Resumable Java Bytecode – Process Mobility for the JVM

Jan Bækgaard PEDERSEN and Brian KAUKE

School of Computer Science, University of Nevada, Las Vegas, Nevada, United States.

`matt@cs.unlv.edu`, `kaukeb@gmail.com`

Abstract. This paper describes an implementation of resumable and mobile processes for a new process-oriented language called ProcessJ. ProcessJ is based on CSP and the π-calculus; it is structurally very close to occam-π, but the syntax is much closer to the imperative part of Java (with new constructs added for process orientation). One of the targets of ProcessJ is Java bytecode to be executed on the Java Virtual Machine (JVM), and in this paper we describe how to implement the process mobility features of ProcessJ with respect to the Java Virtual Machine. We show how to add functionality to support resumability (and process mobility) by a combination of code rewriting (adding extra code to the generated Java target code), as well as bytecode rewriting.

Introduction

In this paper we present a technique to achieve process resumability and mobility for ProcessJ processes executed in one or more Java Virtual Machines.

ProcessJ is a new process-oriented language with syntax close to Java and a semantics close to occam-π [20]. In the next subsection we briefly introduce ProcessJ.

We have developed a code generator (from ProcessJ to Java) and a rewriting technique of the Java bytecode (which is the result of compiling the Java code generated by the ProcessJ compiler) to alter the generated Java bytecode to save and restore state as well as support for resuming execution in the middle of a code segment.

This capability we call transparent mobility [16], which differs from non-transparent mobility in that the programmer does not need be concerned about preserving the state of the system at any particular suspend or resume point. We do not, however, mean that processes may be implicitly suspended at arbitrary points in their execution.

ProcessJ

ProcessJ is a new process-oriented language. It is based on CSP [8] and the π-calculus [10]. Structurally it is very much like occam-π; it is imperative with support for synchronous communication through typed channels. Like occam-π, it supports mobility of processes. Syntactically it is very close to Java (but without objects), and with added constructs needed for process orientation. ProcessJ currently targets the following execution platforms through different code generators (it produces source code which is then compiled using a compiler for the target language):

- Any platform that supports the KRoC [21,23] occam-π compiler. ProcessJ is translated to occam-π, and then passed to the KRoC compiler.

- C and MPI [5], making it possible to write process-oriented programs for a distributed memory cluster or wide area network. ProcessJ is translated to C with MPI message passing calls and passed to a C compiler.
- Java with JCSP [17,19], which will run on any architecture supporting a JVM. ProcessJ is translated to JCSP, which is Java with library-provided CSP support, and passed to the Java compiler.

In this paper we focus on the process mobility of ProcessJ for Java/JCSP. As the JVM itself provides no support for continuations and the Java language provides a restricted set of flow control constructs on which to build such functionality, it was initially not clear whether transparent process mobility could be usefully implemented on this platform.

Like any other process-oriented language, ProcessJ has the notion of processes (procedures executing in their own executing context), but since the translation is to Java, it is sometimes necessary to refer to methods when describing the generated Java code and Java bytecode. A simple example of a piece of ProcessJ code (without mobility) is a multiplexer that accepts input on two input channels (`in1` and `in2`) and outputs on an output channel (`out`):

```
proc void mux(chan<int>.read in1,
              chan<int>.read in2,
              chan<int>.write out) {
  int data;
  while (true) {
    alt {
      data = in1.read() :
        out.write(data);
      data = in2.read() :
        out.write(data);
    }
  }
}
```

where `chan<int>.read in1` declares `c1` to be the reading end of a channel that carries integers, `out.write(data)` writes the value of `data` to the `out` channel, and `alt` represents an "alternation" between the two input guards guarding two channel write statements.

Other approaches such as [2,16] consider thread migration (which involves general object migration) in the Java language, but since ProcessJ is not object-oriented, we do not need to be concerned with object migration at the programmer level. We do use an encapsulation object at the translation level from ProcessJ to Java to hold the data that is transferred (this object serves as a continuation for the translated ProcessJ process). In addition, mobile processes, like in occam-π, are started, and resumed, by simply calling it as a regular procedure (which translates into invoking it as a regular non-static method in the resulting Java code). In this way, we can interpret the suspended mobile as a continuation [7] represented by the object which holds the code, the saved state, and information about where to continue the code execution upon resumption.

