Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

Towards Millions of Processes on the JVM

Jan Bækgaard PEDERSEN [1], and Andreas STEFIK
University of Nevada, Las Vegas, USA

Abstract. In this paper we show how two previously published rewriting techniques for enabling process mobility in the JVM can be combined with a simple non-preemptive scheduler to allow for millions of processes to be executed within a single Java Virtual Machine (JVM) without using the built-in threading mechanism. The approach is tailored toward efficient execution of a large number of (CSP style) processes in Java bytecode running on the Java Virtual Machine. This may also prove useful for languages that, like ProcessJ, target the JVM as an execution platform and which need a much finer level of threading granularity than the one provided by the Java programming language system's threading mechanism.

Keywords. process oriented programming, non-preemptive scheduling, user-level scheduling, Java Virtual Machine, Java, ProcessJ

Introduction

As the Java platform has matured over the last two decades, its runtime system, the Java Virtual Machine (JVM) has become a popular target for many languages. Examples of languages that target the JVM include Clojure [1], Groovy [2], Scala [3], JRuby [4] and Jython [5]. In this paper we consider the ProcessJ [6] language, which is under development at the University of Nevada Las Vegas. ProcessJ has several back-end targets like C (using the CCSP runtime [7]) as well as the JVM. ProcessJ is a process oriented language similar to occam-π [8,9,10,11], but with Java-like syntax. Since ProcessJ is process oriented, processes play a crucial role in the language. Such processes, when executing, need to run independently (i.e., control its own data and event responses like channel communication and barrier synchronization), and therefore must map processes to the underlying runtime (e.g. JVM).

There are two different approaches to targeting the JVM as an execution architecture. Firstly, a compiler can produce Java source code and then use the Java compiler to produce Java class files (containing the bytecode along with other information like field definitions, exception tables etc.), which subsequently can be executed on the JVM. The other approach is to directly generate Java bytecode through a tool like ASM [12], which will produce the appropriate class files. Let us consider these two options in turn.

If a compiler produces Java source code, then this source code must make use of communication libraries to realize processes, channel, barriers etc. A possible approach could be to use the JCSP library [13,14,15]. JCSP provides the necessary entities needed to execute a process oriented program, but it has a one-to-one mapping between JCSP processes and threads, therefore has similar process memory and runtime overheads.

If producing Java bytecode via Java source is not attractive or possible, byte code generation libraries like ASM can be used. ASM is a Java bytecode manipulation tool which can be used to manipulate or create Java class files. Naturally, such generated class files must

[1]Corresponding Author: *Jan Bækgaard Pedersen, Department of Computer Science, University of Nevada, Las Vegas, 89054, NV, USA*. Tel.: +1 702 895 2557; Fax: +1 702 895 2639; E-mail: `matt.pedersen@unlv.edu`.

rely on a library of classes that support process oriented programming. Generating files using JCSP with ASM would not fix the primary concerns, as the reliance on a one-to-one mapping of processes to Java threads remains.

Given that the processes of JCSP map directly to the JVM threading mechanism, running more than a few thousand threads in a single JVM may render it so slow that it becomes useless for all practical purposes. In correspondence with the JCSP developers, they list this limit to be around 7,000 JCSP processes. A Java/JVM thread is generally not conducive to running hundreds of thousands or millions of processes—which previous work has shown does occur in some process oriented systems [16]. For example, the occam runtime system CCSP supports executing millions of processes in one execution. For such scales, mapping of processes to threads may not be a viable solution for process-oriented languages targeting the Java Virtual Machine.

A lightweight threading model that uses the pre-emptive scheduling mechanism of the JVM is not an option; at least not without rewriting parts of the implementation of the JVM, but that renders the idea of portability of the class files moot. Something else must be done. One solution might be to develop a threading model (which we will call the process model from now on) that can be executed by a user-level scheduler. A simple, naïve user-level scheduler is not hard to write but for one caveat: to the best of our knowledge, there is no known technique for pre-emptive scheduling at the user-level. For this approach to work, the scheduler must be non-preemptive and the process model must support (and correctly implement) explicit yielding to ensure execution progress. This kind of scheduling is also referred to as cooperative scheduling. Luckily, this will work well with a process oriented language that implements synchronous, non-blocking, and non-buffered (channel) communication (as well as other blocking primitives like barrier synchronization), as we will see later. At this point, it should be mentioned that implementing an efficient user-level scheduler is not a simple task; indeed, it is a research area all by itself. An example of such work applied in the occam π scheduler can be found in [17].

In this paper, we present a method for generating Java classes that represent ProcessJ processes, which can be executed by a custom non-preemptive user-level scheduler. The rest of the paper is organized as follows: in Section 1 we present a simple non-preemptive scheduler to form the basis for our processes model. In Section 2 we present the background material that forms the basis for the work presented in this paper. Section 3 outlines the code generation that a compiler targeting the system must follow, as well as some simple bytecode rewriting needed to support yielding (called suspension for mobile processes) and re-animation of processes as they are descheduled and rescheduled again. In Section 4 we present test results, and Sections 5 and 6 contain the conclusion and future work, respectively.

1. A Simple User-Level Scheduler in Java for Non-preemptive Processes

Writing a user-level preemptive scheduler, as far as we know, is not possible, whereas a naïve non-preemptive one is fairly straight forward to implement. An (oversimplified) piece of pseudo-code for such a scheduler can be seen in Figure 1: for as long as there are processes in the process queue, take one out, run it (if it is ready to run) until it either terminates or yields; when it yields (or if it was not ready to run), put it back in the queue, and repeat until the queue is empty. In Appendix A Java code outlining our prototype implementation is listed.

For such a scheduler to be usable, the implementation of the processes (and entities like par blocks and read/write calls) must be implemented in such a away that they cooperate in the scheduling; that is, processes must voluntarily give up the CPU and let the scheduler run a different process (i.e., processes must explicitly *yield* and give control back to the scheduler). This is of course necessary as the scheduler itself cannot force a process to yield.

```
Queue<Process> processQueue;
...
// enqueue one or more processes to run
...
while (!processQueue.isEmpty()) {
    Proecss p = processQueue.dequeue();
    if (it p.ready())
        p.run();
    if (!p.terminated())
        processQueue.enqueue(p);
}
```

Figure 1. Pseudo-code for a simple non-pre-emptive scheduler.

A number of issues must be resolved before such a scheduler is of any use, and they include (short answers are given as well, but more in-depth explanations will follow in subsequent sections):

- *How does a procedure yield?* By returning control to the scheduler through a **return** statement.
- *When does a procedure yield and who decides that it does?* Every time a synchronization primitive is called a procedure yields voluntarily.
- *How is a procedure restarted after having yielded?* It gets called and is itself in charge of 'jumping' to the right code location.
- *How is local state (parameters and local variables) maintained?* They are stored in an $Object$ array before yielding and restored upon resumption.
- *How are nested procedure calls handled when the innermost procedure yields?* The caller makes the activation record with the parameters before the call, and yields after, if the procedure itself yielded.

Before tackling these issues, let us first consider the approach taken in [18] to implement mobile ProcessJ processes in the JVM. This technique will serve as the basis for the generation of code for the non-preemptively schedulable processes which are the focus of the paper.

2. Background

2.1. Mobile Processes

In occam-π, a mobile process is a process that can be suspended explicitly using the `suspend` statement. Similarly, a suspend statement is used in ProcessJ to achieve suspension of a mobile process. In both languages, when a process suspends, it stops running and becomes a passive piece of data, which can be communicated on a channel to another process, connected to compatible elements of the environment of that process and resumed simply by calling/invoking it and passing it the appropriate actual parameters. Once a mobile process is restarted, execution continues with the statement immediately following the statement that suspended the process.

In [18] we outlined an approach to implementing such processes on the JVM; the steps taken (which we will elaborate on in Section 3 when we show how to use the technique to implement regular ProcessJ procedures that engage in synchronization such that they can be executed by the scheduler presented in Section 1 and Appendix A) can be summarized as follows:

- A mobile process is encapsulated in a class.
- The class contains an array of Java $Object$s that will serve as the activation record holding the values of parameters while the process is suspended (i.e., after it yields and passes control back to the scheduler).
- Rewriting of the code in the following way:
  - $*$ At the beginning of the procedure, insert code to re-establish the values of the parameters and locals from the stored activation record.
  - $*$ After the parameter restoration code, insert a **switch** statement with empty cases (just a **break**). A later bytecode rewriting will change the breaks to jumps to route the flow of control to the resume point (the instruction immediately following the yield statement).
  - $*$ A number of labels representing the resume points. These are the labels to which the jumps of the **switch** statement will jump.
  - $*$ Code for re-establishing the local variables; this code should follow immediately after the resume point labels.
  - $*$ Code for saving the parameters and locals in an activation record before the yield point.
  - $*$ All *yield* statements replaced by **return**s.

