Resumable Java Bytecode Process Mobility for the JVM

Jan Pedersen and Brian Kauke

Overview

The ProcessJ language

Transparent process mobility

Implementing mobility in the Java/JCSP translation

Intro to ProcessJ

- ProcessJ is a new process-oriented language
- Syntax close to Java and a semantics close to occam-π
- Scoping rules similar to Java
- Provides an approachable environment for broader audience of imperative programmers
- Multiple target platforms

Intro to ProcessJ

Multiplexer Example (Occam)

```
PROC mux (CHAN INT in1?, in2?, out!)
  INT data:
  WHILE TRUE
    ALT
      in1 ? data
        out ! data
      in2 ? data
        out ! data
:
```

Intro to ProcessJ

Multiplexer Example

```
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);
        }
    }
}
```

Intro to ProcessJ

Sequential subset mimics Java/C

- Semicolon delimited statements
- Assignment, arithmetic
- Conditionals (if, switch statements)
- For, While loops

Intro to ProcessJ

Channels look like JCSP channels

Special syntax for:

- Par blocks

```
par { ... }
```

- Alt blocks

```
alt { ... }
```

- Case protocols

```
ProtocolType m;
switch(m = c.read()) {
  case tag1 : ...
  case tag2 : ...
  ...
}
```

Our Compiler

Multiple backends

- C (MPI), Java (JCSP), Occam-π

- Java backend targets Java source code

- Take advantage of javac's optimizer

JCSP / JCSP.net

Offers most of the tools needed

- Serialization

- Network Classloading

- Mobile Channels, Processes

Process Mobility

JCSP supports mobile processes

```
MobileProcess p = in.read();
p.plugin(c1, c2);
p.run();
out.write(p);
```

Correct implementation of MobileProcess is left to the programmer.

Transparent Mobility

Certain behavior implied at suspend point

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

Challenges

JVM does not cleanly support transparent mobility

- No continuations

- Limited control flow constructs (no gotos)

- Nested scopes
  (determining content of activation record is not simple)

Suspending

Support must be added for saving and restoring the local variable part of the JVM activation record

We support parameter change, so parameters do not need to be saved

Control must be returned to caller

Resuming

Restore activation record with saved locals

Jump to appropriate control point
(instruction following the previous suspend)

Goto would be nice here!

General Rewriting Methods

Source code rewriting

- Must separate code into regions between suspend calls

- Java compiler sensitive to visibility of declared variables

- Large overhead in generated lines of code

General Rewriting Methods

Bytecode rewriting

- We lose a lot of information looking at bytecode only

- Requires control flow analysis on bytecode to save and restore variables correctly

Generated Source Code

```
public class Foo {
  public void foo() {
    while(1) {
      ...
      suspend();
      ...
    }
  }
}
```

Generated Source Code

```
public class Foo {
  private Object[] actRec;
  public void foo() {
    while(1) {
      ... // Store locals to actRec
      suspend();
      resume();
      ... // Restore from actRec
    } }
  public static void suspend() {}
  public static void resume() {} }
```

Generated Source Code

```
public class Foo {
  private Object[] actRec;
  private int jumpTarget = 0;
  public void foo() {
    switch (jumpTarget) {
      case 1: break;
      default: break;
    }
    while(1) {
      ... // Store locals to actRec
      jumpTarget = 1;
      suspend();
      resume();
      ... // Restore from actRec
    }
    jumpTarget = 0;
  }
  ... }
```

Generated Source Code

```
public class Foo {
  private Object[] actRec;
  private int jumpTarget = 0;
  public void foo() {
    switch (jumpTarget) {
      case 1: goto #1;
      default: break;
    }
    while(1) {
      ... // Store locals to actRec
      jumpTarget = 1;
      suspend();
      resume(); #1
      ... // Restore from actRec
    }
    jumpTarget = 0;
  }
  ... }
```

Bytecode Transformation

```
...
4:  lookupswitch {
        1: 24;
        default: 27;
    }
24: goto 27
27: ...
...
57: invokestatic suspend()v
60: invokestatic resume()V // #1
...
```

Bytecode Transformation

```
...
4:  lookupswitch {
        1: 60;
        default: 27;
    }
24: goto 27
27: ...
...
57: invokestatic suspend()v
60: nop
...
```

Bytecode Transformation

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

Example

```
proc void sender(chan<MobileType>.write c) {
  Foo foo = new mobile Foo;
  foo();
  c.write(foo);
}

proc void receiver(chan<MobileType>.read c) {
  Foo foo = c.read();
  foo();
}

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

Result

ProcessJ includes a simple method for defining mobile processes in the style of occam-π

We have described a method for translating this into Java with JCSP using minimal bytecode manipulation