1. Resumability

We start by defining the term *resumability*. We denote a procedure as *resumable* if it can be temporarily terminated by a programmer-inserted `suspend` statement and control returned to the caller, and at some later time restarted at the instruction immediately following the suspend point and with the exact same local state, possibly in a different JVM located on a different machine (i.e. all local variables contain the same values as they did when the method was terminated). In a process-oriented language, the only option a process has for communicating with its environment, that is, other processes, is though channel communication, which means that when the process is resumed, it might be with a different set of (channel, channel end, or barrier) parameters; in other situations a process might take certain parameters for initialization purposes [22], which must be provided with "dummy values" upon subsequent resumptions. Therefore, in this paper we consider resumability for procedures where the values of the local variables are saved and restored when the process is resumed, but where each process resumption can happen with a new set of actual parameters. This allows the environment to interact with the process. We start with a formal definition of resumability.

1.1. Formal Definition of Resumability

In this section we briefly define *resumability* for JVM bytecode in a more formal way (we disregard objects and fields as neither are used in ProcessJ.)

Each invocation of a bytecode method has its own evaluation stack; recall, the JVM is a stack based architecture, and all arithmetic takes place on the evaluation stack, which we can model as an array s of values:

$$s = [e_0, e_1, \ldots, e_i]$$

In addition to an evaluation stack, each invocation has its own activation record (AR) (we consider non-static methods, but static methods are handled in a similar manner; the only difference is that a reference to *this* is stored at address 0 in the activation record for non-static method invocations). We can also represent a saved activation record as an array:

$$\mathcal{A} = [this, p_1, \ldots, p_n, v_1, \ldots, v_m],$$

where $this$ is a reference to the current object, p_i are parameters, and v_i are local variables. ($p_1 = \mathcal{A}[1], \ldots, p_n = \mathcal{A}[n], v_1 = \mathcal{A}[n+1], \ldots, v_m = \mathcal{A}[n+m]$), where $\mathcal{A}[i]$ denotes the *value* of the parameter/local variable stored at address i. We do not need to store *this* in the saved activation record as it is automatically replaced at address 0 of the activation record for every invocation of a method, but we include it here as there are instructions that refer to address 0, where *this* is always stored for non-static methods. It is worth mentioning that the encapsulating object used in the ProcessJ to Java translation uses non-static methods and fields; this is necessary since a ProcessJ program might have more mobile processes based on the same procedure.

We can now define the semantic meaning of executing a basic block of bytecode instructions by considering the effect it has on the stack and the activation record. Only the last instruction of such a block can be a jump, so we are working with a block of code that will be executed completely. At this point, it is worth mentioning that at the end of a method invocation the stack is always empty; in addition, a ProcessJ `suspend` statement will translate to a `return` instruction, and at these return (suspend) points, the evaluation stack will also be empty. We consider a semantic function $\mathcal{E}_{JVM}[\![B]\!](s, \mathcal{A})$ where $B = i_0\ i_1\ \ldots\ i_k$ is a basic block of bytecode statements and define:

$$\begin{array}{rcl} \mathcal{E}_{JVM}[\![i_0\ i_1\ \ldots\ i_k]\!](s, \mathcal{A}) & = & \mathcal{E}_{JVM}[\![i_1\ \ldots\ i_k]\!](s', \mathcal{A}'), \text{ where} \\ (s', \mathcal{A}') & = & \mathcal{E}_{JVM}[\![i_0]\!](s, \mathcal{A}) \end{array}$$

We shall not give the full semantic specification for the entire instruction set for the Java Virtual Machine as it would take up too much space in this paper, but most of the instructions are straightforward. A few examples are (we assume non-static invocations here):