In Section 3, we will show how this technique, along with a few further rewriting techniques, will allow us to implement the processes needed for the non-preemptive scheduler.

The technique described in [18] requires bytecode manipulation as well as source code generation. However, in [19] we describe a (more complex) technique to generate resumable code like in [18] but entirely in source code. The latter technique is preferable if the compiler generates source code and utilizes the Java compiler to generate the class files, where the former is preferable when the compiler (using a tool like ASM) generates bytecode directly. In this paper we explain much of the generated Java code in terms of source as it is much easier to read than long listings of Java bytecode operations, but also make use of the bytecode rewriting of [18]. Ultimately, the ProcessJ compiler will generate bytecode using ASM.

2.2. *JCSP and the JVM*

Since JCSP is implemented as a library in Java, a JCSP process is a Java class that extends the JCSP class *CSProcess*. When such a process is executed, for example, in a par block (which in JCSP is also a Java class), it is executed in a Java thread. Thus, a JCSP process maps directly to a Java/JVM thread. This means that JCSP processes are scheduled by the JVM (preemptive) scheduler, which also means that things like blocking I/O calls pose no problems as the JVM scheduler handles such issues without any intervention by the programmer. This is not the case for our implementation. With the implementation we present, which is single-threaded, a blocking I/O call will block the entire execution; it will cause the only execution thread in the scheduler to block. Naturally, this is something that needs to be addressed, and we give a few suggestions to possible solutions in the future work section.

2.3. *occam-π and the CCSP runtime*

As the title of the paper suggests, our goal in this work is to develop a runtime that can support the execution of millions of processes. We already know that the typical Java threading mechanism, and also other threading mechanisms like POSIX [20], are coarse grained, and since threads and operating system processes are scheduled by the operating system, user-level scheduling of some process or thread abstraction is necessary. The CCSP [7] runtime is similar; allowing occam and occam-π to execute millions of processes; the CCSP runtime also implements a non-preemptive scheduler.

ProcessJ also targets CCSP, but since CCSP is architecture dependent, it exists in parallel to what we are trying to achieve with our scheduler and process abstraction on the JVM. Generally, this provides a portability advantage. It seems unlikely, however, that we will achieve the same performance running in the JVM as the occam CCSP runtime. The Java Virtual Machine has 1) higher overhead for bytecode in terms of the amount of memory required to represent a process, 2) the relative slowness compared to optimized C executables [21] and 3) the higher overhead for runtime management. However, since we are not in direct competition with the CCSP runtime, if the JVM could execute a few million processes, then it still may be of value as a target platform for language designers.

2.4. The Missing Gotos

Java does allow labels, which an be targets for *break*s and *continue*s, but not for explicit *goto*s. Though *goto* is a reserved word in Java, programs using the *goto* keyword will not compile. However, if we had the ability to perform explicit jumps in Java source, the implementation which we describe in the next section could be done completely in source and would not require any subsequent bytecode rewriting. This approach can actually be achieved though the *JavaGoto* [22] package, which implements a custom class loader that performs the bytecode rewriting on the fly. The *JavaGoto* package marks *goto*s and labels using ‘empty’ method calls which are replaced by actual labels and *goto*s in the bytecode by the rewriting.

3. Implementation

We have already seen parts of the implementation, namely the scheduler in Section 1, so in this section we look at a number of other issues, and their solutions, associated with implementing non-preemptively schedulable processes (or self-yielding) in the JVM. We have to consider the following points:

- How do we implement a self-yielding process framework in Java?
- How do we handle parameters and locals during the time when a process is not running?
- How do we implement the yielding and resuming of a process?
- How do we handle nested procedures yielding?

We start with the process abstraction.

3.1. Processes

A process is represented by an abstract Java class called *Process*. In order to instantiate this class, its *run*() method must be implemented (typically done in a subclass of *Process*). The *run*() method is the method that the scheduler will call in order to execute the process (until it voluntarily yields or terminates). The *run*() method takes no parameters, so any parameters passed to the original procedure must be passed to the constructor of the class, which will then create an activation record with the parameters and place it on a local activation stack such that the code (in the *run*() method) can access the parameters’ values. All parameters of the original ProcessJ procedure are implemented as specially named local variables and then restored from the activation record. The contents of an activation record will be determined by a live variable analysis performed by the compiler in order to keep the size of activation record as small as possible (though, it should be mentioned that different activation records complicate the idea presented later for reusing existing activation records). See the example in Figures 2 and 3. In addition, a *finalize*() method, which is run when the process terminates, can be overridden. This allows for any clean-up that might be needed; for example, the processes in a par block use the *finalize*() method to decrement the number of running processes so that the code enclosing the par block will be set ready to run (and thus rescheduled again) when all its processes have terminated.

The *Process* class also contains the *yield*() method, which must be called when the process wants to yield and be descheduled. Before discussing the *yield*() method in more detail, we need to define the meaning of a *run label*. In the context of this paper, we refer to a run label as a label (ultimately an address) that marks a starting point of a process being scheduled. Since any process with synchronization events like channel reads and writes must yield, they give rise to run labels being placed before and after those events. In the code examples we mark a label like "L*x*:" or as a method call "*LABEL*(*x*);" where the *LABEL* is an empty method to satisfy the compiler. Note, the "L*x*:" notation will compile, but the label disappears in the class file and is just used for making locations in the source code.

The *yield*() method takes in an activation record (see Section 3.4) and the number of the next run label and then performs the following actions:

- Updates the run label of the passed-in activation.
- Adds the activation to the activation stack.
- Sets the *yielded* field (found in the *Process* class) to *true* so the scheduler knows that the process yielded and did not terminate, and thus must be added to the end of the process queue again for future scheduling.

In addition, a number of other useful methods exist in the *Process* class. Examples include: *terminate()*, which sets the field *yielded* to *false* and the *terminated* field to *true*, *terminated*() which returns *true* if the process has terminated, and methods for adding activation records to the activation stack. Finally methods for querying and setting the process ready (and not ready) are also included.

Let us consider a small ProcessJ procedure (depicted in Figure 2) which takes in an integer parameter x. Figure 3 shows the corresponding *Process* subclass (named _*foo*) and

```
proc void foo(int x) {
  chan<int> c;
  par {
    p1(c.read)
    p2(c.write, x)
  }
}
```

Figure 2. ProcessJ example.

its constructor, which takes in the ProcessJ procedure's original parameter (*x*). The '2' in the parameter list to *Activation*'s constructor call represents the total number of parameters and locals (one for x and one for the local variable *c*). In this case, the *Process* class is compatible with the implementation of the scheduler in Appendix A. Naturally the yielding and correct handling of activations and the activation stack remain, but let us now turn to par blocks.

3.2. Par blocks

A par block in ProcessJ (like the one in Figure 2) is much like the one in occam-π. The process in which a par block is executed will be blocked from running until all the processes in the par block have terminated. Like in JCSP, a par block is implemented though a *Par* class. The *Par* class contains a counter that keeps track of how many of its processes are still running and a reference to the process in which the par block appears. This reference is used

```
public abstract class _foo extends Process {
    public _foo(int x) {
        addActivation(new Activation(new Object[ ] { x } , 2));
    }

    void run() {
        // do all the work of the ProcessJ foo() procedure here (Figure 5).
    }
}
```

Figure 3. Example use of the *Process* class.

to set the process ready to run again once all its sub-process have terminated. The following steps must be taken to create and schedule a par block's sub-processes:

1. Create an instance of the *Par* class with the count 2 (representing *p1* and *p2*).
2. Create an instance of the class *_p1*, which represents *p1* while extending it with an implementation of *finalize()* (to decrement the par block's process counter). This is needed for the last terminating par block to set the enclosing process to *ready* when it terminates.
3. Do the same for *_p2*.
4. Insert both instances into the process queue.
5. Mark the enclosing process as *not ready* so that it does not get scheduled again until the par block has terminated.
6. Yield with the run label being the instruction immediately after the par block.

Figure 4 shows the outline of the code for *_p1*; *_p2* is implemented in a similar manner. (Note, there are no explicit channel ends in this code). The code for *run*() from Figure 3 is shown in

```
public abstract class _p1 extends Process {
     public _p1(Channel<Integer> c) {
        addActivation(new Activation(new Object { c }, 1));
     }

    public void run() {
        // the ProcessJ code for p1 goes here.
        terminate();
    }
}
```

Figure 4. Structure of the *_p1* class.

