things such as the frame and stack pointers as their use is usually implicit. Tampering with these in strange ways is also likely to trigger the security manager. This last point really rules out a byte-code-based kernel.

As a different starting point, the implementation of the existing Thread support framework was examined. The platform used for research and eventual development of the Native JCSP prototype was Linux 2.0/2.2/2.3 running on x86 processors. The source code was a Sun Microsystems Solaris source release (1.2.1) patched with the Linux 'Blackdown' source patches and built with EGCS-1.1, a custom version of glibc 2.1.1, and Lesstif.

4 So how does the existing Thread support work?

Information on how Java actually implements threads appears to be very thin on the ground. One approach that might be expected is for support to be provided in the byte-code instruction set. In fact there are only a couple of instructions involved in concurrency, and they simply relate to entering and leaving a synchronized method, so they are really just locking operations.

From examination of the JDK source code it was established that the bulk of Java's multithreading functionality is provided by native method calls from `java.lang.Thread`. These operate through the Java Native Interface (JNI) mechanism supported from Java version 1.1. Under Java 1.0 and Microsoft's JVM, native calls work quite differently.

These native methods calls invoke C functions, and transform parameters into the equivalent C types where possible. The JNI provides facilities to access any Java-specific data structures such as data members, or even invoke methods.

Usually, an application developer would generate a separate library in the native format that could be loaded as required by the JVM. The major libraries (including the Threads package) included in the JDK all have their native parts implemented inside the JVM and so do not need to be demand-loaded.

Within the JVM there is an extensive layering of interfaces that hide thread functionality. At the top layer there are the calls that correspond directly to the native methods. These then call a seemly redundant layer, which then calls the Hardware Portability Interface (HPI). The HPI layer then either is implemented as *Green* threads or drops down into the native thread library. The Green thread support is the threading implementation that was originally included with Java 1.0. Green threads are user-level threads that provide non-blocking I/O and are co-operatively scheduled. They were largely side-lined in later versions of the JVM because they do not support SMP multiprocessors and have sometimes proved unresponsive to external asynchronous events.

Green threads are more like the JCSP processes that we are aiming for, in that they are more lightweight and are co-operatively scheduled. They are also relatively portable across multiple operating systems because most of the functionality is not greatly dependent on low-level operating system calls. When Green threads do need to access operating system functions, they use standard POSIX calls wherever possible. This has meant that when Java is being ported to a different platform, the Green threads support is usually available before the full native threads.

'Native threads' refer to threads supported directly by the operating system kernel. The current implementation of Native JCSP runs on the Linux operating system. The Linux kernel does not support threads directly, but provides the `clone()` system call. `clone()` is like the standard `fork()` call, but it allows the spawning of kernel processes that share the same address space. Additionally it has flags to tell the kernel to allocate stack space automatically.

The benefits of lightweight threading have become more apparent as Java has matured, and there has been a demand for a reduction in the overheads. The latest versions of the Hotspot dynamic compiler now uses Green threads in combination with native threads to give the best of both worlds and do represent a significant improvement over Java 1.1.

As Java Threads aim to hide some of the less elegant details of previous threading libraries, they do not allow the developer to specify anything about the stack size of each thread. So how does the system allocate stack space?

4.1 Dynamic stacks

Dynamically expanding stacks are implemented by reserving an area of virtual memory of the maximum stack size. Only the very top page of this area is actually allocated physical memory. As the stack is used during program execution it will probably require more than just the minimum stack size (a single page). If the thread accesses the area below the initially allocated page a page fault exception occurs. Normally a page fault would cause the process to terminate if the page had not been allocated, or it might cause the operating system to read in a page from a swap file on disk. In this case, however, it is used to allocate a new page of physical memory. The page below the new page is then set up to trigger the same action and processing is then resumed. See Figure 1

