JVMCSP
Approaching Billions of Processes on a Single-Core JVM

Cabel Shrestha & Matt B. Pedersen
We presented
• ProcessJ
  (Process-Oriented Language)
• Handcrafted JVM runtime: LiteProc (proof of concept)
• ~95,000,000 concurrent processes
An implemented code generator in ProcessJ

New improved runtime
  Faster
  Handles more processes
ProcessJ compiler produces Java source code.

Compiled with Javac.

Instrumented with ASM byte code manipulation tool.

Jar’d together with runtime
User-level scheduler

Cooperative, non-preemptive.

```java
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);
}
```
Essential Questions

- How does a procedure yield?
- When does a procedure yield and who decides?
- How is a procedure restarted after yielding?
- How is local state maintained?
- How are nested procedure calls handled when the innermost procedure yields?
How does a procedure yield? When does it yield and who decides?

**CPA 2014 version**
- Yields by calling return
- Procedures voluntarily give up the CPU at synchronization points

**JVMCSP**
- Yields by calling return
- Procedures voluntarily give up the CPU at synchronization points

Reads, writes, barrier syncs, alts, timer operations: procedure returns to scheduler (Bytecode: return)
How is a procedure restarted?

CPA 2014

- Procedure is simply recalled by scheduler

JVMCSP

- Procedure is simply recalled by scheduler

- How do we ensure that local state survives?
- How do we avoid restart from the top of the code?
Preservation of Local State

CPA 2014

- An activation record structure was used to store locals.
- Each procedure is a class with an activation stack.

JVMCSP

- All locals and fields have been converted to fields.
- Each procedure is a class.
Correct Resumption

CPA 2014

- Insert an empty switch statement at the top of the code to hold jumps.
- Instrument (by hand in decompiled bytecode) jumps to the correct resume points.

JVMCSP

- Insert an empty switch statement at the top of the generated code (source) to hold jumps.
- Instrument (by using ASM) jumps to the correct resume points.

A resume point counter (called runlabel) is kept for each process to remember where to continue.
Each synchronization point is a yield point:

L1:

.. synchronize (read, sync etc)

if (succeeded)
    yield(L2);  // return to L2 when resumed
else
    yield(L1);  // return to L1 when resumed

L2:
Each synchronization point is a yield point:

```plaintext
label(1);
.. synchronize (read, sync etc)
if (succeeded)
    yield(2);
else
    yield(1);
label(2);
```

`yield(i)` will set the runlabel for the process object to `i`. 
Correct Resumption (ASM Instrumentation)

```java
label(1);  // Place the instruction for resumption
.. synchronize
if (succeeded)
    yield(2);
else
    yield(1);
label(2);
```

```
61: aload_0
62: iconst_1
63: invokevirtual label/(I)V
66: ...
...
```

```
61: nop
62: nop
63: nop
64: nop
65: nop
66: ...  // Dummy invocations are removed.
...  // Code following the resumption
```
- This address (61) is associated with runlabel 1.
- Upon resumption, the code must jump to address 61 if the runlabel is 1.

```
61:  nop
62:  nop
63:  nop
64:  nop
65:  nop
66:  ...
... 
```
Correct Resumption (Generated Code)

Generated Java Code

(top of the code)

```java
switch (runlabel) {
    case 0: resume(0);
    break;
    case 1: resume(1);
    break;
    ...
    case k: resume(k);
    break;
}
```

Equivalent Java Bytecode

```
0: aload 0
1: getfield runLabel
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
...```
Correct Resumption (ASM Instrumentation)

0: aload 0
1: getfield runLabel
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
...

0: aload 0
1: getfield runLabel
4: tableswitch // 0 to 2
   0: 32
   1: 35
   2: 43
   default: 48
32: goto 48
35: nop
36: nop
37: goto 61
40: goto 48
...

Placeholder code replaced by nop instructions and gotos adjusted to the correct label addresses
yield(2);  
78: aload_0  
79: iconst_2  
80: invokevirtual yield/(I)V  
83: goto 100  
...  
100: return

Becomes

yield(2) sets the runLabel field.
proc void foo(pt_1 pn_1, ..., tp_n pn_n) {
    ...
    lt_1 ln_1;
    ...
    lt_m ln_m;
    ... statements ...
}
From ProcessJ to Java

