Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

JVMCSP - Approaching Billions of Processes on a Single-Core JVM

Cabel SHRESTHA and Jan Bækgaard PEDERSEN [1],
Department of Computer Science, University of Nevada, Las Vegas, USA

Abstract. In this paper we present the JVMCSP - a runtime system for the JVM and a code generator in the ProcessJ compiler. ProcessJ is a new process-oriented language with a Java-like syntax and CSP semantics. ProcessJ compiles to a number of different runtimes and in this paper focuses on the JVM runtime. The approach followed in the implementation is inspired by previous prototype-work we have done, but in this paper we closely look at the actual implementation and how it differed from our previous assumptions. We also present a number of results that highlight the capabilities of our code generator and runtime. We show that the runtime has a low overhead and we managed to run a program on a single core with 480,900,001 processes and a total of over 1.4 billion runtime objects on the JVM heap.

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

Introduction

A few years ago we started the development of a new process-oriented language called ProcessJ [1]. The two cornerstones of the design of ProcessJ are CSP [2] semantics and Java syntax [3] (without objects). Naturally, a process-oriented language is based on CSP (and possibly the π-calculus for mobility), but the reason for choosing a Java-like syntax is for familiarity and adoption rate. A language that looks and, mostly, behaves like a known language is easier to approach and learn. So, if a programmer already knows Java, then we only have to teach them about process-orientation. In addition to these considerations, we also wanted to make the language portable across many architectures, and we are currently working on several different execution platforms; in this paper we present the results of developing a Java code generator and a JVM runtime system with a simple scheduler (overall called the JVMCSP). At present, we also have an almost complete C-backend that uses the CCSP [4] runtime and multi-core scheduler, but for this paper we concentrate on the JVM runtime.

In order to utilize the JVM as an executing platform, certain issues must be addressed. Firstly, a typical JVM cannot support millions of threads (i.e., instances of the Java `Thread` class), so a different abstraction of processes must be sought. Other experiments [5] with JCSP [6,7,8] determined that the limit of JCSP threads in one JVM is, at most, in the order of tens of thousands rather than the hundreds of millions that we are interested in.

Secondly, if a system does not rely on operating system threads or processes, then the operating system scheduler is of no use for scheduling and executing such non-OS processes. Consequently, such a system must rely on *cooperative non-preemptive scheduling*. Cooperating scheduling relies on processes being willing to give up the CPU rather than relying on the scheduler interrupting and swapping processes in and out to be run. This means that the

[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`.

code generated must contain *yield* points – in essence return points – as well as integrated code to return execution to the latest yield point when a process is rescheduled. Therefore, we have developed a process abstraction and a code-generation scheme for non-OS processes that generates Java source code (which can then be compiled with a regular Java compiler and instrumented using the ASM tool [9,10] to handle the yielding and correct resumption) that work with a simple single-core cooperative scheduler. In addition, runtime representations of channels, barriers, timer, and all the other needed process-orientation primitives have been implemented.

Figure 1 depicts the ProcessJ system. The right-hand side column represents the runtime with the scheduler and the runtime classes for the process-oriented components. These are compiled and a runtime jar file is created (RT.jar). This runtime jar is produced when the compiler is installed. The middle column represents the ProcessJ compiler. The ProcessJ compiler reads `.pj` files and produces `.java` files which are compiled with the Java compiler to produce class files. After compilation, the class files are instrumented by another Java program that uses the ASM tool-classes to produce code that can work correctly with a cooperative scheduler. These rewritten classes are then packaged with the runtime jar file to form an executable jar. This executable jar can be run directly on the JVM (as long as the necessary libraries are installed and located in the correct locations). The compiler also has an option for creating a completely self contained jar file with the needed library files packaged included. The rest of the paper is organized as follows: in Section 1 we present the background for the work we have done in this paper; in Section 2 we illustrate the implementation details of the ProcessJ code generator and runtime components. In Section 3 we present a number of results, Section 4 concludes, and Section 5 briefly touches upon future work and possible extensions.

1. Background

The ground word for this project was laid in [11] in which we explored the idea of cooperative scheduling of user-level processes on the JVM. We illustrated how to generate code for the JVM (by ways of producing Java source that after compilation is instrumented and rewritten) that uses processes that correctly cooperate in scheduling without using the `Thread` or `Runnable` Java classes. We investigated the implementation needs for such a system, and obtained some promising results by running a number of experiments based on hand-coded samples. At the time we did not have a code generator for the ProcessJ compiler and we had not considered all the consequences of what this might mean for the initial approach. As we shall see, the basic framework that we developed in [11] is sound, but a number of changes were needed. This work ([11]) was originally inspired by [12,13] which concerns mobile processes on the JVM.

1.1. A Simple Non-Preemptive Scheduler

In [11] we described a single-core scheduler depicted in Figure 2. This scheduler is extremely simple, but it performs the basics required by a cooperative scheduler, so the runtime components developed must work correctly with it.

In the following subsection we recap the essential questions posed in [11] for correctly developing a process abstraction that works with a such a scheduler.