$$
\begin{array}{rcl}
\mathcal{E}_{JVM}[\![\texttt{iload 1}]\!]([\ldots],\mathcal{A}) & = & ([\ldots,a_1],\mathcal{A}) \\
 & & \text{where } \mathcal{A} = [this, a_1, \ldots, a_{n+m}] \\
\mathcal{E}_{JVM}[\![\texttt{istore 1}]\!]([\ldots,e_{k-1},e_k],\mathcal{A}) & = & ([\ldots,e_{k-1}],[this,e_k,a_2,\ldots,a_{n+m}]) \\
 & & \text{where } \mathcal{A} = [a_0,a_1,\ldots,a_{n+m}] \\
\mathcal{E}_{JVM}[\![\texttt{goto X}]\!]([\ldots],\mathcal{A}) & = & \mathcal{E}_{JVM}[\![B']\!]([\ldots],\mathcal{A}) \\
 & & \text{where } B' \text{ is the basic block} \\
 & & \text{that starts at address } \texttt{X}. \\
\mathcal{E}_{JVM}[\![\texttt{ifeq X}]\!]([e_0,e_1,\ldots,e_k],\mathcal{A}) & = & \mathcal{E}_{JVM}[\![B']\!]([e_0,e_1,\ldots,e_k],\mathcal{A}), \\
 & & \text{if } e_k = 0 \text{ where } B' \text{ is the basic block} \\
 & & \text{that starts at address } \texttt{X}. \\
\mathcal{E}_{JVM}[\![\texttt{ifeq X}]\!]([e_0,e_1,\ldots,e_k],\mathcal{A}) & = & \mathcal{E}_{JVM}[\![B']\!]([e_0,e_1,\ldots,e_k],\mathcal{A}), \\
 & & \text{if } e_k \neq 0 \text{ where } B' \text{ is the basic block} \\
 & & \text{that immediately follows } \texttt{ifeq X}. \\
\mathcal{E}_{JVM}[\![\texttt{invokevirtual f}]\!]([\ldots,q_1,\ldots,q_j],\mathcal{A}) & = & ([\ldots,r],\mathcal{A}) \\
 & & \text{where } r \text{ is the return value of} \\
 & & \mathcal{E}_{JVM}[\![B_f]\!]([\,],\mathcal{A}') \\
 & & \text{and } \mathcal{A}' = (q_0,q_1,\ldots,q_j,\perp,\ldots,\perp), \\
 & & q_0 \text{ is an object reference, and} \\
 & & f(q_1,\ldots,q_j) \text{ is the actual} \\
 & & \text{invocation. } B_j \text{ is the code for a} \\
 & & \text{non-void method } f.
\end{array}
$$

where $\perp$ represents the *undefined value*. This is more of a semantic trick than reality as no activation record entries are ever left undefined at the actual use of the value (the Java compiler assures this), but here we simply wish to denote that the values of the locals might not have been assigned a value by the user code at this moment.

Now let $B = i_0\ i_1\ \ldots\ i_{j-1}\ i_j\ i_{j+1}\ \ldots\ i_k$ be a basic block of instructions (from the control flow graph associated with the code we are executing), and let i_j represent a imaginary `suspend` instruction (as mentioned, eventually it becomes a `return`):

$$
\begin{array}{rcl}
\mathcal{E}_{JVM}[\![B]\!]([\,],\mathcal{A}) & = & \mathcal{E}_{JVM}[\![i_{j+1}\ \ldots\ i_k]\!](s',\mathcal{A}') \text{ where} \\
(s',\mathcal{A}') & = & \mathcal{E}_{JVM}[\![i_0\ i_1\ \ldots\ i_{j-1}]\!]([\,],\mathcal{A})
\end{array} \tag{1}
$$

or equivalently:

$$
\mathcal{E}_{JVM}[\![i_0\ i_1\ \ldots\ i_{j-1}\ i_j\ i_{j+1}\ \ldots, i_k]\!](s,\mathcal{A}) = \mathcal{E}_{JVM}[\![i_0\ i_1\ \ldots\ i_{j-1}\ i_{j+1}\ \ldots\ i_k]\!](s,\mathcal{A});
$$

simply ignore the `suspend` instruction i_j. Naturally, if the code is evaluated in two stages as in the first semantic definition, the *invoking* code must look something like this (assuming B is the body of a method *foo*()):

.
.
foo(..); // Execute $i_0\ \ldots\ i_{j-1}$
foo(..); // Execute $i_{j+1}\ \ldots\ i_k$
.
.

We call this form of resumability "*resumability without parameter change*" since (1) uses $\mathcal{A}'$ and not an $\mathcal{A}''$ where $\mathcal{A}''$ has the same local variables but different parameter values (i.e, the parameters passed to *foo* are exactly the same for both calls). Resumability without parameter changes is not particularly interesting from a mobility standpoint in a process-oriented language; typically we wish to be able to supply different parameters (most often channels, channel ends, and barriers) to the process when it is resumed (especially because the parameters could be channel ends which allow the process to interact with a new environment, that is, the process receiving the mobile). It turns out that if we can implement resumability without parameter change in the JVM (i.e., devise a method of restoring activation records between invocations), then the more useful type of resumability with parameter change comes totally free of charge! For completeness, let us define this as well:

Let us consider again the basic block code $B = i_0\, i_1\, \ldots\, i_{j-1}\, i_j\, i_{j+1}\, \ldots\, i_k$, where again i_j represents a `suspend` instruction that returns control to the caller, and let us assume that the code in B is invoked by the calls *foo*$(v_1, \ldots, v_n)$, and *foo*$(v'_1, \ldots, v'_n)$ respectively.