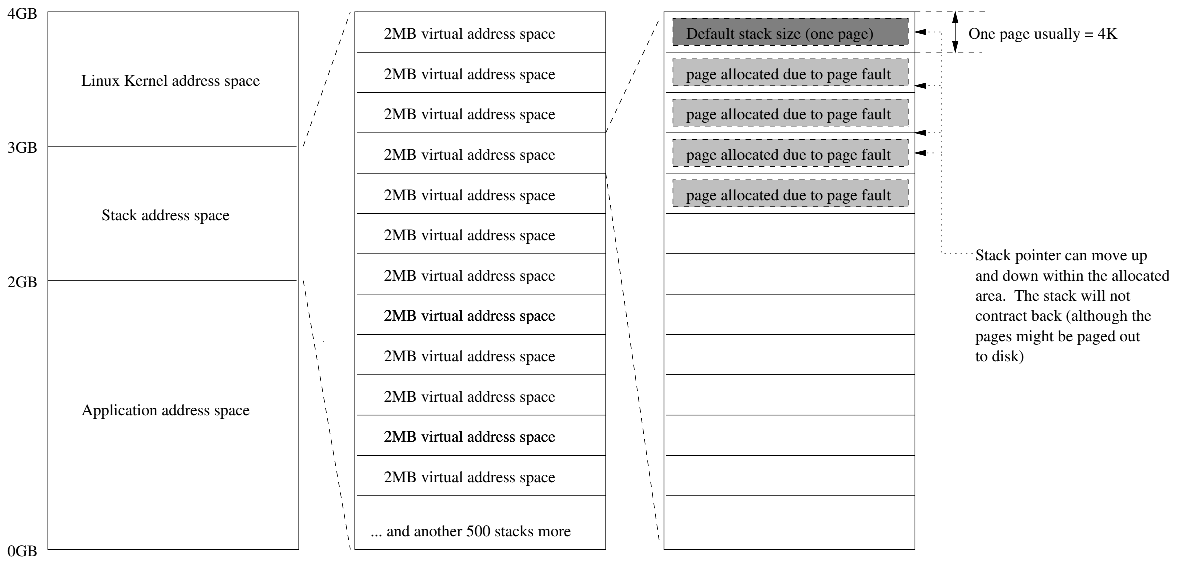
4GB
3GB
2GB
0GB
Linux Kernel address space
Stack address space
Application address space
2MB virtual address space
... and another 500 stacks more
Default stack size (one page)
page allocated due to page fault
One page usually = 4K
Stack pointer can move up and down within the allocated area. The stack will not contract back (although the pages might be paged out to disk)

Figure 1: How dynamically expanding stacks are mapped into Linux's virtual address space

The number of these stacks is limited by the virtual address space of the host architecture. Usually this is 32 bit, and under Linux about a quarter of the address space is given over to stack allocation. This means that all stacks must fit within about one gigabyte of virtual address space. Given that each stack typically has a maximum size of around two megabytes, there is an implicit limit of 512 stacks.

For the current generation of multi-threaded applications this would usually be sufficient, but there are examples (web servers, applications servers and so on) where this limit is already being reached. An additional aspect of JCSP is the promotion of the extensive use of concurrency as a natural part in the expression of algorithms and real world objects. For the moment, the only way to circumvent this limit is to reduce the maximum stack size. This can only work if *all* of the threads can live within this reduced stack size, so it can only be used to solve certain problems.

New architectures, like the Intel Itanium (IA-64), and even some existing architectures, such as the API (previously DEC) Alpha, will largely cure the problem. The 64-bit virtual address spaces of these architectures should provide ample space for stacks, and allow the increase of maximum stack sizes. Alternatively, Dijkstra's method of allocating activation records from a stack could be abandoned in favor of something more tolerant of concurrency, such as the method Brinch Hansen used for SuperPascal [8]. The use of this approach with occam has also been explored by Wood [9].

4.2 LinuxThreads

Because the `clone()` call only provides the basis for a native threads library, there are numerous libraries available for Linux: Bare-Bones Threads, DCEThreads, FSU Pthreads, JK Threads and Radke Threads. Perhaps the best known is LinuxThreads which provides a POSIX 1003.1c threads-compatible implementation on top of `clone()`. LinuxThreads was originally only available as a separate package that was not included with most Linux distributions. It has now become integrated into the GNU C Library (glibc), so it is effectively the standard implementation.

The JVM used for the development of Native JCSP is based on the Linux version of the JDK1.2.1 ported to Linux by the 'Blackdown' group. The Blackdown group chose to use LinuxThreads as the basis for their Native Threads support.

5 Chosen implementation method

LinuxThreads, like CCSP, is a multi-threading C library and API (although in CSP terms we would prefer to call them processes rather than threads, they are essentially the same thing). It should therefore be possible to implement native JCSP by interfacing the JVM to CCSP in a similar fashion to that in which the JVM is interfaced to LinuxThreads.