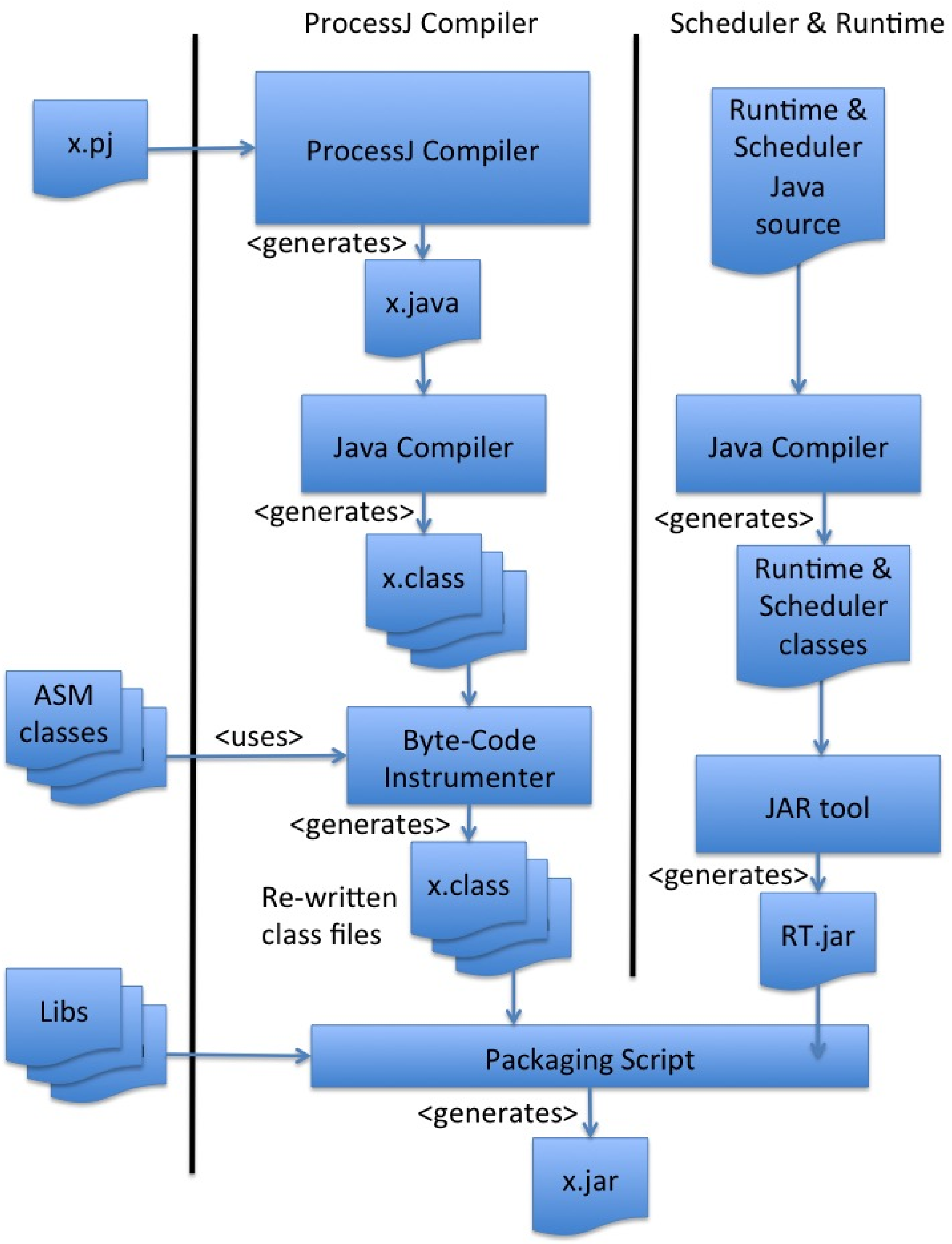

Figure 1. The processJ project overview.

1.2. Essential Questions

In [11] we described a number of requirements to a runtime system in order to use such a scheduler, and these were:

- *How does a procedure yield?*
- *When does a procedure yield and who decides that it does?*
- *How is a procedure restarted/resumed after having yielded?*
- *How is local state (parameters and local variables) maintained?*
- *How are nested procedure calls handled when the innermost procedure yields?*

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

Figure 2. A simplified pseudo-code for a simple non-pre-emptive scheduler.

Let us consider all five of these requirements and discuss the answers originally given in [11] and compare them to the actual implementation in the current version of the ProcessJ compiler.

• *How does a procedure yield?*
The answer to this question remains the same. Namely, a process must voluntarily give up the CPU and return control to the scheduler (remember, it is the scheduler's OS-thread that runs all the processes' code). Therefore, yielding is just a *return* statement (or in reality, a *goto* and a *return* at the end of the code).

• *When does a procedure yield and who decides that it does?*
Again, the answer to this question remains unchanged. It is still the process' own responsibility to determine when it should yield. We have stuck with the idea that a process yields at synchronization points like channel communications, barrier synchronizations, alts, timer timeouts, and par blocks.

• *How is a procedure restarted after having yielded?*
Once again, the original idea has survived. Since a process yielded by calling *return*, it is restarted by simply invoking it again. Naturally, things are not quite that simple. The two major obstacles that must be overcome are:

1. How do we ensure the survival of local state between invocations? This we will discuss in the next paragraph.
2. How do we avoid a procedure starting at the beginning after each invocation?

The second part is done by generating an empty switch statement at the beginning of each procedure and then later instrumenting it with appropriate jump instructions in the compiled bytecode.

• *How is local state (parameters and local variables) maintained?*
Here we see the first major diversion from the original approach. In [11] we associated every procedure invocation with an *activation record* that would be created and stored before the procedure yielded (by simply placing the values of all the local variables in scope into an *Object* array and storing this array on a simulated activation record stack. One downside to this approach is that when a procedure is invoked by the scheduler, all the local variables must be re-established from this stored activation record. It added a lot of complexity to the generated Java code, and correctly maintaining the activation record stack for each process (one activation record was needed for each subsequent procedure call as well) was a complicated