Figure 5. The constructor for *_p1* will create and place an activation record on the process' activation stack. The code for *run*() (of which the par block is a part) will not run again until the par block sets it ready (see the *decrement*() method in the code of Figure 6). The implementation of the *Par* class is shown in Figure 6. It should be pointed out that with the current implementation, all components of a par block must be method calls, it is not possible to do simple things like assignments[1]. Since the rewriting creates new classes in which the statements in the par block are executed, a simple assignment would cause scoping issues as

[1]The same issue exists in JCSP.

```
    void run() { // in _foo
L0:
        ...
        c = new Channel<Integer>();
        final Par _par1 = new Par(2, this);

        // this ensures that this process will not get scheduled until
        // it is marked ready by the (terminating) par block!
        setNotReady();

        // make the new processes and add them to the process queue
        Process p1 = new _p1(c) {
            public void finalize() {
                _par1.decrement();
            }
        };
        processQueue.enqueue(p1);

        Process p2 = new _p2(c, x) {
            public void finalize() {
                _par1.decrement();
            }
        };
        processQueue.enqueue(p2);

        // then yield (and return) to run the processes in the par
        yield(new Activation(new Object[] { x, c }, 2 ), 1);
        return;
L1:
        ...
        terminate();
    }
```

Figure 5. Code for *run*() in *_foo*.

the left-hand side local would not be in scope inside the created class. We do have ideas on how to resolve this issue.

3.3. Channels

Next, we focus on the implementation of a channel. No exclusive access control to the channel is needed if the runtime system is single-threaded. For a multi-threaded runtime, exclusive access control must be maintained with appropriate locks. See Section 6 for thoughts on how to implement a multi-threaded runtime and scheduler.

A templated *Channel* class with a *read*() and a *write*() method is sufficient.

Figure 7 shows the *Channel* class with the implementation of the *read*() and *write*() methods. In addition, the class contains the following four fields: a private *data* field that holds the data to be transmitted across the channel, a Boolean field *ready* that is set to true once the data has been written to the channel, and finally *reader* and *writer* which hold references to the writing and reading processes. These values are needed as the receiver needs

```
public class Par {
    // the process to schedule when done
    private Process process;

    // the number of processes in the par
    private int processCount;

    public Par(int processCount, Process p) {
        this.processCount = processCount;
        this.process = p;
    }

    public void decrement() {
        processCount--;
        if (processCount == 0) {
            // Release the process with this par block in
            process.setReady();
        }
    }
}
```

Figure 6. Implementation of the *Par* class.

to set the sender ready in order for it to be rescheduled and vice versa, depending on who got to their respective read or write first. It must be noted that the *read*() and *write*() methods of the *Channel* class may only be invoked if the *isReadyToRead*() and *isReadyToWrite*() (of Figure 7) methods have been invoked and returned *true* respectively. It is the compiler's job to ensure that sound code is generated with respect to the use of a channel's methods, that is, since there are no explicit channel ends, the compiler must make sure to generate only read code for a reader and write code for a writer.

Let us consider the implementations of the reads and writes. Simply calling *read*() or *write*() is not sufficient; this is where the compiler needs to generate code that correctly uses the *Channel* and *Process* objects. Remember, channel communication is synchronous, so the reader cannot continue past a *read*() if the writer is absent, and a writer cannot continue past a *write*() call until the reader is ready. Of course, a reader cannot monopolize the CPU if the writer is not ready. If it did, the system would deadlock as it is single-threaded. Given this situation, the reader must yield and upon re-scheduling try the read again. The reader sets itself not ready (Figure 8), so it will not be scheduled until the writer sets it ready again (Figure 7, last line of the *write* method), at which time the call *c.isReadyToRead*(**this**) (Figure 8) will return *true* and the read call will succeed. If the writer is ready, then the reader can receive the data, mark the sender ready, and continue (*read*() method of Figure 7 and Figure 8). The actual implementation (with run labels added) can be seen in Figure 8. The '2' in the yield call in Figure 8 denotes the run label L2 that marks the first line of the code; in other words, if the channel is not ready for a read (because the writer is not ready), the process is set not ready and then yields. Once the writer is ready to write the data, it will set the reader back to ready so it will get scheduled (*write*() method of Figure 7). Also note, there is no need to yield after the read happens (*b* = *c.read*(**this**)), but it would not be incorrect to do so, in which case the run label should be the next one (in this case 3).

Like the reader cannot monopolize the CPU, neither can the writer. If the reader is not

```
public class Channel< T > {
    // the data item communicated on the channel.
    private T data;
    // is there any data?
    public boolean ready = false;

    private Process writer = null;
    private Process reader = null;

    // calls to read and write must be properly
    // controlled by the channel end holders.
    public void write(Process p, T item) {
        data = item;                                          // save the data item
        writer = p;                              // hold on to the writers reference
     writer.setNotReady();    // not to be scheduled again until data has been read
        ready = true;                      // channel is ready, data has been delivered
        if (reader != null)                                 // if a reader got here first
            reader.setReady();                // set it ready so it can be scheduled again
    }

    public T read(Process p) {
        ready = false;                                          // not ready to read again!
        // we need to set the writer ready as the sync
        // has happened when the data was read
        writer.setReady();
        // clear the reader and write
        writer = null;
        reader = null;
        return data;
    }

    public boolean isReadyToRead(Process p) {
        reader = p;                                                    // set the reader
        return ready;
    }

    public boolean isReadyToWrite() {
        return !ready;
    }
}
```

Figure 7. Implementation of the *Channel* class.

ready when the writer is, the writer will set itself not ready and yield (Figure 9), and will not be rescheduled until a reader is ready and sets the writer to ready. A typical write looks like the code shown in Figure 9 (again with run labels added). If the channel is ready for a write, a data value is passed to it (in the *write*() method of Figure 7) and the process yields. This is of course necessary because all channel communication is synchronous. The *write*() call will set the process not ready, so it will not be rescheduled before the read has happened. The

```
L2:
    ... restore parameters and locals
    if (c.isReadyToRead(this)) {
        b = c.read(this);
    } else {
        // channel not ready, no need to reschedule until it is.
        setNotReady();
        yield(new Activation( ... new activation record ... , ...), 2);
        return;
    }
L3:
    ...
```

Figure 8. Reading from a channel.

```
L1:
    ... restore parameters and locals
    if (c.isReadyToWrite()) {
        c.write(this, 42);
        yield(new Activation( ... new activation record ... , ...), 2);
        return;
    } else {
        yield(new Activation( ... new activation record ... , ...), 1);
        return;
    }
L2:
    ...
```

Figure 9. Writing to a channel.

process will then yield and go back to the end of the process queue. The run label passed to the yield call should be the run label of the instruction following the write.

If the channel was not ready for a write (which could happen when implementing many-to-one and many-to-many channels) it simply yields and sets the next run label to the beginning of the **if** in the code in Figure 9; this way the write will be retried next time the process gets scheduled. It should be noted that to avoid side-effects, if the value to be sent is the result of a method call that could perform synchronizations, it should be lifted out of the $write()$ call and handled like any other method that could yield.

3.4. Activation Record and the Activation Stack

In [18], a limitation of the techniques described was that no procedure call made by a mobile process could call *suspend*(). This restriction means, that when a mobile process suspends, we only need to save its parameters and locals. This can be done by adding an *Object* array in the class that represents the Java version of the mobile process. This is a restriction that we could not have made for this work as it would be impossible to imagine that channel ends and barriers could not be passed as parameters. If, for example a process calls a procedure with the reading end of a channel, and the procedure then performs a read, then the process must yield. This means that both the process' and the called procedure's parameters and locals must be saved in (separate) activation records (one for each) and kept on an activation stack, a very similar approach taken by the JVM and any other runtime systems allowing procedure calls/method invocations.

3.5. Processes Calling Procedures

We need to consider two different cases when it comes to processes or procedures calling other procedures. If the callee does not participate in any synchronization events, no rewriting needs to take place. However, if the callee does synchronize, for example by calling a read on a channel that was passed in as a parameter, then the callee must be able to yield, and thus require rewriting in the exact same way as the caller.

What happens when the callee returns? It could be because it terminated, in which case the caller needs not take any special measures and can simply continue its execution until it either terminates or yields because of a synchronization event. If the callee terminated, it would not have left an activation record on the activation stack. If it yielded, it would have left an activation record on the activation stack, and the caller must therefore also yield and add its activation record to the front of the activation stack (really, the activation stack is not a stack but a linked list). Consider the small example in Figure 10 (A larger ProcessJ example imple-

```
proc int g(chan<int>.read in) {
    int result = c.read();
    return result;
}

proc void p1(chan<int>.read in) {
    int a;
    a = g(in);
    ...
}
```

Figure 10. ProcessJ process calling a procedure.

menting the *Prefix* process from the CommsTime benchmark can be found in Appendix B). The *read*() in *g*() will yield, which mean that *p1*() must yield. When rescheduled, the flow of control must make it through *p1*() back to *g*() and re-attempt the read. We have already seen how to accomplish the rewriting in *g*(). The code for *p1*() is shown in Figure 11. As we can see, if *g*() yielded, *p1*() will yield and resume at L1 the next time *p1*() is scheduled; when *g*() is called on subsequent invocations (because it yielded), it will behave correctly as it was rewritten according to the techniques described earlier. The code following L0 establishes the activation record that *g*() will read when called the first time.