$$\begin{array}{rcl} \mathcal{E}_{JVM}[\![i_0\, \ldots\, i_{j-1}\, i_j\, i_{j+1}\, \ldots\, i_k]\!](s, \mathcal{A}) & = & \mathcal{E}_{JVM}[\![i_{j+1}\, \ldots\, i_k]\!](s'', \mathcal{A}'') \text{ where} \\ \mathcal{A} & = & [a_0 = \mathit{this}, a_1 = v_1, a_2 = v_2, \ldots, a_n = v_n, a_{n+1} = \perp, \ldots, a_{n+m} = \perp] \\ \mathcal{A}'' & = & [a_0 = \mathit{this}, a''_1 = v'_1, a''_2 = v'_2, \ldots, a''_n = v'_n, a''_{n+1} = a'_{n+1}, \ldots, a''_{n+m} = a'_{n+m}] \\ (s', \mathcal{A}') & = & \mathcal{E}_{JVM}[\![i_0\, i_1\, \ldots\, i_{j_1}]\!](s, \mathcal{A}) \\ \mathcal{A}' & = & [a'_0, \ldots, a'_{n+m}] \end{array}$$

We call this "*resumability with parameter changes*". The above extends to loops (through multiple basic block code segments), and to code blocks with more than one `suspend` instruction. As we can see from the semantic function $\mathcal{E}_{JVM}$, the activation record must "survive" between invocations/suspend-resumptions; local variables are saved and restored, parameters are not stored and are changed according to each invocation's actual parameters. Naturally we must assure that the locations in the activation record holding the locals are restored before they are referenced again.

2. Target Bytecode Structure

All the extra code needed to save and restore state upon suspension and resumption can be generated by the ProcessJ code generator; only the code associated with resuming execution in the middle of a code block will require bytecode rewriting.

Let us consider a very simple example with a single `suspend` statement (the following is a snippet of legal ProcessJ code):

```
type proc mobileFooType();
mobile proc void foo() implements mobileFooType {
  int a;
  a = 0;
  while (a == 0) {
    a = a + 1;
    suspend;
    a = a - 1;
  }
}
```

The resulting bytecode would look something like this:

```
public void foo();
  Code:
   0:    iconst_0
   1:    istore_1       ; a = 0;
   2:    iload_1
   3:    ifne 20        ; while (a == 0) {
   6:    iload_1
   7:    iconst_1
```

```
    8:   iadd
    9:   istore_1      ;   a = a + 1;
    10:  ???           ;   suspend handled here
    13:  iload_1
    14:  iconst_1
    15:  isub
    16:  istore_1      ;   a = a - 1;
    17:  goto 2        ; }
    20:  return
}
```

Since the `suspend` is handled in line 10 by inserting a `return` instruction, we need to store the local state *before* the return, and upon resuming the execution, control must be transferred to line 13 rather than starting at line 0 again, and the state must be restored *before* executing line 13. This requires three new parts inserted into the bytecode:

1. Code to save the local state (in the above example the local variable `a`) before the `suspend` statement in line 10.
2. Code to restore the local state before resuming execution of the instructions *after* the previous `suspend` statement, that is, after line 1 and before line 13.
3. Code to transfer control to the right point of the code depending on which `suspend` was most recently executed (before line 0).

Thus the goal is to automate the generation of such code. 1 and 2 can be done completely in Java by the ProcessJ code generator, and 3 can be done by a combination of Java code and bytecode rewriting.

Before turning to this, let us first mention a few restrictions that mobile processes have in ProcessJ: processes have no return type (the equivalent in Java is a void method), and mobile processes cannot be recursive. The semantics for a recursive mobile process are not yet clear, and we do not see any obvious need for recursion of mobiles at this time.

3. Source Code Rewriting

As mentioned, the ProcessJ code generator emits Java source code, which is then compiled using the Java compiler, and the resulting bytecode is subsequently rewritten.

Let us describe the Java code emitted from the ProcessJ compiler first. To transform the *foo* method from the previous section into a resumable process, we encapsulate it in a Java *Object* that contains two auxiliary fields as well as the process rewritten as a Java method and two dummy placeholder methods.

1. The method is encapsulated in a new class:

```
public class Foo {

  private Object[] actRec;
  private static void suspend() { }
  private static void resume() { }
  private int jumpTarget = 0;

  public void foo() {
    ...  switch statement that jumps to resume point.
    int a;
    a = 0;
    while (a == 0) {
      a = a + 1;
      ...  code to save the current state.
```

```
   suspend();
   resume();
   ...  code to restore the previous state.
   a = a - 1;
  }
 }

}
```

where `actRec` represents a saved activation record. The `suspend` and `resume` methods are just dummy methods that are added to satisfy the compiler (more about these later). Finally, a field `jumpTarget` has been added. `jumpTarget` will hold non-negative values (0 if the execution is to start from the beginning), and 1, 2, ... if the execution is to resume from somewhere within the code (i.e., not from the start).