task. A simpler approach is to remove all local variables from the generated code and replace them with fields. After all, a process (a procedure running concurrently with other processes) is represented by an object that represents the process. The runtime system has a generic *Process* class that each procedure extends, and upon generating the code we simply remove all locals and make them fields. At first, this idea was discarded because of the perception that the cost of accessing fields would be prohibitive compared to accessing locals. However, after running some tests, we concluded that accessing fields incurred an overhead of between 1% and 2.5%, and in doing so, we also saved the overhead of creating the activation record objects, storing and manipulating them, as well as restoring the local variables when a procedure is invoked again. In addition, this choice made code generation much simple compared to the approach originally suggested in [11]. Parameters are handled in a similar manner; rather than passing them directly to the procedure, which has been translated into the *run()* method of a class extending the general *Process* class, they are also converted to fields and set through a constructor call on the extending class.
A ProcessJ file called `x.pj` that looks like this:

```
proc void f(...) { ... }
proc void g(...) { ... }
```

is compiled into a Java filed `x.java`:

```
public class x {
  public static class f extends PJProcess {
    // locals and paramters of f(...)
    public f(...) { ... }
    public void run() { ... }
  }
  public static class g extends PJProcess {
    // locals and paramters of g(...)
    public g(...) { ... }
    public void run() { ... }
  }
}
```

• *How are nested procedure calls handled when the innermost procedure yields?*
The original approach in [11] suggested that the caller handle invocations of other procedures (note, we are not talking about a parallel block here, but rather simple sequential procedure invocations). This involves creating and maintaining activation records for the called procedure as well, but in addition, a yield in a called procedure would have to propagate all the way back through the caller and the caller's caller etc. and similarly a re-invocation would have to find its way back to the procedure that yielded. All in all, a lot of bookkeeping was necessary. By sacrificing a little speed for nicer auto-generated code we handle a call to a procedure that may yield as executing the procedure call as a concurrent process in a par block. This generates a new process for the callee, and the caller would be marked *not ready* to run until the callee has finished, in which case it sets the caller back to *ready* and the scheduler can schedule the caller again. Therefore, a call like

```
f(a,b,c);
```

becomes

```
  par {
    f(a,b,c);
  }
```

This will work without any other considerations as the implementation of a par block has an implicit barrier at the end; that is, the process with the procedure call will remain in a non-ready state until all the procedure calls in the par block have terminated (in this case, there is only ever one) – the last process to terminate sets the original process ready to run again.

2. Implementation

2.1. Runtime System Overview

In this section we will briefly consider the overall runtime system design, and then, in the next section, look at the individual components and how they are used in the generated code. Figure 3 illustrates the general runtime system. The core of the runtime is the very simple

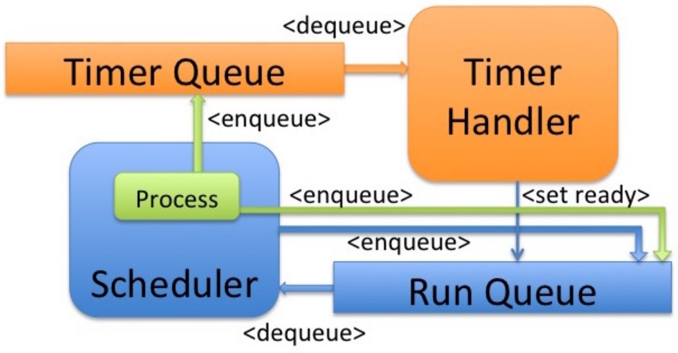

Figure 3. Runtime system overview.

single-core cooperative scheduler described in Section 1.1. This scheduler interacts with a simple queue structure in which processes to run are dequeued from the head and processes that are finished running (because they yielded) are added to the tail of the queue.

When the scheduler runs a process, this process can add new processes to the tail of the run queue – this typically happens when a par-block or a par-for is encountered. The scheduler runs in its own Java thread. However, there is a second Java thread that runs (independently) at the same time as the scheduler, namely, a timer handler. The timer handler is an independent thread that continuously attempts to remove expired timer objects from a delay queue. A delay queue is a priority queue from the `java.concurrent` library on which a call to its *dequeue()* method is blocking. Once the *dequeue()* method returns, it does so with an expired timer object. This timer object knows to which process it belongs, and the timer handler can now set the corresponding process ready to run (meaning that it will be executed by the scheduler when it gets to the head of the queue). There are a few technicalities with canceled timers of alt statements, but we shall not dwell on that at this point. Processes create new timers as well as new processes, and such timers are inserted into the timer handler's queue when needed.

As a side remark, it is worth noting, that were we to implement this entire runtime system on top of a runtime that did not support some sort of threading, things get more complicated: it would then be the scheduler's job to handle the timer queue, which means that the coarseness of the timers in the queue will vary based on the time slices taken up by the processes run by the scheduler. That is, if the scheduler is executing a long running process and a timer has expired, then the process will not know so without performing the checking itself. This may be a problem in the future when targeting runtimes such as the Java Script backend.

2.2. Java Runtime Components

2.2.1. State Retention

As already discussed, all locals and parameters are transformed into fields in the generated Java code. A field representing a local variable will have a name of the form `_ldXname`, where X is an integer number (no integer is reused, and they increase by one for every new local), and `name` represents the actual name of the local. Formal parameters are transformed in much the same way but with the prefix `_ld` replaced by `_pd`. In the following sections we have chosen to replace the `_ld` and `_pd` by the original variable names from the ProcessJ code for clarity (except for the one example in the following section).