3.6. Alt

No process oriented programming language is complete without a way to alternate over (input) channel ends. ProcessJ implements the classic alt from CSP, though only with input guards for now (output guards, timers and barriers are reasonably simple to implement in a single-threaded runtime as well). A ProcessJ example of an alt alternating over two input channels can be seen in Figure 12. Since the scheduler is single-threaded, implementing an alt can be done by simply picking one of the ready channels. Once a writer has committed

```
    ...
L0:
    addActivation(new Activation(new Object[ ] { c }, 2));
L1:
    a = g();
    if (yielded) {
        yield(new Activation(new Object[ ] { c, a }, 2), 1);
        return;
    }
    ...
    terminate();
```

Figure 11. Implementation of *p1*().

```
public proc void bar(channel<int>.read in1,
                     channel<int>.read in2) {
    int x;
    alt {
        x = in1.read() : { ... }
        x = in2.read() : { ... }
    }
}
```

Figure 12. ProcessJ alt example.

to a channel synchronization, the channel is ready, and the alt construct can pick any of the ready channels. The implementation of the *Alt* class is shown in Figure 13. The *Alt* class

```
public class Alt {
    private Object[ ] channels;
    public Alt(Object... channels) {
        this.channels = channels;
    }

    // -1: no channels are ready
    public int getReadyIndex() {
    for (int i=0; i<channels.length; i++) {
        if (((Channel<? extends Object>)channels[i]).readyToReadAlt())
            return i;
        }
        return -1;
    }
}
```

Figure 13. The *Alt* class.

is instantiated with an array of channels and the *getReadyIndex*() method returns the index of a ready channel. Note, The method *readyToReadAlt*() simply checks if the channel has a writer committed to the communication yet. This method was added to the *Channel* class to accommodate the implementation of the alt.

Figure 14 shows code that uses the alt. After constructing it, the *getReadyIndex* method returns an index which is used for switching. If the method returns -1, and therefore does not match any of the cases, the default case is executed, and the process yields and retries next time it is scheduled.

```
    ...
L1:
    if (runLabel == 1) {
        // restore locals
        runLabel = -1;
    }
    Alt alt = new Alt(in1, in2);
    switch(alt.getReadyIndex()) {
    case 0: // in1
        in1.isReadyToRead(this);
        x = in1.read(this);
        ...
        break;
    case 1: // in2
        in2.isReadyToRead(this);
        x = in2.read(this);
        ...
        break;
    default: // no ready channels
        yield(new Activation(new Object[] { in1, in2, x }, 3), 1);
        return;
    }
    ...
```

Figure 14. Code using the *Alt* class.

4. Runtime Tests

Reporting on correctness tests is of course rather difficult, but to get an idea of the capabilities of the system we have implemented, we have run a number of tests on two different machines (with different amounts of memory). The two execution architectures on which we tested are

- Mac Pro 4.1, OS X Snow Leopard, Intel i7 Quad-core Xenon 2.93 MHz with 8GB RAM. On this machine we managed to run tests with up to 22,000,000 processes
- AMD dual 16 core Opteron 6274 (2.2 GHz) with 64GB 1,333 MHz DDR3 ECC Registered RAM running CentOS 6.3 (Linux 2.6.32). On this machine we managed to run tests up to 100,000,000 processes.

Though it should be noted, the runtime system (the scheduler) uses just one core.

We tested a simple program with a reader and a writer running in parallel communicating one integer value. The reader calls a procedure to do the send, and the writer reads the channel directly (see Figure 15). The **par for** will add $2 * n$ processes to the process queue so it reaches maximum size (which is $2 * n + 1$) before running anything. The way the process queue is filled, each reader will be scheduled (and yield) before its writer, so each reader and each writer will be scheduled twice (thus go around the process queue twice). The reason for stopping the tests at 22,000,000 and 100,000,000 respectively was simply a question of degrading performance. When it was obvious that there was so much overhead because of paging or other memory issues, we stopped increasing the number of processes. It might be possible to run bigger samples on both machines, but the times would be unacceptably poor. It should be noted that the channel *c* is not shared between instances of the inner par block, but only between the two processes in the par block. We are still working on the syntactic issues associated with parallel for loops and similar things. The times (real, user and system) reported where obtained with the OS X/Linux shell built-in command `time`. The Mac with 8GB RAM runs Java version 1.6.0_26, and the -Xmx flag was given the value 8G. The AMD machine with 64GB RAM runs Java version 1.6.0_24, and the -Xmx flag was given the value 64G. To test the maximum number of processes that can be run in *one single-threaded* JVM,

```
proc int g(chan<int>.read in) {
    int result = c.read();
    return result;
}

proc void p1(chan<int>.read in) {
    int a;
    a = g(in);
}

proc void p2(chan<int>.write in) {
    c.write(42);
}

proc void main(int n) {
    par for (int i=0; i<n; i++) {
        chan<int> c;
        par {
            p1(c.read);
            p2(c.write);
        }
    }
}
```

Figure 15. ProcessJ test program.

and to gain some knowledge about performance, we have executed the code, obtained by rewriting the above ProcessJ code, ten times for each value of *n* (starting at 2,000 all the way up to 22,000,000 for the Mac and 100 million for the AMD). The time reported is the best of each of the 10 runs. We ran the test on the Mac and the AMD for 2,000, 20,000, 100,000, 200,000, 1,000,000, 2,000,000 to 22,000,000 in steps of 2,000,000, and continued to 100,000,000 on the AMD, first in steps of 2,000,000 to 30,000,000 and then in steps of 10,000,000 all the way to 100,000,000. The time shown in Figures 16, 17, 18, 19, and 20 are all based on the *real* time (i.e., the wall clock time) from Table 1.

Table 1. Time measurements in seconds.

	Mac				AMD			
Number of Processes	Real Time	User Time	System Time	Total Time	Real Time	User Time	System Time	Total Time
2,000	0.179	0.225	0.033	0.258	0.141	0.15	0.052	0.202
20,000	0.288	0.412	0.040	0.452	0.343	0.445	0.058	0.503
100,000	0.715	1.981	0.188	2.169	0.475	0.846	0.076	0.922
200,000	1.043	2.976	0.208	3.184	0.550	0.898	0.090	0.988
1,000,000	3.678	11.745	0.494	12.239	2.456	21.233	4.063	25.296
2,000,000	6.900	21.422	0.849	22.271	9.042	42.812	8.196	51.008
4,000,000	13.559	42.631	1.567	44.198	22.588	88.304	18.900	107.204
6,000,000	20.312	64.512	2.316	66.828	28.217	150.380	26.150	176.53
8,000,000	27.019	86.494	3.069	89.563	35.703	276.555	32.986	309.541
10,000,000	33.967	108.940	3.765	112.705	53.858	343.264	31.407	374.671
12,000,000	40.998	132.855	4.580	137.435	65.226	483.807	37.344	521.151
14,000,000	48.005	155.932	5.595	161.527	90.195	569.434	43.174	612.608
16,000,000	72.199	185.503	9.891	195.394	103.211	676.649	41.916	718.565
18,000,000	116.668	211.246	11.970	223.216	98.041	718.234	51.715	769.949
20,000,000	552.964	279.982	25.049	305.031	166.047	1148.218	63.168	1211.386
22,000,000	767.245	336.626	35.167	371.793	174.428	1235.173	71.503	1306.676
24,000,000	—	—	—	—	175.481	1,236.130	81.863	1,317.993
26,000,000	—	—	—	—	235.759	1,435.805	85.566	1,521.371
28,000,000	—	—	—	—	243.219	1,611.822	77.575	1,689.397
30,000,000	—	—	—	—	235.587	1,647.064	89.196	1,736.260
32,000,000	—	—	—	—	326.800	1,965.065	107.430	2,072.495
34,000,000	—	—	—	—	319.788	2,045.741	102.725	2,148.466
36,000,000	—	—	—	—	321.788	2,038.874	108.610	2,147.484
38,000,000	—	—	—	—	337.983	2,283.136	105.309	2,388.445
40,000,000	—	—	—	—	344.113	2,320.590	99.907	2,420.497
50,000,000	—	—	—	—	447.756	2,903.146	133.154	3,036.300
60,000,000	—	—	—	—	589.306	3,750.600	162.203	3,912.803
70,000,000	—	—	—	—	753.439	4,362.060	221.815	4,583.875
80,000,000	—	—	—	—	772.803	4,746.371	287.332	5,033.703
90,000,000	—	—	—	—	1,131.736	5,792.862	332.786	6,125.648
100,000,000	—	—	—	—	2,750.785	6,921.055	271.813	7,192.868