 ```java
public class A {
    public static class foo extends PJProcess {
        pt 1 pn 1;
        pt 2 pn 2;
        ...
        lt 1 ln 1;
        ...
        lt m ln m;
        public foo(pt 1 pn 1, ..., tp n pn n) {
            this.pn 1 = pn 1;
            ...
            this.pn n = pn n;
        }
        public void run() {
            switch (runlabel) {
                case 0:
                    resume(0);
                    break;
                case 1:
                    resume(1);
                    break;
                ...
                case k:
                    resume(k);
                    break;
                ...
            }
        }
    }
}
```

Process foo lives in a file called A.pj
public class A {
    public static class foo extends PJProcess {
        pt \_1 pn\_1;
        pt\_2 pn\_2;
        ...
        lt\_1 ln\_1;
        ...
        lt\_m ln\_m;
    }
}

Locals and Parameters are turned into fields
public class A {
    public static class foo extends PJProcess {
        pt1 pn1;
        pt2 pn2;
        ...
        lt1 ln1;
        ...
        lt_m ln_m;

        public foo(pt1 pn1, ..., tp_n pn_n) {
            this.pn1 = pn1;
            ...
            this.pn_n = pn_n;
        }
    }
}

Constructors set the parameters
public class A {
    public static class foo extends PJProcess {
        pt\textsubscript{1} \textit{p_1};
        pt\textsubscript{2} \textit{p_2};
        ...
        lt\textsubscript{1} \textit{l_1};
        ...
        lt\textsubscript{m} \textit{l_m};

        public foo(pt\textsubscript{1} \textit{p_1}, \ldots, 
                    tp\textsubscript{n} \textit{p_n}) {
            this.\textit{p_1} = \textit{p_1};
            ... 
            this.\textit{p_n} = \textit{p_n};
        }
    }

    public void run() {
    }
}

\textbf{run method is called by the scheduler}
public class A {
    public static class foo extends PJProcess {
        pt_1 pn_1;
        pt_2 pn_2;
        ...
        lt_1 ln_1;
        ...
        lt_m ln_m;

        public foo(pt_1 pn_1, ..., tp_n pn_n) {
            this.pn_1 = pn_1;
            ...
            this.pn_n = pn_n;
        }
    }
}

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

resume() calls replaced by jumps to label()s
public class A {
    public static class foo extends PJProcess {
        pt1 pn1;
        pt2 pn2;
        ...
        lt1 ln1;
        ...
        lt_m ln_m;

        public foo(pt1 pn1, ..., tp_n pn_n) {
            this.pn1 = pn1;
            ...
            this.pn_n = pn_n;
        }
    }
    public void run() {
        switch (runlabel) {
            case 0: resume(0);
                break;
            case 1: resume(1);
                break;
            ...
            case k: resume(k);
                break;
        }
    }
}

Code is translated ProcessJ + generated primitives
Yielding in Nested Calls

CPA 2014

- Maintain a complex activation stack.
  - Constant creation and destruction of activation records.
  - Resumptions started from the outermost procedure and worked its way in.

JVMCSP

- Calls of procedures that may yield

\[ \text{par} \, \{ \quad f(x) \quad \} \]
PJProcess represents a process.
PJPPar represents a par block.
PJChannel represents a channel.
  PJOne2OneChannel, PJOne2ManyChannel, PJMany2OneChannel, PJMany2ManyChannel
PJBarrier represents a barrier.
PJTimer represents a timer.
PJAlt represents an alt.
par { 
  f(8);
  g(9);
}

final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
Create new PJPar object with 2 processes

```java
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
setNotReady();
yield(1);
label(1);
```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
setNotReady();
yield(1);
label(1);
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
setNotReady();
yield(1);
label(1);
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
```java
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Set process with par not ready to run
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
(new A.g(8) {
    public void finalize() {
        par1.decrement();
    }
}).schedule();
setNotReady();
yield(1);
label(1);
When ready again continue here
x = in.read();

... label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
    yield(3);
} else {
    setNotReady();
in.addReader(this);
    yield(2);
}
label(3);
... 
label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
    yield(3);
} else {
    setNotReady();
in.addReader(this);
    yield(2);
}
label(3);
... 
label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
    yield(3);
} else {
    setNotReady();
in.addReader(this);
yield(2);
}
label(3);
... label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
    yield(3);
} else {
    setNotReady();
    in.addReader(this);
    yield(2);
}
label(3);
... 
label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
    yield(3);
} else {
    setNotReady();
    in.addReader(this);
    yield(2);
}
label(3);

If no, set this process not read to run
... label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
yield(3);
} else {
    setNotReady();
in.addReader(this);
yield(2);
} label(3);

Add the reader to the channel
... label(2);
if (in.isReadyToRead(this)) {
    x = in.read();
yield(3);
} else {
    setNotReady();
in.addReader(this);
yield(2);
}
label(3);
Channel writes are similar to reads.

Channels with shared ends must be claimed.