2. The code for `foo` must also be rewritten to support resumability:
   - Support must be added for saving and restoring the local variable part of the JVM activation record; this is done through the `Object` array `actRec`.
   - A `lookupswitch` JVM instruction [9] must be added; based on the `jumpTarget` field it will jump to the instruction following the last `suspend` executed. A simple Java `switch` statement that switches on the `jumpTarget` will translate to such a `lookupswitch` instruction.

3.1. Saving Local State

A Java activation record consists of two or three parts: Local variables, parameters and for non-static methods, a reference to *this* stored at address 0 in the activation record. The layout is illustrated in Figure 1. Recall, we need the encapsulated method to be non-static. Since *this* never changes for an object, and since each resumption of the method provides a new set of parameters, all we have to save is the set of locals. As we rely on the JVM invocation instructions, each invocation of a method creates its own new JVM activation record that contains *this*, the provided parameters, and room for the locals. The first step in resuming the method is to restore the locals to the state they were in when the method was suspended. We use an array of `Objects` to store the m locals. If the field `jumpTarget` has value 0, representing that the method starts from the top (this is the initial invocation of the process), no restoration of locals is necessary as the execution starts from the beginning of the code (and the ProcessJ and Java compilers have assured that no path to a use of an uninitialized variable exists). On subsequent resumptions, the saved array of locals must be restored, and the value of the field `jumpTarget` determines from where execution should continue (immediately after the `return` instruction that suspended the previous activation of the method).

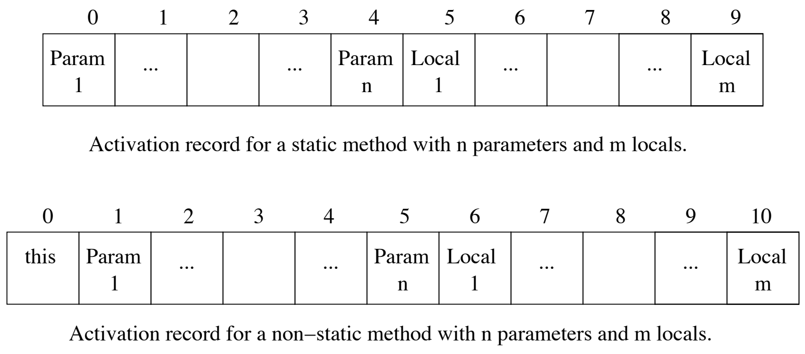
0 1 2 3 4 5 6 7 8 9
Param 1
...
...
Param n
Local 1
...
...
Local m
Activation record for a static method with n parameters and m locals.
0 1 2 3 4 5 6 7 8 9 10
this
Param 1
...
...
Param n
Local 1
...
...
Local m
Activation record for a non–static method with n parameters and m locals.

Figure 1. JVM Activation Records.

If for example a method has locals a, b, and c of integer type, we can save an Object array with their values in the following way by using the auto-boxing feature provided by the Java compiler:

```
actRec = new Object[] { a, b, c};
jumpTarget = ...;
```

and they can be restored in the following manner:

```
a = (Integer)actRec[0];
b = (Integer)actRec[1];
c = (Integer)actRec[2];
```

Both of these code blocks are generated by the ProcessJ code generator, the former before the suspend and the latter after.

3.2. Resuming Execution

When a method is resumed (by being invoked), the jumpTarget field determines where in the code execution should continue; namely immediately after the return that suspended the previous invocation. We cannot add Java code that gets translated by the Java compiler for this; in order to do so we would need a *goto* instruction (as well as support for labels), and although goto is a reserved word in Java, it is not currently in use. To achieve this objective, we must turn to bytecode rewriting.

We need to insert a lookupswitch instruction that switches on the jumpTarget field, and jumps to the address of the instruction following the return that suspended the previous invocation. We can generate parts of the code with help from the Java compiler; we insert code like this at the very beginning of the generated Java code:

```
switch (jumpTarget) {
  case 1: break;
  default: break;
}
```

There will be as many cases as there are suspends in the ProcessJ code. We get bytecode like this:

```
4:    lookupswitch{
               1: 24;
               default: 27 }
24:   goto    27
27:   ...
```

In the rewriting of the bytecode all we have to do is replace the absolute addresses (24 and 27) in the switch by the addresses of the resume points. The addresses of the resume points can be found by keeping track of the values assigned to the jumpTarget field in the generated Java code or by inspecting the bytecode as explained below.