Figure 16 illustrates the real runtime for the program with the above mentioned number of processes. Figure 17 illustrates the time in microseconds that each par of processes (*p1*() and *p2*()) took to execute (including the communication). As we can see, the best performance of 7 microseconds per pair of processes happened in the range of 1,000,000 - 14,000,000. We also ran the same program on an AMD processor with 64GB, and here we managed to run as many as 100,000,000 processes (which was about on the limit of what the machine could handle), and the results can be see in Figures 18 (for 2,000 to 22,000,000 processes) and 19 (for 22,000,000 to 100,000,000 processes). The per processor pair (one sender and one receiver) cost on the AMD machine seems higher than on the Mac; we cannot say why at this time. Figure 20 illustrates these numbers. The sweet spot seems to be in the range of 1,000,000 to around 90,000,000 processes. Like with the Mac, a certain start-up cost must be paid to start the JVM. To get an idea of what part of the program takes what amount of time, we investigated the 20,000,000 processes case. It turns out that setting up the process queue (without having executed anything yet) takes 61 seconds (6.117 microseconds per

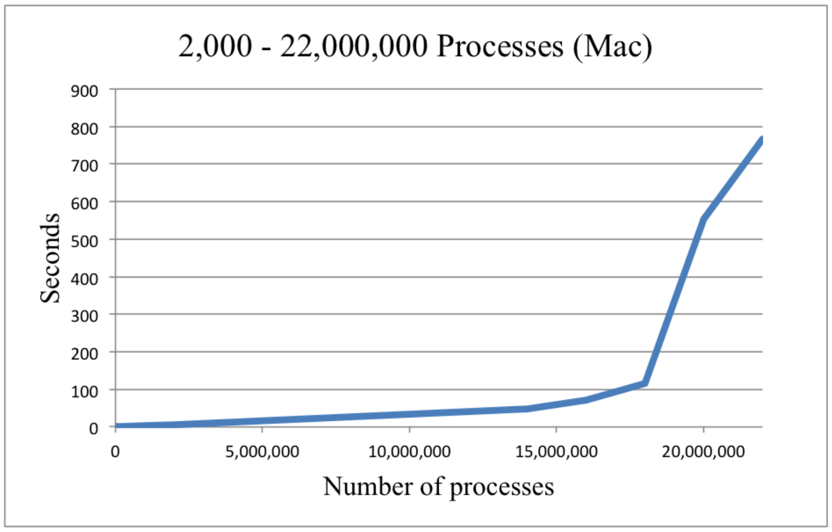

Figure 16. 2,000 - 22,000,000 processes on Mac with 8GB RAM.

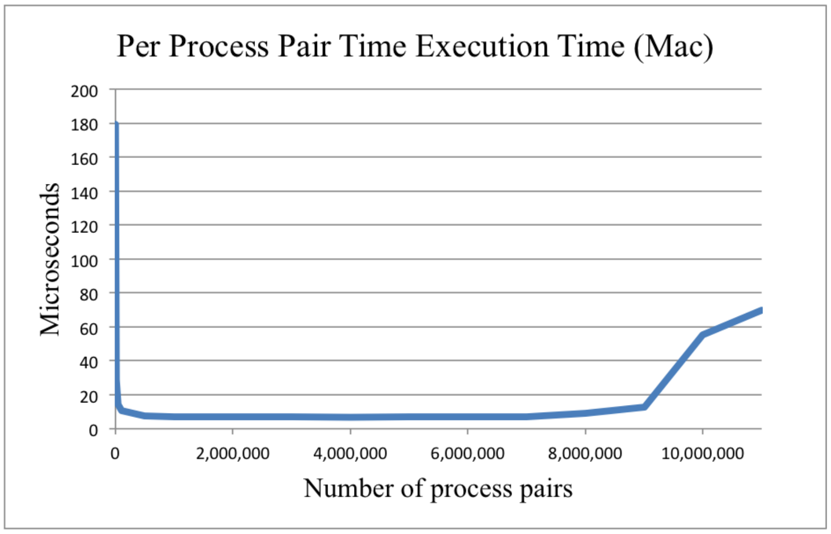

Figure 17. Execution time in microseconds on a per process pair basis (Mac).

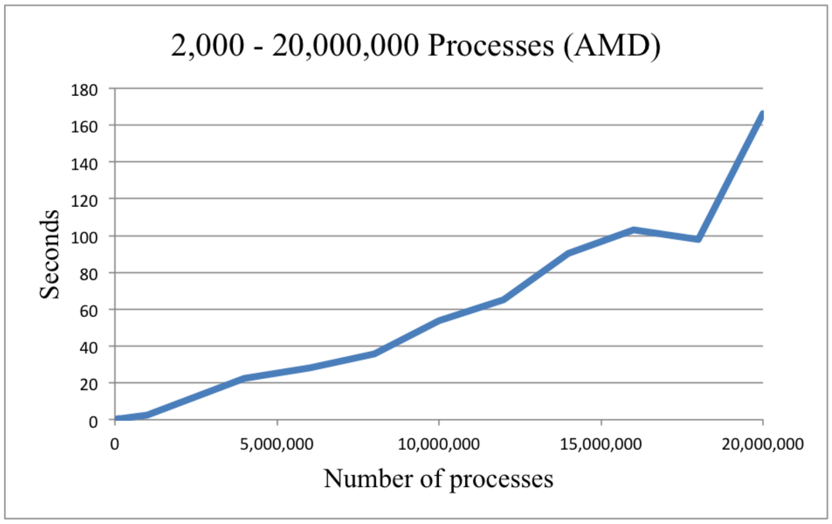

Figure 18. 2,000 - 20,000,000 processes on AMD with 64GB.

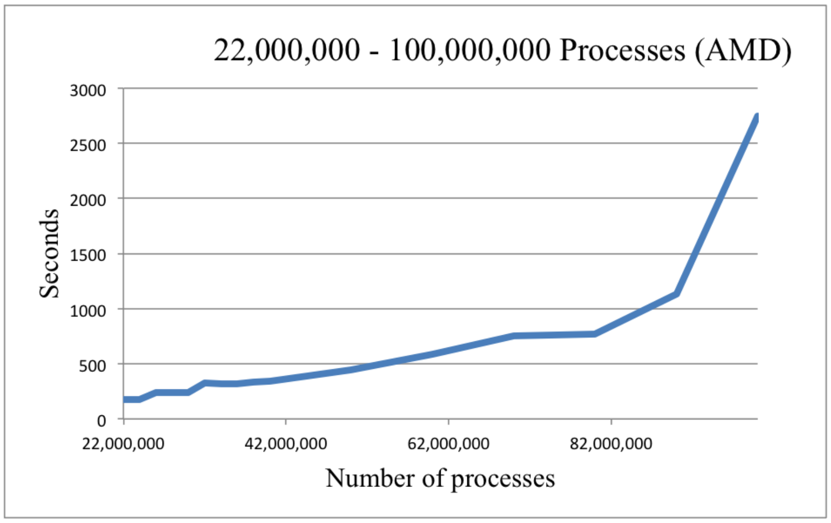

Figure 19. 22,000,000 - 100,000,000 processes on AMD with 64GB.

process pair) and the actual execution time takes 106 seconds which is 10.57 microseconds per process pair. In Table 1, it is interesting to note that the equation:

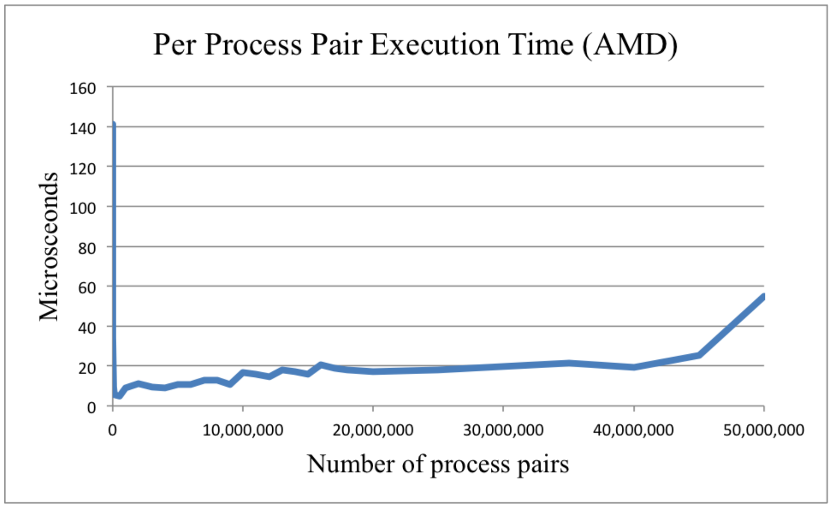

Figure 20. Execution time in microseconds on a per process pair basis (AMD).

$$Real\ Time = User\ Time + System\ Time$$

does not hold. As a matter of fact, the real time is much lower than the total time, which is the user time plus the system time. Figures 21 and 22 show the real time for both architectures as well as their user + system time. An obvious observation is, that the user + system time moves more rapidly away from the real time as the number of processes grow, and even more so on the AMD architecture.