2.2.2. Processes

Any ProcessJ procedure that has calls to the `yield()` method is transformed into a class. Consider a generic ProcessJ procedure like this:

```
proc void foo(pt1 pn1, pt2 pn2, ..., tpn pnn) {
  ...
  lt1 ln1;
  ...
  ltm lnm;
  ... statements ...
}
```

which by the code generator is transformed into the following Java class (assuming the ProcessJ file in which the procedure came from is called A.pj):

```
public class A {
  public static class foo extends PJProcess {
    pt1 pn1;
    pt2 pn2;
    ...
    lt1 ln1;
    ...
    ltm lnm;

    public foo(pt1 pn1, pt2 pn2, ..., tpn pnn) {
      this.pn1 = pn1;
      ...
      this.pnn = pnn;
    }

    public void run() {
       switch (runlabel) {
         case 0: resume(0); break;
         case 1: resume(1); break;
         ...
```

```
          case k: resume(k); break;
        }
        ... statements' ...
      }
    }
  }
```

where `... statements' ...` contains placeholders of the form `label(i)` representing the address that the `resume(j)` calls in the switch statement at the beginning of the `run()` method represents jumps to. For example, consider the following small procedure:

```
proc void f(chan<int>.read in, int count) {
  int sum = 0;
  while (count > 0) {
    int val;
    val = in.read();
    sum = sum + val;
    count--;
  }
  println(sum);
}
```

which the ProcessJ compiler translates to (here, for illustrative purposes we have retained the generated field names):

```
import java.util.*;
import ProcessJ.runtime.*;

public class Main {
  public static class f extends PJProcess {
    // local variables and parameters
    PJChannel<Integer> _pd$in;      // parameter 'in' of 'f'
    int _pd$count;                 // parameter 'count' of 'f'
    int _ld0$sum;                        // local 'sum' of 'f'
    int _ld1$val;                        // local 'val' of 'f'

      // constructor to set initial value of parameters
    public f(PJChannel<Integer> _pd$in, int _pd$count) {
      this._pd$count = _pd$count;
      this._pd$in = _pd$in;
    }

    public synchronized void run() {
      switch(this.runLabel) {
        case 0: break;
        case 1: resume(1); break;
        case 2: resume(2); break;
      }
      _ld0$sum = 0;
      while( (_pd$count > 0) ) {
        label(1);
        if(_pd$in.isReadyToRead(this)) {
          _ld1$val = _pd$in.read(this);
          yield(2);
        } else {
          setNotReady();
          _pd$in.addReader(this);
          yield(1);
        }
        label(2);
        _ld0$sum = (_ld0$sum + _ld1$val);
        _pd$count--;
      };
      std.io.println( _ld0$sum );
      terminate();
    }
  }
}
```

When compiled with the Java compiler, the start of the Java byte code looks something like this:

```
 0: aload 0
 1: getfield runLabel I
 4: tableswitch  // 0 to 2
    0:       32
    1:       35
    2:       43
    default: 48
32: goto 48

35: aload 0
36: iconst 1
37: invokevirtual resume/(I)V
40: goto 48

43: aload 0
44: iconst 2
45: invokevirtual resume/(I)V

48: aload_0
49: iconst_0
50: putfield _ld0$sum I
53: aload_0
54: getfield _pd$count I
57: ifle 158
60: aload_0
61: iconst_1
62: invokevirtual label/(I)V
65: aload_0
...
```

The switch statement has been translated into a `tableswitch` instruction and the calls to `resume` follow at addresses 35 and 43. At address 48 we see the `label(1)` invocation from the generated code. We then use the ASM byte code rewriting tool to locate the addresses of all the `label()` invocations and replace them by `nop` instructions, and then we replace the `resume()` invocation by goto instructions that transfer control to the appropriate address determined by the location of the `label()` invocations. For the byte code above, we get

```
 0: aload 0
 1: getfield runLabel I
 4: tableswitch  // 0 to 2
    0:       32
    1:       35
    2:       43
    default: 48
32: goto 48

35: nop          // was aload 0
36: nop          // was iconst 1
37: goto 62      // was invokevirtual resume/(I)V
40: goto 48

43: nop          // was aload 0
44: nop          // was iconst 2
45: goto 129     // was invokevirtual resume/(I)V

48: aload_0
49: iconst_0
50: putfield _ld0$sum I
53: aload_0
54: getfield _pd$count I
57: ifle 158
60: nop          // was aload_0
61: nop          // was iconst_1
62: nop          // was invokevirtual label/(I)V
63: nop          // was invokevirtual label/(I)V
64: nop          // was invokevirtual label/(I)V
...
65: aload_0
```

where `129` in address 45's goto is the address of the `label(2)` invocation. It looks like the code is extremely inefficient with all the extra jumps and `nop` operations, but in reality, if we removed the extra superfluous instruction, the Java runtime verifier will complain. This is due to something called the Java stack frame map. It contains information about the types of the parameters and locals stored in the activation record on a line-by-line basis. Unfortunately, ASM does not do a good enough job re-calculating these frame maps when we make changes to the code; this is obviously something that we need to look into in order to optimize the generated code. Also note, a goto operation takes three 16-bit words, therefore, the addresses 37-39, 40-42, 45-47 represent the goto instructions. Similarly, because an *invokevirtual* takes up three words the `nop` instructions in lines 63 and 64 were added to ensure the correct instruction alignment. It should be noted that the `label()` and `resume()` methods could have been declared static and that would have saved the `aload_0` instruction – we shall investigate this further in the next version of the compiler.

In the following subsections, when we show generated Java code, we have retained the names of the locals used in ProcessJ for readability reasons, but in reality, these names would have been replaced by auto-generated new names for the equivalent fields.

2.2.3. *Par Blocks*

In ProcessJ a parallel block is simply a block of curly-brackets with the keyword `par` prefixed:

```
par {
  f(8);
  g(9);
}
```

is compiled into the following code:

```
final PJPar par1 = new PJPar(2, this);

(new A.f(8){
  public void finalize() {
        par1.decrement();
      }
}).schedule();

(new A.g(9){
  public void finalize() {
    par1.decrement();
  }
}).schedule();

setNotReady();
yield(1);
label(1);
```

The `PJPar` runtime element is initialized with the number of processes in the par block, namely 2, and the `finalize()` methods of these will decrement a counter in the `PJPar` object and once the count reaches 0, the process containing the par block will be *ready* to run again. If the processes in the par block are procedure calls like `f(8)` then the `f` class can be extended to contain a `finalize()` method that correctly decrements the number of processes that belong to the par block. It can then immediately be instantiated and scheduled (by calling the `schedule()` method on it). We assume that both `f` and `g` are processes declared in a file called `A.pj`.