Since we replaced the suspend keyword by calls to the dummy suspend() and resume() method, we can look for the static invocation of resume():

```
52: aload_0
53: iconst_1
54: putfield         #5; //Field jumpTarget:I
57: invokestatic     #18; //Method suspend:()V
60: invokestatic     #19; //Method resume:()V
63: ...
```

(here found in line 60), and the two instructions immediately before the suspend call will reveal the jumpTarget value that the address (60) should be associated with. The instruction in line 53 will be one of the iconst_X (X=1,2,3,4,5) instructions or a bipush instruction. For the above, the lookupswitch should be rewritten as:

```
4:    lookupswitch{
                 1: 60;
                 default: 27 }
24:   goto     27
27:   ...
```

Furthermore the lines 57 and 60 must be rewritten to be a return (this cannot be done before compile time, as the Java compiler will complain about unreachable code) and a nop respectively. Alternatively, the resume method can be removed and the jump target will be the instruction following the suspend call.

4. Example

Let us rewrite the previous example to obtain this new *Foo* class:

```
public class Foo {

  private Object[] actRec;
  private static void suspend() { }
  private static void resume() { }
  private int jumpTarget = 0;

  public void foo() {
    int a;
    switch (jumpTarget) {              // Begin: jump
      case 1: break;
      default: break;
    }                                  // End: jump
    a = 0;
    while (a == 0) {
      a = a + 1;
      actRec = new Object[] { a };     // Begin: save state
      jumpTarget = 1;                  // End: save state
      suspend();
      resume();
      a = (Integer)actRec[0];          // restore state
      a = a - 1;
    }
    jumpTarget = 0;                    // Reset jumpTarget
  }

}
```

Note that the jumpTarget should be set to 0 before each original return statement to assure that the next time the process is resumed, it will start from the beginning. This is very close to representing the code we really want, and best of all, it actually compiles.

Note also that the line saving local state must include *all locals in scope*. If the rewriting is done solely in bytecode, this would require an analysis of the control flow graph (CFG) associated with the code – like the approach taken in the Southampton Portable Occam Compiler (SPOC) [12]. But since we generate the store code as part of the code generation from the ProcessJ compiler, we have access to all scope information. It is further simplified by the fact that scoping rules for ProcessJ follows those of Java (when removing fields and objects).

Let us look at the generated bytecode. Because of the incomplete switch statement, every invocation of `foo` will always execute the `a = 0` statement (i.e. start from the beginning):

```
public void foo();
  Code:
   0:   aload_0
   1:   getfield jumpTarget I   // switch ( jumpTarget ) {
   4:   lookupswitch{
                1: 24;          //   case 1: ...
                default: 27 }   //   default: ...
  24:   goto    27              // }
  27:   iconst_0
  28:   istore_1                // a = 0;
  29:   iload_1                 // while (a == 0) {
  30:   ifne 83
  33:   iload_1
  34:   iconst_1
  35:   iadd
  36:   istore_1
  37:   aload_0                 //   a = a + 1;
  38:   iconst_1
  39:   anewarray java/lang/Object
  42:   dup
  43:   iconst_0
  44:   iload_1
  45:   invokestatic java/lang/Integer.
                                valueOf(I)Ljava/lang/Integer;
  48:   aastore
  49:   putfield actRec [Ljava/lang/Object;
                               //   actRec = new Object[]{a};
  52:   aload_0
  53:   iconst_1
  54:   putfield jumpTarget I   //   jumpTarget = 1;
  57:   invokestatic suspend()V //   suspend;
  60:   invokestatic resume()V  //   // resume point
  63:   aload_0
  64:   getfield actRec [Ljava/lang/Object;
  67:   iconst_0
  68:   aaload
  69:   checkcast java/lang/Integer
  72:   invokevirtual java/lang/Integer.intValue()I
  75:   istore_1                //   a = (Integer)actRec[0];
  76:   iload_1
  77:   iconst_1
  78:   isub
  79:   istore_1                //   a = a - 1;
  80:   goto 29                 // }
  83:   aload_0
  84:   iconst_1
  85:   putfield jumpTarget I   // jumpTarget = 0;
  88:   return
}
```

Lines 0–24 represent the switch statement, 38–54 the save state code, 57–60 the suspend/resume placeholder method calls, 63–75 the restore state code, and 83–88 the rewritten original return code.