5.1 Context switching

The most complex problem in implementing multi-threading is actually achieving the switching from one process to another. Once this has been achieved, communication and synchronization functions can fall into place relatively easily. The reason for this is that context switching is an operation that requires the greatest knowledge of the programming environment. A context switch requires all of the state of the currently running thread/process to be saved and so intimate knowledge of the system's per-thread structures is required. Contrast this with most other operations which are really just operating on structures internal to the kernel (and therefore far more familiar).

Because of this the primary focus of the development of Native JCSP was to be able to create JCSP processes and switch between them using the CCSP kernel's scheduler.

The first step to understanding what was required was to trace the operation of the existing threading mechanism.

One caveat concerning native JCSP is that it must still allow the continued operation of the existing Java threads mechanism. This is necessary to enable JCSP to utilize the huge library of APIs that make Java a useful platform. Most if not all of them rely on standard Java Threads behaviour in some form or other, so it important for that behaviour to stay the same.

5.2 Startup

Thread or process initialization and startup are often split into two separate stages using different API calls. But in Java, threads are started immediately as they are created (through the `Thread()` constructor, either directly or via a derived class). Java is able to do this because the user does not need either to set up parameters to be passed to the thread, or to specify the amount of stack space to allocate. The parameters either can be passed in via the constructor, or may not be required at all because the thread is implemented as a class and the required data may be available as class data members (properties). The stack size is not needed because of the dynamic allocation scheme described above.

When a thread object is created, its constructor performs some housekeeping of other objects (primarily to maintain the system of thread groups). As the constructor is a normal Java method, this housekeeping code is written in normal Java that is executed by the JVM. It then calls a special native method called `start()` that directly invokes a C function inside the JVM using the JNI interface.

5.3 The Environment

Within the JVM, every Java Thread has its own structure, called its *environment*, that holds all data specific to that particular thread. This includes pointers to its execution stack, its thread descriptor, and so on. Whenever a native method is called, a pointer to this structure is passed. In the case of a static native method, a pointer to the Java class in which the native method is declared is also passed. If the native method is not static, then a pointer to the object of which it is a method is passed instead. These two pointers allow the native method to invoke other Java methods from C. This ability is the key to enabling context switching.

When creating a new context in CCSP or LinuxThreads, a new stack (dynamic or otherwise) needs to be allocated and the starting function must be set, along with its parameters, on the stack or in registers (depending on the architecture). Similarly, a Java Thread needs its environment structure set up, its dynamic stack initialized, and various other bits of housekeeping. The JVM then spawns a new thread by calling the relevant function in the LinuxThreads library. This *new* environment structure, and either a class pointer or object pointer to the class or object whose `run()` function we are going to invoke, are then passed to the initial C function that is executed when the thread starts up. This function simply acts as if it were a native method that has just been called from Java. The newly started function has a valid environment and a class/object pointer, so it can, using the JNI, invoke the Java `run()` method of the class or object that represents the new thread. The new thread is now started. It would now seem to be a simple case of performing the same sequence of operations to start a CCSP process rather than a LinuxThreads thread.

6 The co-existence of Threads and Processes

If JCSP processes really *replaced* Java Threads, then the user of the library would be unable to execute any code inside a JCSP process that relied upon the standard Java Threads. For example, the execution of a `synchronized` method or a `wait()` or `notify()` would cause the JVM to translate the operation into a call to the underlying threads library. To do this it would find the thread descriptor pointer field in the Environment structure and try to pass that to LinuxThreads to perform the operation. This would fail because the thread descriptor pointer field would contain a pointer to the CCSP process descriptor, rather than a LinuxThreads thread descriptor.

Limiting JCSP to not allowing the use of the standard thread facilities would make Native JCSP so limiting as to be pointless. Even the execution of `System.out.println()` requires the acquisition of a lock through a `synchronized` method call (to prevent interleaving on the output). A method whereby a piece of code could be running in both a Java Thread and a JCSP Process would be more desirable.

The simple case is that all JCSP processes operate within a single Java Thread. There should not be a problem in sharing a single Thread descriptor over multiple JCSP processes as long as no process holds a lock used by another JCSP process over a context switch. As context switches can only happen during calls into the kernel, it should be relatively easy to avoid this.