For non-invocations the `PJProcessJ` class is extended anonymously and code is generated for the non-invocation in the `run()` method, for example a statement like

```
x = in.read();
```

in a par-block will generate the following code:

```
new PJProcess(){
  public synchronized void run() {
    switch(this.runLabel) {
      case 0: break;
      case 2: resume(2); break;
      case 3: resume(3); break;
    }
    label(2);
    if(in.isReadyToRead(this)) {
      x = in.read(this);
      yield(3);
    } else {
      setNotReady();
      in.addReader(this);
      yield(2);
    }
    label(3);
    terminate();
  }

  public void finalize() {
    par1.decrement();
  }
 }.schedule();
```

2.2.4. *Channels*

ProcessJ supports 4 different kinds of channels: one-to-one, one-to-many, many-to-one, and many-to-many. A general Java class `PJChannel` serves as the superclass for the classes `PJOne2OneChannel`, `PJOne2ManyChannel`, `PJMany2OneChannel`, and `PJMany2ManyChannel` that represent the 4 different kinds of channels. The functionality for reading and writing is defined in the `PJChannel` class which is a parameterized class, where the parameter is the equivalent Java type carried by the channel in ProcessJ. A typical read operation on a read channel end in ProcessJ may look like this:

```
v = a.read();
```

and the code generated becomes:

```
tmp1 = c;                          // temp to avoid side effects
label(1);                           // return here if read fails
if (tmp1.isReadyToRead(this)) {   // check if there is data
  v = tmp1.read(this);                        // the actual read
  yield(2);                       // yield and return to label(2)
} else {
  setNotReady();                  // set process not ready to run
  tmp1.addReader(this);          // add the reader to the channel
  yield(1);                       // yield and return to label(1)
}
```

```
label(2);                         // return here if read succeeds
...
```

yield(2) and label(2) are added simply for fairness to other processes – they can be removed without any consequences. Similarly, a write in ProcessJ looks like this:

```
b.write(v);
```

which becomes the following code in Java:

```
tmp2 = b;                           // temp to avoid side effects
label(3);                           // return here if write fails
if (tmp2.isReadyToWrite()) {   // check if channel is empty
  tmp2.write(this, v);                       // the actual write
  yield(4);                        // yield and return to label 4
} else {
  setNotReady();                  // set process not ready to run
  yield(3);                        // yield and return to label 3
}
label(4)                          // return here if write succeeds
```

Again, yield(4) and label(4) are just for fairness. If the channel-end expression on which the read is called is more complex than a simple name, a temporary variable is introduced in order to avoid issues that may arise with possible side effects.

2.2.5. Claim

Shared channel ends must be *claim*ed before they can be read from or written to. A *claim* of shared channel ends looks like this in ProcessJ:

```
claim (a, b) {
  ... statements ...
}
```

(the channel ends a and b could have been more complex expressions of channel-end type or abbreviations). The generated Java code looks like this:

```
tmp1 = a;                          // temp to avoid side effects
tmp2 = b;                          // temp to avoid side effects
label(1);                          // return here if claim fails
if (!(tmp1.claim() && tmp2.claim())) {     // try to claim
  tmp1.unclaim();       // unclaim the shared channel ends
  tmp2.unclaim();
  yield(1);                       // yield and return to label 1
}
... statements ...                         // original statements
tmp1.unclaim();         // unclaim the shared channel ends
tmp2.unclaim();
```

It should be noted that before the yield() call, the process remains *ready* to run (this differs from the normal approach of setting processes *not-ready* when having executed a synchronizing event that did not conclude. That means that a claim that fails and goes back at the end of the run queue is ready to run immediately. This is not the most efficient implementation, but it is a fairly simple approach that works for now. This is on the list of things that need to be addressed in the next iteration of the compiler.

2.2.6. *Timers and The Timer Queue*

A timer can be used to delay execution by calling `timeout()` on it, but also as a guard in an alt. For a timeout call outside an alt:

```
t.timeout(100);
```

simply translates to

```
t.start(100);
setNotReady();
yield(1);
label(1);
```

where `t` in the generated Java code is an instance of the `PJTimer` class, and the call to `start()` inserts the timer into the timer queue - a separate thread that we described in Section 2.1.

2.2.7. *Alts*

We have implemented the alt to always act like a prioritized alt. That is, the lexicographically first ready guard is always chosen. Let us consider how to generate code for the ProcessJ alt statement.

1. Instantiate a `PJAlt` class.
2. Declare timers used for timeouts as guards and temporaries for channel read guards to avoid side effect issues.
3. Create an array with all the guards.
4. Evaluate all pre-guards and create a corresponding Boolean array (guards without a preguard gets a true entry); this is only ever done once per execution of the alt.
5. The method `setGuards()` is called on the alt object and passed the arrays of pre-guards and guards. This method return false if all Boolean guards are false. If this is the case, a runtime exception is raised.
6. The method `getReadyGuardIndex()` is invoked on the alt object. If no guards are ready, -1 is returned; otherwise, the index of the ready guard is returned. When a shared channel end is ready and chosen, it is reserved for the alt (this acts like an implicit claim of the channel end) and automatically unreserved when read.
7. This index is used as a case in a switch-statement to execute the code associated with the chosen guard.
8. If no guard is ready, timers are started (if this is the first time the guards are evaluated, only, and inserted into the timer queue) and the process yields. It resumes to recall the `getReadyGuardIndex()` method on the alt object.
9. If a guard was ready and its associated code was executed, all timers started for the alt are killed and the process yields; once resumed it continues where it left off. This last yield is not technically necessary, but we added it for fairness.