As pointed out above, this code is not correct; a number of things still need to be changed:

- Line 4 is the jump table that must be filled with correct addresses. If the field `jumpTarget` equals 1, execution continues at the invocation of the dummy `resume()` method – line 60. The default label is already correct and can be left unchanged.
- Line 57, the dummy `suspend()` invocation, should be replaced by a `return` instruction (we could not simply place a Java return instruction in the source code because the compiler would complain about the code following the return statement being unreachable).
- Line 60, the dummy `resume()` invocation should be replaced by a `nop`. This only serves as a placeholder; theoretically we could have used address 63 in the `lookupswitch`.

An example of use in ProcessJ could be this:

```
proc void sender(chan<mobileFooType>.write ch) {
  // create mobile
  mobileFooType mobileFoo = new mobile foo;
  // invoke foo (1st invocation)
  mobileFoo();
  // send to different process
  ch.write(mobileFoo);
}

proc void receiver(chan<mobileFooType>.read ch) {
  mobileFooType mobileFoo;
  // receive mobileFooType process
  mobileFoo = ch.read();
  // invoke foo (2nd invocation)
  mobileFoo();
}

proc void main() {
  chan<MobileFooType> ch;
  par {
    sender(ch.write);
    receiver(ch.read);
  }
}
```

The resulting Java/JCSP code looks like this:

```
import org.jcsp.lang.*;

public class PJtest {

  public static void sender(ChannelOutput ch_write) {
    Foo mobileFoo = new Foo();
    mobileFoo.foo();
    ch_write.write(mobileFoo);
  }

  public static void receiver(ChannelInput ch_read) {
    Foo mobileFoo;
    mobileFoo = (Foo)ch_read.read();
    mobileFoo.foo();
  }
```

```
    public static void main(String args[]) {

      final One2OneChannel ch = Channel.one2one();

      new Parallel(new CSProcess[] {

        new CSProcess() {
          public void run() {
            sender(ch.out());
          }
        },

        new CSProcess() {
          public void run() {
            receiver(ch.in());
          }
        }

      }).run();

    }

  }
```

One small change is still needed to support mobility across a network. Since the generated Java code is a class, this can be made serializable by making the generated classes implement the `Serializable` interface. An object of such a class can now be serialized and sent across a network. Welch et al. [18] provide such a mechanism in their `jcsp.net` package as well.

Since the rewriting described encapsulates the mobile process in a new class, objects of that class can be sent as data across the network and the mobile process inside that object can be resumed by invoking the method that encapsulates the mobile process (*mobileFoo.foo()* above).

5. Related Work and Other Approaches

Approaches to process mobility can be categorized as either transparent or non-transparent, sometimes termed strong and weak migration (mobility), respectively [2,6]. With non-transparent mobility the programmer must explicitly provide the logic to suspend and resume the mobile process whenever necessary. Existing systems such as `jcsp.mobile` [3,4] already provide this functionality. Transparent mobility significantly eases the task of the programmer, but requires support from the run-time system which does not exist within the Java Virtual Machine.

Some early approaches to supporting resumable programs in Java involved modification of the JVM itself [2]. In our view, however, one of the most important advantages of targeting the JVM is portability across the large installed base of Java runtime environments. Therefore any approach that extends the JVM directly is of limited utility.

Some success has been demonstrated using automated transformation of Java source code [6]. Due to the lack of support within the language for labeled gotos, this approach suffers from a proliferation of conditional guards and a corresponding increase in code size.

Bytecode-only transformations methods targeting general thread resumability in Java are explored in [1] and [16]. These approaches require control flow analysis of the bytecode in order to generate code for the suspend point. Alternatively, the Kilim [15] actor-based framework uses a CPS bytecode transformation to support cooperatively scheduled lightweight threads (fibers) within Java.

Another example of bytecode state capture can be found in Java implementation of the object-oriented language Python (Jython [13]) in order to support generators, a limited form of co-routines [11]. This is perhaps the most similar to our implementation, even though generator functions are somewhat different in concept and application from ProcessJ procedures. We wish, however, to be able to utilize the existing Java compilers to produce optimized bytecode with our back-end.

The process-oriented nature of ProcessJ allows us to adopt a simple hybrid approach that combines Java source and bytecode methods.

6. Conclusion

In this paper we have shown that a compiler for a process-oriented language can provide transparent mobility using the existing Java compiler tool chain with minimal modification. We developed a simple way to generate Java source code and rewrite Java bytecode to support resumability and ultimately process mobility for the ProcessJ language.

We described the Java source code generated by the ProcessJ compiler, and also demonstrated how to rewrite the Java bytecode to save and restore local state in between resumptions of code executions as well as how to assure that execution continues with the same local state (but with possibly new parameter values) at the instruction following the previous suspension point.