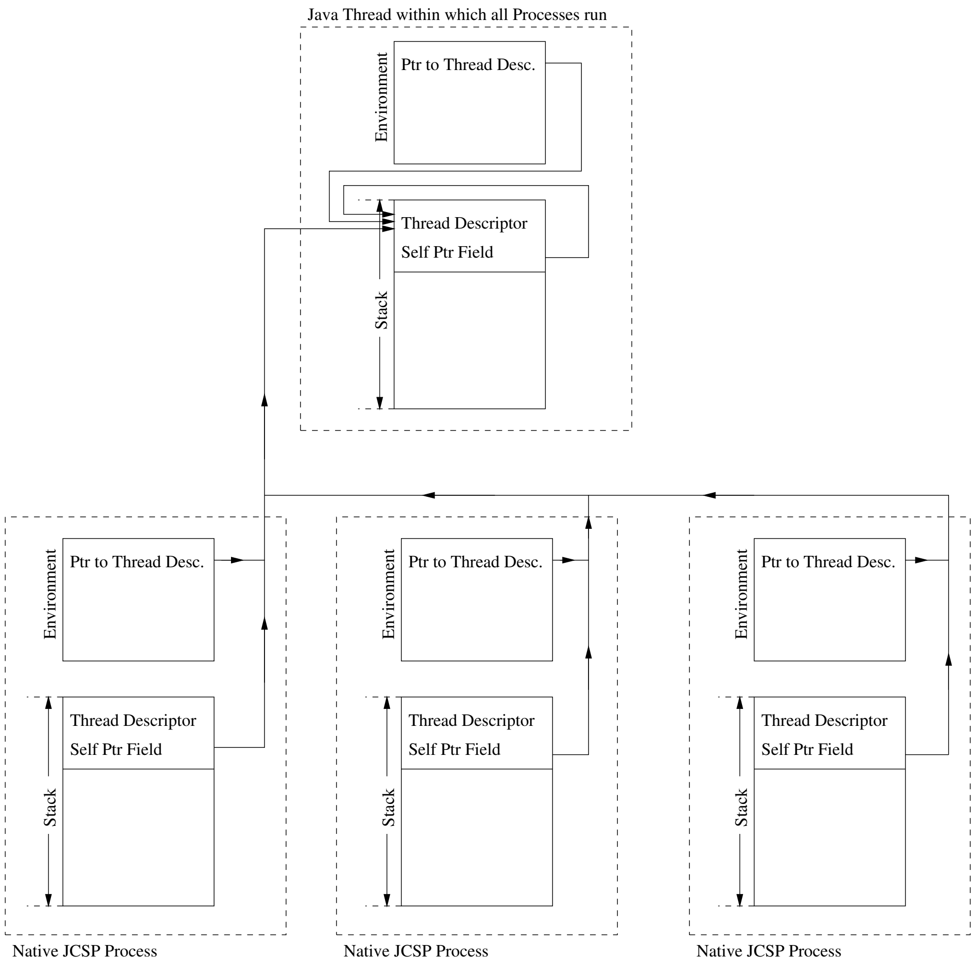

Figure 2: How Native JCSP makes all Processes appear as one unified thread

A more complex case is spreading the execution of JCSP processes over multiple Java Threads. This would enable JCSP Processes to execute blocking operations without suspending execution of all other JCSP processes. It also enables the use of SMPs, and this is discussed further in Section 8.

The primary reason for initially implementing this simple case is that the underlying CCSP kernel is not thread-safe by default. If interrupted it may leave data structures in an intermediate state.

However, this sharing of the thread descriptor turned out to be quite difficult to implement. The main problem is that whenever a Java Thread operation is performed it translates down to a call to LinuxThreads. Internally, most LinuxThreads operations require the current thread descriptor. The desired effect is for LinuxThreads to see the same thread descriptor for JCSP Processes. Unfortunately the mechanism for identifying the current thread descriptor is interfered with by JCSP processes all having separate stack spaces.

6.1 LinuxThreads' incompatibility with CCSP

On examination of the LinuxThreads source code (which is available under the LGPL open source licence) it became clear that two different mechanisms have been used to identify the thread descriptor of any arbitrary running thread.