A number of questions arise from this graph and these numbers: firstly, if the program is single-threaded (which it is), how come the real time is not equal to the user + system time (or at least not just off by a small constant factor). The best answer we can give is that the JVM might do things like garbage collection in a separate thread, which will count towards more user + system time, but run concurrently with the program, thus not impacting the real time. This might also explain the bigger gap as the number of processes grows and the bounds of the memory are reached: more and more frequently the garbage collector is started. Secondly, it seems strange that the user + system time on the AMD is much bigger than on the Mac for the same number of processes. Again, why we do not know. It should be

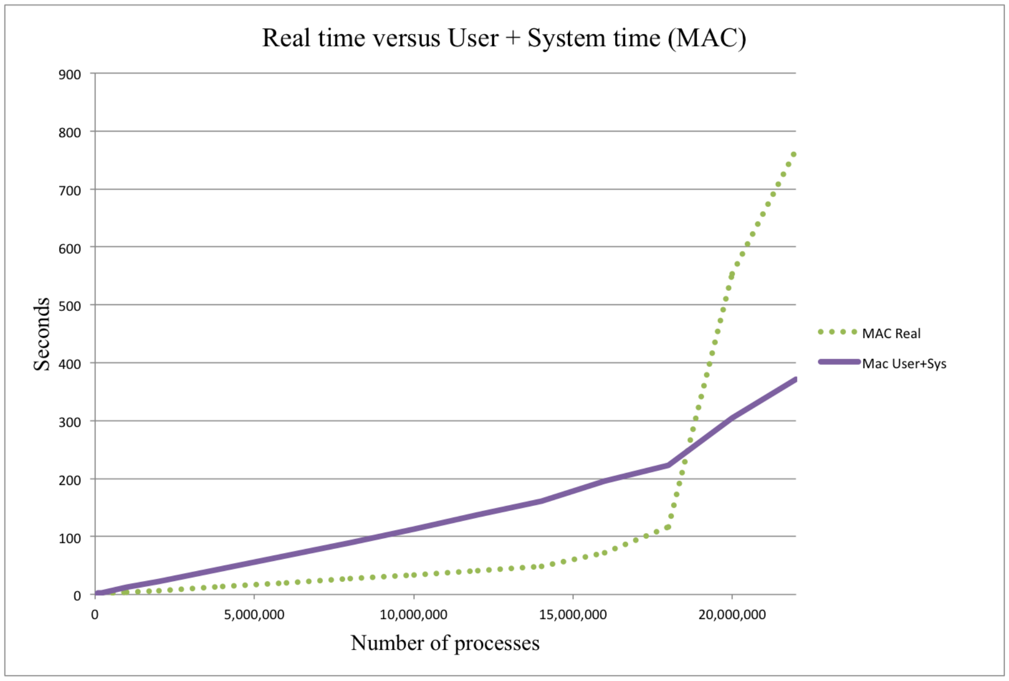

Figure 21. Real time versus user + system time on the Mac.

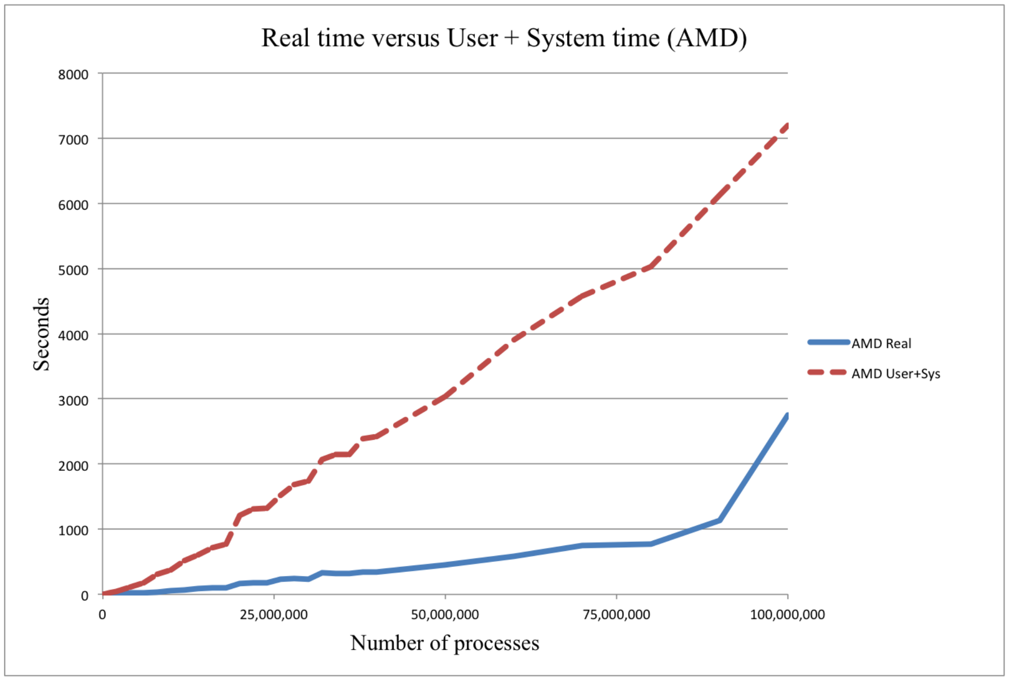

Figure 22. Real time versus user + system time on the AMD.

noted, that if we look at the real time for numbers of processes that can be executed on both architectures (see Figure 23), they remain fairly close up to 16,000,000 processes (again with a small disadvantage for the AMD architecture) until the Mac runs out of physical memory.

4.1. CommsTime

To estimate the time it takes to perform a channel communication, we have implemented the CommsTime benchmark (Figure 24 on page 20 illustrates the process network). The CommsTime benchmark consists of 4 sub-processes (*prefix*, *consumer*, *delta*, and *succ*) and for each number consumed by the consumer process, 4 channel communications must happen. The

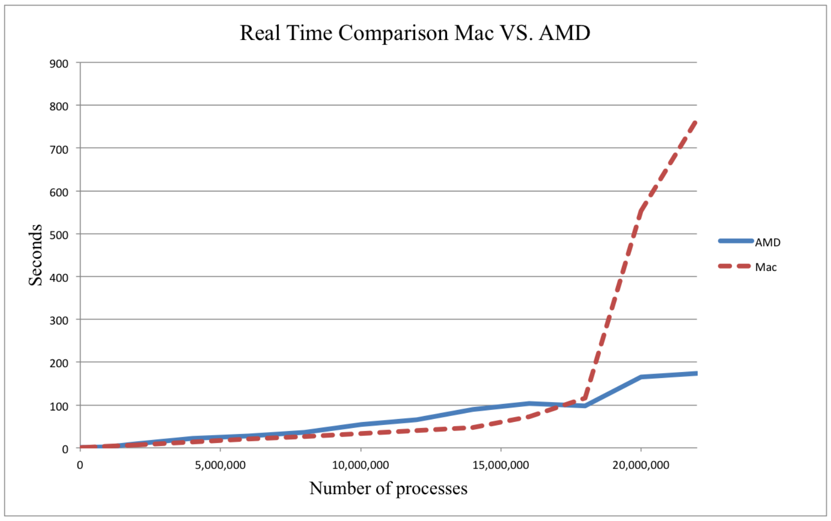

Figure 23. Real time for both Mac and AMD for 2,000 - 22,000,000 processes.

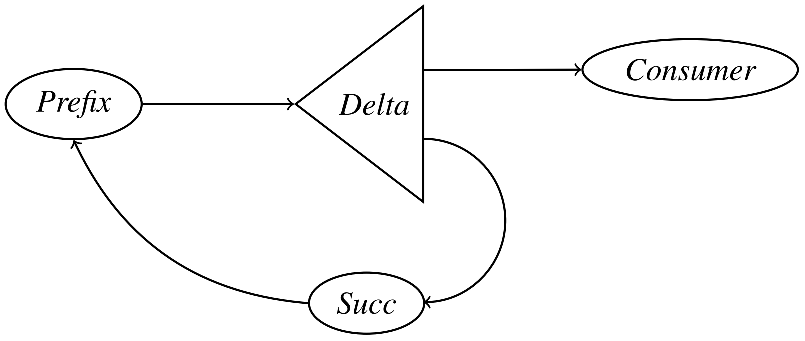

Figure 24. The CommsTime network.

Table 2. Commstime results.