7. Future Work

A number of interesting issues remain to be addressed. For ProcessJ, where we have channels, an interesting problem arise when assigning a parameter of channel end type to a local variable. If a local variable holds a reference to a channel end, and the process is suspended and sent to a different machine, the end of the channel now lives on a different physical machine. This is not a simple problem to solve; for occam-π, the pony [14] system addresses this problem. One way to approach this problem is to include a channel server, much like the one found in JCSP.net [18], that keeps track of where channel ends are located; this is the approach we are working with for the MPI/C code generator. Mobile channels can be handled in the same way, but are outside the scope of this paper.

Other issues that need to be addressed include how resource management is to be handled; if a mobile process contains references to (for example) open files that are not available on the JVM to which the process is sent, accessing this file becomes impossible. We may wish to enforce certain kinds of I/O restrictions on mobile processes in order to more clearly define their behavior under mobility.

With a little effort, the saving and restoration could be gathered at the beginning and the end of the method saving some code/instructions, but for clarity reasons we used a different approach (as presented in this paper).

8. Acknowledgments

This work was supported by the UNLV President's Research Award 2008/2009. We would like to thank the reviewers who did a wonderful job in reviewing this paper. Their comments and suggestions have been valuable in producing a much stronger paper.

References

[1] Sara Bouchenak. Techniques for Implementing Efficient Java Thread Serialization. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA03)*, pages 14–18, 2003.

[2] Sara Bouchenak and Daniel Hagimont. Pickling Threads State in the Java System. In *Third European Research Seminar on Advances in Distributed Systems*, 2000.

[3] Kevin Chalmers and John Kerridge. `jcsp.mobile`: A Package Enabling Mobile Processes and Channels. In Jan Broenink and Herman Roebbers and Johan Sunter and Peter Welch and and David Wood, editor, *Communicating Process Architectures 2005*, pages 109–127, 2005.

[4] Kevin Chalmers, John Kerridge, and Imed Romdhani. Mobility in JCSP: New Mobile Channel and Mobile Process Models. In Alistair McEwan and Steve Schneider and Wilson Ifill and Peter Welch, editor, *Communicating Process Architectures 2007*, pages 163–182, 2007.

[5] Jack Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[6] Stefan Fünfrocken. Transparent Migration of Java-based Mobile Agents - Capturing and Reestablishing the State of Java Programs. In *Mobile Agents*, pages 26–37. Springer Verlag, 1998.

[7] R. Hieb and R.K. Dybvig. Continuations and Concurrency. *ACM Sigplan Notices*, 25:128136, 1990.

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

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, 2nd Edition*. Prentice Hall PTR, 1999.

[10] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[11] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31:1–31, 2009.

[12] D.A. Nicole, M. Debbage, M. Hill, and S. Wykes. Southampton's Portable Occam Compiler (SPOC). In A.G. Chalmers and R. Miles, editors, *Proceedings of WoTUG 17: Progress in Transputer and Occam Research*, volume 38 of *Concurrent Systems Engineering*, pages 40–55, Amsterdam, The Netherlands, April 1994. IOS Press. ISBN: 90-5199-163-0.

[13] Samuele Pedroni and Noel Rappin. *Jython Essentials*. O'Reilly Media, Inc., 2002.

[14] Mario Schweigler and Adam T. Sampson. pony - The occam-π Network Environment. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 77–108, Amsterdam, The Netherlands, September 2006. IOS Press.

[15] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Procedings of the European Conference on Object Oriented Programming (ECOOP)*, pages 104–128. Springer, 2008.

[16] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable Support for Transparent Thread Migration in Java. In *ASA/MA*, pages 29–43. Springer Verlag, 2000.

[17] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Process Techniques and Applications*, volume 1, pages 51–57, Las Vegas, Nevada, USA, June 2000. CSREA, CSREA Press. ISBN: 1-892512-52-1.

[18] Peter H. Welch, Jo R. Aldous, and Jon Foster. CSP Networking for Java (JCSP.net). *Lecture Notes in Computer Science*, 2330:695–708, 2002.

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

[20] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes: introducing occam-π. 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.

[21] Peter H. Welch, Jim Moores, Frederick R. M. Barnes, and David C. Wood. *The KRoC Home Page*. http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[22] Peter H. Welch and Jan B. Pedersen. Santa Claus - with Mobile Reindeer and Elves. In *Proceedings of Communicating Process Architectures*, 2008.

[23] Peter H. Welch and David C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments*, volume 47 of *Concurrent Systems Engineering*, pages 143–166, Amsterdam, The Netherlands, March 1996. World occam and Transputer User Group, IOS Press.