One method, that appears to have now been abandoned, is to store the thread descriptor in the code segment register of the x86[10]. Because Linux runs a flat memory model, rather than the segmented memory model used by the 16-bit x86 architecture, applications never need to use the segment registers. The GCC compiler never normally generates code that uses them, so as long as the application developer does not write any specific code to use them, things will work correctly. Although this method has been abandoned, ironically, it would probably have prevented the incompatibility problem.

The second method, which is currently in use, is to store the thread descriptor at the very top of the dynamically allocated stack and start the stack pointer a little further down the initial page. The top of each stack is aligned to an address of its maximum size. For example, a two megabyte stack would be allocated to start from a two megabyte boundary. This means that the thread descriptor can be found simply by masking off the bottom bits from the stack pointer (which will put us at the very bottom limit of the stack), then adding two megabytes to point at the top. Unfortunately this means that when LinuxThreads attempts to find out the current thread within a JCSP Process, it goes to the top of a stack managed by CCSP, which doesn't have the current LinuxThreads thread descriptor, and so fails to work.

To combat this problem, the thread descriptor of the initializing Java Thread was copied onto the top of each JCSP Processes stack. This worked well for operations that did not modify the thread descriptor, but those that did caused all of the copies to fall out of synchronization. Fixing this required producing a special version of LinuxThreads. Within the LinuxThreads thread descriptor there is a field that just points back to its own structure; the function call to get the current thread descriptor was modified to take advantage of this. Rather than directly reference the top of that structure, it would, instead, insert another level of indirection and access the field that pointed to itself. This meant that the modified library worked in exactly the same way for all existing applications. However, for Native JCSP, the copy of the thread descriptor at the top of the stack was modified (in fact, we need only copy this one field, but it must be at the same offset), so that the field that is supposed to point back to its parent structure actually points back to the original thread descriptor. This means that all thread operations now occurred on the same thread descriptor, so all JCSP Processes effectively appeared as the same thread.

7 The current prototype

The current prototype was developed with the modified LinuxThreads library and the same principles as the Native Thread implementation. It can successfully launch JCSP processes that run under a single Java Thread and allow the use of standard Thread operations. The JCSP processes are context-switched correctly by the underlying CCSP kernel and preliminary performance results are available. The source code to the performance test follows:

```
import jcsp.lang.Process;
import java.io.*;

class P1 extends Process {
  public P1() {
    this.start();
  }
  public void run() {
    System.out.println("Time read in P1 is :"+
                       System.currentTimeMillis());
    long t1 = System.currentTimeMillis();
    for (int i=0; i<1000000; i++) {
      this.reschedule();
    }
    long t2 = System.currentTimeMillis();
    System.out.println("Time read in P1 (after loop) is :"+
                       System.currentTimeMillis());
    System.out.println("difference is:"+(t2-t1));
  }
}

class P2 extends Process {
  public P2() {
    this.init();
    this.start();
  }
  public void run() {
    for (int i=0; i<1000000; i++) {
      this.reschedule();
    }
  }
}

class tst {
  public static void main(String[] args) {
    Process np2 = new P2();
    Process np1 = new P1();
    np1.stop(); // stop() is static so it means
                // the current thread should enter
                // the kernel to allow out two Processes
                // to start running
  }
}
```

7.1 Performance

The prototype currently only supports the interpreted JVM although it is thought that it should be relatively straightforward to refine the implementation to support both the JIT and Hotspot.

Threading System	*Context Switch Time*
Java Native Threads (JDK1.2.1)	$20\mu s$
Java Green Threads (JDK1.2.1)	$11\mu s$
Native JCSP (JDK1.2.1)	$0.8\mu s$
CCSP (EGCS1.1)	$0.35\mu s$

Figure 3: Performance on a 300MHz Intel Celeron

Figure 3 gives some performance figures. These should not change substantially when all the JCSP features are implemented.

Although a context switch may appear a rather crude measure of performance, most other kernel functions should take approximately the same time as nearly all operations are $O(1)$. Additionally, the most expensive part of a kernel function is usually the context switch. Some functions will even be faster if they do not require a switch.

There is some slack (function calls going down through the layers of APIs) that could be removed to further speed the Native JCSP version.

It should also be noted that although Native JCSP is around 15–25 times faster than the standard threads in JDK1.2.x, it is well over 100 times faster than the JDK1.1 Threads implementation.