Let us consider the following ProcessJ alt with one of each type of legal guard:

```
timer t;
int v;
alt {
  (a >0) & v = in.read() : {
    // statements
  }
  t.timeout(100) : {
    // statements
  }
  skip : {
    // statements
  }
}
```

```
PJAlt alt = new PJAlt(3, this);          // create alt object
                          // initialize timers for timeouts
t = new PJTimer(this, 100);
tmp1 = in;
                                             // guard array
Object[] guards = {tmp1, t, PJAlt.SKIP_GUARD};
tmp0 = (a > 0);                                // pre-guard
                                         // pre-guard array
boolean[] boolGuards = {tmp0, true, true};
                               // set guards and pre-guards
boolean bRet = alt.setGuards(boolGuards, guards);
if (!bRet) {           // check if all pre-guards are false.
  System.out.println("RuntimeException: One of the Boolean
                      pre-guards must be true.");
  System.exit(1);
}

label(2);              // return here if no guards are ready

chosen = alt.getReadyGuardIndex();      // pick ready guard
switch(chosen) {
  case 0:                                     // read guard
    v = tmp1.read(this);                     // do the read
    ... statements ...
    break;
  case 1:                                  // timeout guard
    ... statements ...
    break;
  case 2:                                     // skip guard
    ... statements ...
    break;
  case -1:                               // no ready guards
                // start timers if this is the first time
    if (!t.started) {
      t.start();
    }
    break;
}

if (chosen == -1) {          // yield if no guard was ready
  yield(2);
} else {
  if (t.started && !t.expired) {             // kill timers
    t.kill();
  }
  yield(3);                              // yield for fairness
}
label(3);                    // return here if the alt succeeds
```

As with claims, alts that yield do so without being set *not-ready*. Again, this also need changing, but does not effect the correctness of the code. It should be noted that pre-guards are only evaluated once every time an alt is executed. If the alt suspends, upon resumption, the pre-guards are not re-evaluated.

2.2.8. Barriers

A barrier synchronization in ProcessJ looks like this:

```
  sync (b);
```

simply translates to a method invocation like this:

```
  b.sync(this);   // decrement counter, equeue, set not ready
  yield(1);                     // yield and return to label 1
  label(1);      // return here when everyone has sync'd on b
```

When the sync method is invoked, the counter kept in the `PJBarrier` object is decremented and the process is added to a queue such that it can be set ready to run when every process enrolled on the barrier has called `sync()`. The call to `sync()` also sets the process not ready to run.

To correctly handle terminating processes that are enrolled on barriers, the `finalize()` method of the process withdraws it from the barrier (this is consistent with the semantics laid out in [14]. The `finalize()` method of a process is called by the scheduler before a process is terminated and it is by barriers and par blocks for controlling when to set ready the processes owning them.

3. Results

We have implemented a number of tests not only to determine the correctness of the code generation and the runtime components, but also to determine the performance of the system. Because of the way we represent processes as objects, there is bound to be some overhead associated with object creation – every time a process is created an object is instantiated. This is a cost that we cannot change and we return to consider it in the next subsection.

3.1. Time & Context-Switches (Mandelbrot Picture)

We implemented a simple Mandelbrot set computation and generated a picture of size 4,000×3,000 (12,000,000 pixels). We tested four different versions of this program:

- A sequential Java program.
- A sequential ProcessJ program.
- A concurrent ProcessJ program with the outer loop (3,000 rows) parallelized.
- A concurrent ProcessJ program with both loops (12,000,000 pixels) parallelized.

It should be noted, that the only difference between the sequential Java and ProcessJ versions is the `import` statement and the addition of the word `proc` in front of the Java methods, and the difference between the sequential and concurrent ProcessJ version is the addition of the keyword `par` in front of the for-loop(s). Table 1 shows the runtime results in seconds, the total number of processes created, and the total number of context switches. The time reported is the average time of 10 executions measured using the `time` shell command. The number of context switches was determined by instrumenting the scheduler to keep count. As Table 1

Table 1. Mandelbrot picture results.

Version	Time (Sec.)	#Processes	Context Switches
Java sequential	6.24	1	0
ProcessJ sequential	6.21	1	0
ProcessJ row parallelized	6.05	3,001	3,001
ProcessJ pixel parallelized	31.98	12,000,001	12,003,001

shows, the runtimes of the sequential Java and sequential ProcessJ versions are comparable. The row-parallelized ProcessJ version show a similar runtime, which is to be expected as the current runtime utilizes a single core scheduler only. The important thing to note here is, that the overhead of creating processes and performing context switches is not prohibitive; however, the execution time for the per-pixel parallelized version is very high compared to the other versions. This is because of the 4,000 times more process creations. The process creation time is equivalent to the Java time it takes to allocated an object on the heap. The lesson learned from this last version is, that when parallelizing an implementation, the level of granularity must be considered: The amount of work per pixel is too small compared to the overhead of creating a new process. Once the object-creation time (25.57 seconds) has been subtracted from the 31.98 seconds of the pixel parallelized version, the runtime was 6.41, which is comparable to the other three versions. The overhead of process creation by object instantiation is in the order of 2.15 μ-seconds per process object.

3.2. Context-Switching Time (CommsTime)

In [11] we implemented CommsTime on two different architectures (a Mac OS X machine and an AMD server). We ran tests with the prototype runtime and scheduler presented in [11] and JCSP [7] and we have reproduced these numbers in the first two columns of Table 2, referred to as *LiteProc*; the third column reports the numbers of the same CommsTime program compiled with the ProcessJ compiler using the new code generator and runtime. The test architectures we used in this paper for the CommsTime are the same as we used in [11]:

- Mac Pro 4.1, OS X Snow Leopard, Intel i7 Quad-core Xenon 2.93 MHz with 8GB RAM.
- 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).