	Mac / OS X		AMD / Linux	
	LiteProc	JCSP	LiteProc	JCSP
μs / iteration	9.26	27.0	13.56	136.0
μs / communication	2.31	6.0	3.9	35
μs / context switch	1.322	3.0	1.937	17

scheduler reports that each communication requires 7 context switches (though the JCSP version reports 8 context switches). The CommsTime benchmark was executed 30 times on each architecture and the best time achieved was reported. The consumer process prints out every 10,000 steps, and the entire benchmark was run for exactly 120 seconds and then stopped. The number reported is the largest number printed out within the allotted 120 seconds (i.e., the estimate might be off by as much as 10,000 steps). We also ran the SeqCommsTime demo program included in the JCSP 1.1rc4 version. Table 2 lists these the measured times on both the Mac and the AMD machine for our system (labelled LiteProc) and JCSP. As the results in Table 2 show, our rewriting techniques along with the very naïve single-threaded scheduler achieves a performance that is almost exactly 3 times as fast as the standard JCSP implementation.

There are a number of optimizations that could be performed to make our approach run even faster. I the current implementation every time an activation record is created (before a yield point), a new one is allocated; the existing one is still available and could simply be updated for the locals that have changed. This can all be determine though a variable analysis on the source code.

4.2. Alt

We tested the alt with a simple two producers and a single consumer multiplexer system where the consumer received 10,000,000 messages (the producers were willing to communicate as many messages as the consumer could consume). On the Mac the average time per communication was 1.32 microseconds, and on the AMD it was 2.02 microseconds.

5. Conclusion

In this paper we have presented a code generation and bytecode rewriting technique for implementing ProcessJ processes in the JVM by the millions. We have also presented a simple scheduler that can be used to execute the process in the JVM.

We have successfully tested the system and shown that, on a regular office workstation, it is feasible to run (translated ProcessJ) programs with tens of millions of processes. In addition we succeeded in executing 100,000,000 processes on a 64GB memory AMD machine. The main limitation to performance is the amount of memory available on the system.

We see that the Mac tops out at approximately 18,000,000 processes, and since the AMD has 4 times as much memory, we would expect it to start showing extremely poor performance at around $4*18{,}000{,}000 \approx$ 70-80,000,000 processes, and indeed it does (performance starts to seriously decline at around 80,000,000).

6. Future Work

The system we have presented in this paper utilizes a rewriting technique and implementation that takes advantage of only a single operating systems thread in the JVM. That is of course, in the long run, not acceptable. Ultimately, we want an n core CPU to always execute n schedulers in separate JVM threads. A initial, but probably not very scalable, solution is outlined here; we need to make two changes to the system:

1. Rather than having just one instance of the process queue, we create n instances, and rather than just running one copy of the scheduler, we start n Java threads each running a copy of the scheduling algorithm and each being associated with its own process queue.
2. Since access to a channel (and other synchronization entities), can no longer be guaranteed to be exclusive, race conditions can now happen. The obvious, and correct solution is to re-implement the classes representing the synchronization entities like channels and barriers to only allow exclusive access. The various methods can be declared *synchronized* and access to whole objects can be protected by locks using *synchronized blocks*. It should be noted, implementing a channel using the *wait-notify* paradigm in Java is not possible as both *wait* and *notify* operate on Java threads. The current implementation where the sender sets the reader back to ready and vice-versa is a simple implementation of the wait-notify paradigm in our system.

It should also be noted that this approach will not increase the number of processes that can be run on a single JVM (single or multi-threaded) as this number is determined by the memory size of the machine on which the JVM runs. However, in theory, adding more cores and thus more process queues and schedulers, we should see a speed up in execution time; at least until the runtime is significantly impacted by page faults and the fact that the JVM heap exceeds the total physical memory size.

Rather than using a linked list to hold the activation records for each process, an array could be used; the amount of space taken up by just one linked link node object could easily cover the space of an array with at least a couple of indices. Whether this would greatly improve performance is not known, but it is worth exploring.

As mentioned in Section 4, the way activation records are created and maintained could be optimized; before a yield, we now create a brand new activation record (array of *Objects*) rather than simply reusing and updating the one that already exists.

Another optimization, which could be done to the single-threaded runtime, is to split the process queue into two queues: one for the ready and one for the not ready processes. Doing this also eliminates the *notReadyCounter* from the scheduling code as a deadlock has happened when the queue with the ready processes is empty and the one with the not-ready ones is not empty. Naturally, this is not necessarily as easy as it first appears, but it should be reasonably straightforward to implement what we can call a 'naïve' version of a multi-threaded scheduler. There is really no need for the not ready processes to cycle through the process queue if they cannot be run. When a not ready process becomes ready it gets moved to the ready queue and eventually gets run; when a process become not ready, it gets moved to the not ready queue. Alternatively, like both JCSP and CCSP, non-ready processes could be held by the channel on which they are blocking. When the channel communication happens, the writer can reschedule the reader (if the reader arrived at the channel communication first), or the reader can reschedule the writer (if the writer got there first).

One last issue that eventually needs to be addressed is blocking I/O. If a procedure blocks in an I/O call no further progress will be made until the procedure unblocks. This can be a difficult issue to tackle with a single-threaded scheduler, however with a multi-threaded scheduler a possible solution could be to allocate one or more threads in the scheduler to deal with procedure calls that can block (e.g. blocking I/O). Alternatively, processes performing I/O could be executed in real JVM threads and thus scheduled by the JVM. Appendix C illustrates how this could be achieved.

We are fully aware that multi-core scheduling is a complicated and eventually some of the techniques described in [23] might be adaptable to our approach.

Acknowledgements

We would like to thank the reviewers who clearly spent a lot of time reading the paper and making suggestions on how to improve the contents as well as the presentation. Also, we would like to thank the paper-specific editor, Kevin Chalmers, for re-reading the paper and making sure we got everything just right, and Peter Welch for quickly looking over the alt section, which was added after the paper was initially reviewed.

References

[1] Amit Rathore. *Clojure in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2011.

[2] D. Koenig, A. Glover, P. King, G. Laforge, and J.Skeet. *Groovy in Action*. Manning Publications Co., 2007.

[3] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[4] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 1st edition, 2011.

[5] Samuele Pedroni and Noel Rappin. *Jython Essentials*. O'Reilly, Beijing, 2002.

[6] Jan Bækgaard Pedersen and Marc L. Smith. ProcessJ: A Possible Future of Process-Oriented Design. In Peter H. Welch, Frederick R. M. Barnes, Jan F. Broenink, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2013*, pages 133–156, November 2013.

[7] James Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[8] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes: introducing occam-pi. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[9] Frederick R. M. Barnes and Peter H. Welch. Communicating Mobile Processes. In Ian East, Jeremy Martin, Peter H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, volume 62, WoTUG-27 of *Concurrent Systems Engineering Series, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[10] Peter H. Welch and Frederick R. M. Barnes. Mobile Barriers for occam-π: Semantics, Implementation and Application. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, volume 63, WoTUG-28 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[11] Peter H. Welch and Frederick R. M. Barnes. A CSP Model for Mobile Channels. In *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 17–33, Amsterdam, The Netherlands, September 2008. IOS Press. ISBN: 978-1-58603-907-3.

[12] E. Bruneton and R Lenglet and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, November 2002.

[13] Peter H. Welch, Neil C.C. Brown, James Moores, Kevin Chalmers, and Bernard Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[14] Peter H. Welch and Paul D. Austin. *Communicating Sequential Processes for Java (JCSP) Home Page*. Systems Research Group, University of Kent, 2010. `www.cs.kent.ac.uk/projects/ofa/jcsp`.

[15] Peter H. Welch. Java Threads in the Light of occam/CSP. In Peter H. Welch and André W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[16] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 249–266, July 2007.

[17] G. Ritson, Adam T. Sampson, and Fred R. M. Barnes. Multicore scheduling for lightweight communicating processes. In John Field and Vasco Thudichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, 2009.

[18] Jan B. Pedersen and Brian Kauke. Resumable Java Bytecode - Process Mobility for the JVM. In *The thirty-second Communicating Process Architectures Conference, CPA 2009, organised under the auspices of WoTUG, Eindhoven, The Netherlands, 1-6 November 2009*, pages 159–172, 2009.

[19] Matthew Sowders and Jan B. Pedersen. Mobile Process Resumption In Java Without Bytecode Rewriting. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, July 2011.

[20] The IEEE and The Open Group. POSIX Threads: IEEE Std 1003.1, 2013 Edition, 2013. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`.

[21] The Computer Benchmark Games. `http://benchmarksgame.alioth.debian.org`.

[22] Javagoto - Goto for your Java Programs. `http://javagoto.com`.

[23] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009. `http://www.cs.kent.ac.uk/pubs/2009/2928/`.