8 Future developments

The basic framework is complete. The prototype's structuring of its `Process` class does not actually fit very well with the JCSP class model. The class has since been remodelled to work correctly in JCSP's class hierarchy. This primarily involved deciding where to locate the native methods involved with launching processes. In fact the best place to put them is in the JCSP `Parallel` class directly rather than providing a lower-level `Process` class as is done in the prototype. CCSP also needed to be extended to add new `List` functions that take arrays of `Processes`, `Channels` and so on, to ease the implementation all the other primitives via the JNI.

The primary aim is to implement the full JCSP API. Once this is done, there are several other interesting possibilities.

CCSP does have some support for running in multiple system threads. This is implemented as a global spin lock that all entries to the kernel must aquire. Because kernel calls are mostly very short, there should be little lock contention. This is the same method of avoiding race conditions as is used to enable Linux 2.0 to be used with SMPs. To allow greater scalability, Vella's lock-free algorithms [11] for a KRoC CSP kernel could be implemented. This would allow excellent scalability as the only thing used is atomic swaps, so contention is never for more than one instruction.

The global spin-lock option was designed to spread computation over Shared Memory Multiprocessors, but could equally be used to support multiple Java Threads. It should be possible to use this to transparently allow JCSP to continue processing other processes if one blocks due to a blocking I/O operation. It should also be relatively easy to allow Native JCSP to take advantage of SMPs by running separate copies of the kernel in different Java Threads. The operating system should then load-balance them over multiple CPUs (in the absence of processor affinity facilities).

9 Conclusion

The current implementation successfully demonstrates the viability of a native version of JCSP. The performance benefits alone should allow much more concurrency to be used within Java applications. It is hoped that soon the full API will be supported and that the system will work under just-in-time compilers and the Hotspot dynamic compiler.

The prototype has demonstrated excellent performance and has shown the levels of performance that are possible.

The future is then focussed on ultra-efficient SMP implementations and enabling the use of blocking method calls more easily.

References

[1] Peter H. Welch. *JCSP Home Page* http://www.cs.ukc.ac.uk/projects/ofa/jcsp/

[2] D. C. Wood & P. H. Welch. *The Kent Retargetable* occam *Compiler.* Proceedings of WoTUG-19: Parallel Processing Developments, edited by B. C. O'Neill. IOS press, 1996. ISBN 90–5199–261–0.

[3] M. Debbage, M. Hill, S. Wykes and D. Nicole. *Southampton's Portable Occam Compiler (SPOC)* Proceedings of WoTUG-17: Progress in Transputer and occam Research, edited by R. Miles and A. Chalmers. IOS press, 1994. ISBN 90–5199–163–0.

[4] B. M. Cook. *A Fast C Kernel for Portable* occam *Compilers* Proceedings of WoTUG-18: Transputer and occam Developments, edited by P. Nixon. IOS press, 1995. ISBN 90–5199–222–X.

[5] J. Moores. *CCSP – A portable CSP-based run-time system supporting C and* occam Proceedings of WoTUG-22: Architectures, Languages and Techniques, edited by B. M. Cooke. IOS press, 1999. ISBN 90–5199–480–X.

[6] Peter H. Welch and Jeremy M. R. Martin. *Formal Analysis of Concurrent Java Systems* Proceedings of Communicating Process Architectures – 2000, edited by P. H. Welch and A. W. P. Bakkers. IOS press, 2000.

[7] Bill Venners. *Inside the Java virtual machine* New York, London, McGraw-Hill, 1998. ISBN 0–07–913248–0

[8] Per Brinch Hansen. *Efficient Parallel Recursion* SIGPLAN Notices 30(12): 9-16 (1995).

[9] David C. Wood. *An Experiment with Recursion in* occam. Proceedings of Communicating Process Architectures – 2000, edited by P. H. Welch and A. W. P. Bakkers, pp. 193-204. IOS press, 2000.

[10] *Sandpile.org – The world's leading source for pure technical 80x86 processor information.* http://www.sandpile.org/

[11] K. Vella and P. H. Welch. *CSP/*occam *on shared-memory multiprocessor workstations.* Proceedings of WoTUG-22: Architectures, Languages and Techniques, edited by B. M. Cooke. IOS press, 1999. ISBN 90–5199–480–X.