Table 2. CommsTime results.

	Mac / OS X			AMD / Linux		
	LiteProc	JCSP	JVMCSP	LiteProc	JCSP	JVMCSP
μs / iteration	9.26	27.00	**8.30**	13.56	136.00	**7.52**
μs / communication	2.31	6.00	**2.08**	3.90	35.00	**1.88**
μs / context switch	1.32	3.00	**0.69**	1.94	17.00	**0.63**

For the Mac / OS X machine we saw an improvement in iteration and communication time of around 10% and an improvement of context switching of almost 50%. The improvement in context switching time is most likely because of the removal of the code associated with creating activation record objects and storing into and retrieving from them the values of the parameters and local variables.

3.3. *Max Process Count*

In order to determine the max number of simple processes the ProcessJ runtime can support, we implemented a simple par-for loop that starts two processes that are connected by two channels, one in each direction. These processes exchange two values, one in each direction. The code looks like this:

```
import std.strings;

proc void foo(chan<int>.read c1r, chan<int>.write c2w) {
  int x;
  par {
    x = c1r.read();
    c2w.write(10);
  }
}

proc void bar(chan<int>.write c1w, chan<int>.read c2r) {
  int y;
  par {
    y = c2r.read();
    c1w.write(20);
  }
}

proc void main(string[] args) {
  par for (int i=0; i<string2int(args[1]); i++) {
    chan<int> c1, c2;
    par {
      foo(c1.read, c2.write);
      bar(c1.write, c2.read);
    }
  }
}
```

We ran this test on an Intel Xeon CPU (32-core) E5-2630 v3 @ 2.40GHz with 128GB RAM running GNU/Linux (3.10.0-327.4.5.el7.x86 64). Table 3 shows a number of different run

Table 3. Max number of processes.

# of Processes	# of Context Switches	Execution Time (secs)	Memory Used (GB)
7,000,001	15,000,002	7.53	1.79
10,500,001	22,500,002	16.03	3.02
14,000,001	30,000,002	25.86	4.10
210,000,001	450,000,002	642.80	63.91
350,000,001	750,000,002	1235.12	94.50
420,000,001	900,000,002	1443.40	125.82
476,000,001	1,020,000,002	1800.79	126.11
480,900,001	1,030,500,002	1801.40	126.20

times and space requirements for the test program. The largest number we achieved was 480,900,001 – just 4% away from half a billion processes in the run queue at the same time. It is worth noting that not only did we generate 480,900,001 processes (each being one object in memory), but also 961,800,000 channel objects, so the heap at some time contained a total of 1,442,700,000 runtime objects. The total number of context switches for this run was well over one billion. Figure 4 graphically depicts the data in Table 3 – we have scaled the x-axis

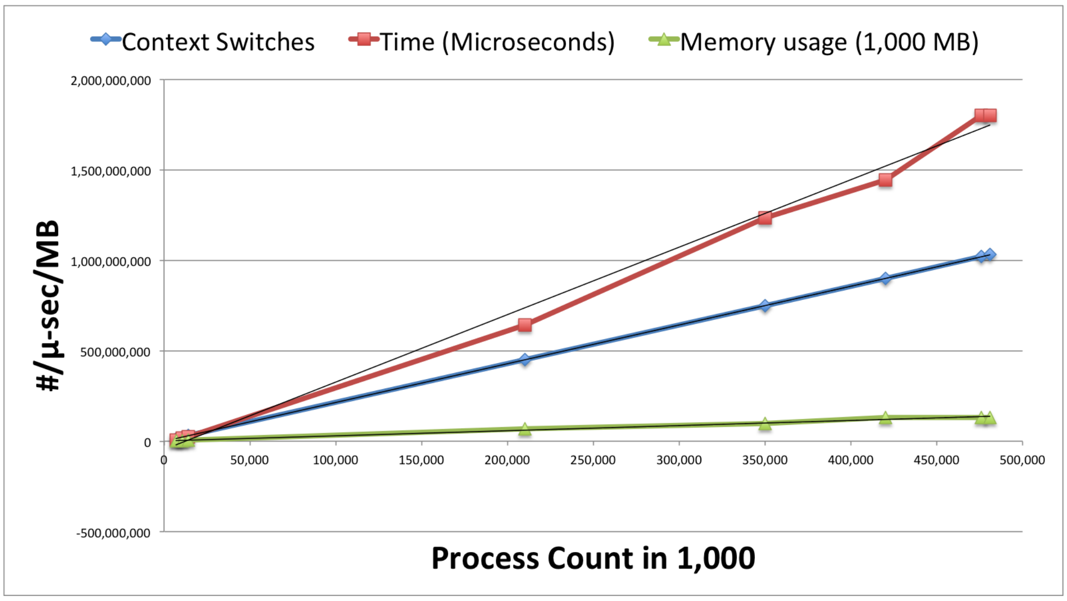

Figure 4. Results.

to represent the number of processes in thousands, and we have scaled the time to μ-seconds and memory usage is given in MB. As we can see in Figure 4, the actual results fit snugly with the trend-lines (which is obvious for the number of context switches, and possibly also for the memory usage) more importantly the measured speed fits the trend-line which leads us to believe that the program is memory bound; that is, with more memory we can most likely run more processes.

3.4. Overhead

Table 4 shows the sizes of the runtime objects. These sizes were found by serializing instances and determining the size of the corresponding byte array. The runtime elements marked with a * are given in *base sizes*, that is, they grow in size depending on the number of elements

Table 4. Sizes of Runtime Elements

Runtime Class	Size
PJProcess*	68 bytes
PJAlt*	223 bytes
PJBarrier*	127 bytes
PJChannel	143 bytes
PJOne2OneChannel*	204 bytes
PJOne2ManyChannel*	269 bytes
PJMany2OneChannel*	269 bytes
PJMany2ManyChannel*	283 bytes
PJPar	133 bytes
PJTimer (without a process)	119 bytes
PJTimer (with a process)	182 bytes

associated with the element; for example, a `PJProcess` grows linearly in terms of the number of local and parameters the ProcessJ code has. A `PJChannel` serves as the super class for all other channel types and is never instantiated separately.