Appendix A. The Scheduler

For completeness, the Java code that implements the simple scheduler used for the tests in this paper is illustrated in Figure 25.

```
public static void schedule(ProcessQueue pq[ ]) {
    int notReadyCounter = 0;
    while (pq.size() > 0) {
        // grab the next process in the process queue
        Process p = pq.dequeue();
        // is it ready to run?
        if (p.ready()) {
            // reset the notReadyCounter
            notReadyCounter = 0;
            // yes it was ready, so run it
            p.yielded = false;
            p.run();
            // and reset the notReadyCounter
            notReadyCounter = 0;
            // did the process terminate
            if (!p.terminated()) {
                // did not terminate, so insert in process queue
                pq.enqueue(p);
            } else {
                // finalize before terminating the process
                p.finalize();
            }
        } else {
            // no, not ready, put it back in the process queue
            // and count it as not ready
            pq.enqueue(p);
            notReadyCounter++;
        }
        // if we have seen all the processes
        // and none were ready we have a deadlock
        if (notReadyCounter == pq.size() && pq.size() > 0) {
            System.out.println("No processes ready to run." +
                               "System is deadlocked");
            System.exit(1);
        }
    }
}
```

Figure 25. Java code for the simple scheduler.

Appendix B. CommsTime in ProcessJ

This section shows the ProcessJ version of the CommsTime benchmark and illustrates, in detail, the implementation of the *Prefix* process; we show the equivalent Java code and byte code rewriting.

```
import std.io;

public proc void Prefix(int init, chan<int>.read in,
                        chan<int>.write out) {
  out.write (init);
  while (true) {
    int val = in.read();
    out.write(val);
  }
}
public proc void Delta(chan<int>.read in, chan<int>.write out1,
                       chan<int>.write out2) {
  while (true) {
    int val = in.read();
    out1.write(val);
    out2.write(val);
  }
}
public proc void Succ(chan<int>.read in, chan<int>.write out) {
  while (true) {
    int val = in.read();
    out.write(val + 1);
  }
}
public proc void Consumer(chan<int>.read in) {
  while (true) {
    int val = in.read();
    if (val % 10000 == 0)
      println(val);
    }
}
public proc void CommsTime() {
  chan<int> a,b,c,d;
  par {
    Prefix(0, a.read, c.write);
    Delta(a.write, d.read, b.read);
    Succ(b.read, c.write, a.write);
    Consumer(d.read);
  }
}
```

Figure 26. The *Prefix* process in ProcessJ.

Figure 27 shows the code generated for the *Prefix* process. This code was slightly optimized by moving the **return** statement (implemented as **if** (TRUE) **return**;, see the explanation below) out of both branches of the channel-write code. Once the code has been compiled, the cases in the **switch** statement (marked with a *) must be adjusted to jump to the locations of the *LABEL()* method. For clarity we add a call to an empty method *Goto()*. *LABEL()* is also just an empty method used to mark the location of the labels and are explained in Appendix B. These are marked with a + in Figure 27.

```
    public abstract class _Prefix extends Process {
        public _Prefix(int init, Channel<Integer> in, Channel<Integer> out) {
            addActivationFirst(new Activation(new Object[ ] {init, in, out}, 4));
        }

        public void run() {
            /** Declare parameters */
            int $init = 0;
            Channel<Integer> $in = null;
            Channel<Integer> $out = null;
            Activation activation = getActivation();
            int runLabel = activation.getRunLabel();
            /** Restore parameters */
            $init = (Integer)activation.getLocal(0);
            $in = (Channel<Integer>)activation.getLocal(1);
            $out = (Channel<Integer>)activation.getLocal(2);
            /** Jump */
            switch (runLabel) {
*           case 0: Goto(0); break;
*           case 1: Goto(1); break;
*           case 2: Goto(2); break;
*           case 3: Goto(3); break;
*           case 4: Goto(4); break;
            }
+           LABEL(0);
            if ($out.isReadyToWrite()) {
                $out.write(this, $init);
                yield(new Activation(new Object[] {$init, $in, $out}, 4), 1);
            } else {
                setNotReady();
                yield(new Activation(new Object[] {$init, $in, $out}, 4), 0);
            }
            if (TRUE) return;
+           LABEL(1);
            while (TRUE) {
                int val = 0;
+               LABEL(2);
                if (runLabel == 2) {
                    val = (Integer)activation.getLocal(3);
                    runLabel = -1;
                }
                if ($in.isReadyToRead(this)) {
                    val = $in.read(this);
                } else {
```

```
                setNotReady();
                yield(new Activation(new Object[] { $init, $in, $out, val}, 4), 2);
                if (TRUE) return;
            }
+           LABEL(3);
            if (runLabel == 3) {
                val = (Integer)activation.getLocal(3);
                runLabel = -1;
            }
            if ($out.isReadyToWrite()) {
                $out.write(this, val);
                yield(new Activation(new Object[] {$init, $in, $out, val}, 4), 4);
            } else {
                setNotReady();
                yield(new Activation(new Object[] {$init, $in, $out, val}, 4), 3);
            }
            if (TRUE) return;
+           LABEL(4);
            if (runLabel == 4) {
                val = (Integer)activation.getLocal(3);
                runLabel = -1;
            }
        }
        terminate();
    }
}
```

Figure 27. Java implementation of the *Prefix* process.

Labels

To better illustrate the location of the label locations we have used an empty method **void** *LABEL*(**int** *label*) { }. The code generated for these invocations has one of two shapes (as shown in Figures 28 and 29). The code in Figure 28 has a label before the three lines associated with the dummy *LABEL*() invocation. In this situation, the label *Label216* can be substituted for case 1 (line 217 shows that it is label 1 as the constant 1 is loaded onto the stack) in the switch statement. In Figure 29 there is no label before the *LABEL*() invocation, so a label needs to be inserted and this new label goes in case 2 in the switch statement. Now lines 216, 217, 218, 260, 261, and 262 can be removed.

```
Label216:
  .line 54
  216: aload_0
  217: iconst_1
  218: invokevirtual _Prefix/LABEL(I)V
```

Figure 28. *Label*() invocation with preceding bytecode label.

```
258: istore 6
.line 62
260: aload_0
261: iconst_2
262: invokevirtual _Prefix/LABEL(I)V
```

Figure 29. *Label*() invocation without preceding bytecode label.

Switch Statement

The switch statement generated by the Java compiler looks like the one shown in Figure 30. The labels are incorrect, and need to be replaced by the ones we discussed in the previous section. The old labels and the code associated with them (which follows immediately after the switch) can be removed from the bytecode (though it will never get executed). Also, the default case will never be executed. Note, in a `tableswitch` instruction the two numbers following the instruction denote the low and the high key, thus `Label76` represents the value 0, `Label79` the value 1 and so on.

```
41: tableswitch 0 4
  Label76
  Label79
  Label82
  Label85
  Label88
  default : Label88
```

Figure 30. Switch statement in bytecode.

Returns

To trick the compiler to not generate errors about unreachable code, we implemented the **return** following the *yield*s as "**if** (TRUE) **return**;" (TRUE is a final Boolean field with the value **true**), and that generates bytecode like the code shown in Figure 31. Lines 163, 164

```
163: aload_0
164: getfield _Prefix2/TRUE Z
167: ifeq Label216
170: return
```

Figure 31. Generated bytecode for **return**.

and 167 can simply be removed, leaving the **return** as the only instruction.

Appendix C. Blocking Calls

As mentioned, blocking calls to for example library routines etc. can be a problem with a single-threaded scheduler. In this section we describe a simple way to solve this problem by utilizing Java/JVM threads to execute blocking code. We start with the *Blocking* class shown in Figure 32. This class extends the Java *Thread* class, and can therefore be executed separately by the JVM scheduler. If a blocking call is to be executed, it can be placed in the *runit*() method of an instance of the *Blocking* class and run by invoking the *start*() method. This will in turn invoke the *run*() method which will first invoke the *doit*() method and then set the process back to ready. The code with the blocking call needs to create an instance

```
public abstract class Blocking extends Thread {
    protected Process pr;
    protected Object returnValue = null;
    public Blocking(Process p) {
        this.pr = p;
    }

    public abstract void doit();

    public void run() {
        doit();
        pr.setReady();
    }

    public Object getReturnValue() {
        return returnValue;
    }
}
```

Figure 32. The *Blocking* class.

of the *Blocking* class, but also set itself not ready to run. This means that the process is not scheduled again before it is marked ready. Marking it ready is the job of the *Blocking* class; more specifically, the last line of the *run*() method calls *setReady*() on the process. Figure 33 illustrates the use of the *Blocking* class; note, after starting the JVM thread to execute the *Blocking* object's *run*() method, the process must yield. When finally ready again, execution continues immediately after the *yield*() call. If the *doit*() method set the *returnValue* field,

```
...
this.setNotReady();
Blocking b = new Blocking(this) {
    public void doit() {
        // do the blocking call here
        // set the field returnValue if needed
    }
};
b.start();
yield(new Activation( ... new activation ..., ...), 1);
LABEL(1);
// return value available in b.getReturnValue()
...
```

Figure 33. Example use of the *Blocking* class.

it can be accessed through a reference to the *Blocking* object. In the line after the label in Figure 33, the expression *b.getReturnValue*() would retrieve the return value of the blocking call.