4. Conclusion

In this paper we presented the implementation of the JVMCSP code generator and runtime system for ProcessJ. This is an improved realization of the approach suggested in [11] and is part of the M.Sc. work in [15]. We have shown that we can execute a very large number of processes – on the order of 3.5-4.0 million depending on the number of local variables, channels etc. per gigabyte of main memory. We have shown that context switching is not prohibitively expensive for a cooperatively scheduled runtime system.

5. Future Work

A number of important improvements can be made to the system. Notably, the first major improvement is a multi-core scheduler. In this section we briefly mention some of the more important improvements.

5.1. Multi-Core Scheduler

A multi-core scheduler for the ProcessJ runtime is being developed. We have already designed the runtime elements with proper synchronized accesses, though it was not required for the single-core scheduler, so that it will not require too many modifications to work with a multi-core scheduler.

5.2. Libraries

Libraries are an important part of any language. We already have some basic libraries for standard in/out, random number generation and a string library. But we intend to develop more sophisticated libraries for graphics and data structures.

5.3. Blocking I/O Calls

Blocking I/O calls are going to be handled by spawning new individual threads to run in, so we would need to have the `PJProcess` class implement the runnable interface eventually. The drawback to this is that it will add to the size (in bytes) of each process.

5.4. Mobile Processes and Polymorphic Resumption

In order to implement mobile processes with polymorphic resumption interfaces [13], the only changes needed to the code generator is as follows:

- A number of extra parameter-setter methods to support the polymorphic nature of the mobiles must be added to the class generated for the mobile process.
- When invoking a mobile process, a par-block wrapper is needed.
- Support accessors for determining the available procedure interface must be implemented.

In addition to change to the code generator, the implementation of the name resolution phase for mobile processes with polymorphic resumption interfaces must be changed to conform with the rules defined in [13].

5.5. Implementation Improvement

5.5.1. Alternative (alts)

Alts are currently implemented as a busy-wait where they yield with a status ready to run and keep getting rescheduled until one of the guards is ready. We would like to improve them by yielding with a not ready status and have the ready guard wake up the alting process. This is not as simple as it sounds since it involves a fair amount of bookkeeping in various runtime elements. We also do not allow barriers in alt blocks and nested alt blocks for now, since this requires a few design improvements as well.

5.5.2. Claims

Just like alt blocks, this is also implemented as a busy-wait mechanism for now and we want to change that as well.

5.5.3. Run Queue

The current single core scheduler has a single run queue which holds both ready to run and not ready to run processes. A good improvement on this would be to have two different run queues; one for the ready processes and the other for not-ready processes. This will decrease the overhead of iterating through non-ready processes that the current scheduler does. But this also requires some bookkeeping and process-waking-up mechanisms in many runtime elements. This improvement will be done as a part of the multi-core scheduler.

5.6. New Backends and Runtimes

A number of interesting developments in possible new backends and runtimes have happened recently. In [16] a JavaScript runtime is described; this runtime is a perfect target for ProcessJ and something we will be looking at. A new C++ runtime system, C++CSP [17], is something we will seriously consider as well, even though there is a much smaller limit to the number of possible processes per core. However, the threading has been abstracted away, which mean that the runtime can be placed on almost any thread or process abstraction.

Acknowledgments

Thanks to Patrick Daleiden for proof reading this paper for us.

References

[1] Jan B. Pedersen and Marc L. Smith. ProcessJ: A Possible Future of Process-Oriented Design. In *"Communicating Process Architectures 2013"*, number WoTUG-35 in Concurrent System Engineering Series. Open Channel Publishing, 2013.

[2] Charles A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification – Java SE 7 Edition, March 2013. `https://docs.oracle.com/javase/specs/jls/se7/html/`.

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

[5] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. *Concurrency and Computation: Practice and Experience*, 22:965–980, March 2010.

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] Eric Bruneton. ASM 4.0 - A Java bytecode engineering library, 2011.

[11] Jan B. Pedersen and Andreas Stefik. Towards Millions of Processes on the JVM. In Peter H. Welch et al., editor, *"Communicating Process Architectures 2014"*, volume 71 of *Concurrent System Engineering Series*. Open Channel Publishing, August 2014.

[12] 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.

[13] Jan B. Pedersen and Matthew Sowders. Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces. In *The thirty-third Communicating Process Architectures Conference, CPA 2011, organised under the auspices of WoTUG, Limerick, Ireland, June 19-22 2011*, pages 71–85, 2011.

[14] Frederick R. M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier synchronisations for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of PDPTA 2005*. CSREA press, June 2005.

[15] Cabel Shrestha. The JVMCSP Runtime and Code Generator for ProcessJ in Java . Master's thesis, University of Nevada Las Vegas, May 2016.

[16] Kurt Micallef and Kevin Vella. Communicating Generators in JavaScript. In Kevin Chalmers, Jan B. Pedersen, Brian Vinter, Kenneth Skovhede, and Peter H. Welch, editors, *Proceedings of Communicating Process Architectures (CPA) 2016*, volume 73, August 2016.

[17] Kevin Chalmers. A Modern C++CSP Library. In Kevin Chalmers, Jan B. Pedersen, Brian Vinter, Kenneth Skovhede, and Peter H. Welch, editors, *Proceedings of Communicating Process Architectures (CPA) 2016*, volume 73, August 2016.