Functionality to claim and unclaim is included in PJ...2... channel classes.
Timers and the Timer Queue

- Timers are handled by a TimerQueue and a TimerHandler.
  - The TimerQueue is a delay-queue.
  - Timeout calls cause insertions into TimerQueue

- TimerHandler dequeues expired timers from the TimerQueue.
  - Sets corresponding processes ready to run.
Timers and the Timer Queue
Timers

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

Becomes

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

\[ t.\text{start}(100) \text{ will insert a new timer object into the TimerQueue.} \]
Barriers

sync(b);

Becomes

b.sync(this);
yield(1);
label(1);

b.sync(this) will
* decrement the barrier’s process counter
* enqueue the process in the barrier’s process list
* set itself not ready

When counter reaches 0 all processes are set ready.
We probably do not have time for this... but they are cool.
Results

- Timing
- Context switching
- Max process count
- Overhead (we will skip this one too)
Timings and Context Switches

Mandelbrot fractal image 4,000 x 3,000 (12,000,000 pixels)

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (Sec.)</th>
<th># Processes</th>
<th># Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Sequential</td>
<td>6.24</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ProcessJ Sequential</td>
<td>6.21</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ProcessJ row parallel</td>
<td>6.05</td>
<td>3,001</td>
<td>3,001</td>
</tr>
<tr>
<td>ProcessJ pixel parallel</td>
<td>31.98</td>
<td>12,000,001</td>
<td>12,003,001</td>
</tr>
</tbody>
</table>
## CommsTime

<table>
<thead>
<tr>
<th></th>
<th>Mac / OS X</th>
<th></th>
<th>AMD / Linux</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPA’14</td>
<td>JCSP</td>
<td>JVMCSP</td>
<td>CPA’14</td>
</tr>
<tr>
<td>μs/iteration</td>
<td>9.26</td>
<td>27.00</td>
<td>8.30</td>
<td>13.56</td>
</tr>
<tr>
<td>μs/communication</td>
<td>2.31</td>
<td>6.00</td>
<td>2.08</td>
<td>3.90</td>
</tr>
<tr>
<td>μs/context switch</td>
<td>1.32</td>
<td>3.00</td>
<td>0.69</td>
<td>1.94</td>
</tr>
</tbody>
</table>
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);
        }
    }
}
<table>
<thead>
<tr>
<th># Processes</th>
<th># Context Switches</th>
<th>Execution Time (Secs.)</th>
<th>Memory Usage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,000,001</td>
<td>15,000,002</td>
<td>7.53</td>
<td>1.79</td>
</tr>
<tr>
<td>10,000,001</td>
<td>22,500,002</td>
<td>16.03</td>
<td>3.02</td>
</tr>
<tr>
<td>14,000,001</td>
<td>30,000,002</td>
<td>25.86</td>
<td>4.10</td>
</tr>
</tbody>
</table>
# Max Process Count

<table>
<thead>
<tr>
<th># Processes</th>
<th># Context Switches</th>
<th>Execution Time (Secs.)</th>
<th>Memory Usage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,000,001</td>
<td>15,000,002</td>
<td>7.53</td>
<td>1.79</td>
</tr>
<tr>
<td>10,000,001</td>
<td>22,500,002</td>
<td>16.03</td>
<td>3.02</td>
</tr>
<tr>
<td>14,000,001</td>
<td>30,000,002</td>
<td>25.86</td>
<td>4.10</td>
</tr>
<tr>
<td>210,000,001</td>
<td>450,000,002</td>
<td>642.80</td>
<td>63.91</td>
</tr>
<tr>
<td>350,000,001</td>
<td>750,000,002</td>
<td>1,235.12</td>
<td>94.50</td>
</tr>
<tr>
<td>420,000,001</td>
<td>900,000,002</td>
<td>1,443.40</td>
<td>125.82</td>
</tr>
<tr>
<td># Processes</td>
<td># Context Switches</td>
<td>Execution Time (Secs.)</td>
<td>Memory Usage (GB)</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
<td>------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>7,000,001</td>
<td>15,000,002</td>
<td>7.53</td>
<td>1.79</td>
</tr>
<tr>
<td>10,000,001</td>
<td>22,500,002</td>
<td>16.03</td>
<td>3.02</td>
</tr>
<tr>
<td>14,000,001</td>
<td>30,000,002</td>
<td>25.86</td>
<td>4.10</td>
</tr>
<tr>
<td>210,000,001</td>
<td>450,000,002</td>
<td>642.80</td>
<td>63.91</td>
</tr>
<tr>
<td>350,000,001</td>
<td>750,000,002</td>
<td>1,235.12</td>
<td>94.50</td>
</tr>
<tr>
<td>420,000,001</td>
<td>900,000,002</td>
<td>1,443.40</td>
<td>125.82</td>
</tr>
<tr>
<td>476,000,001</td>
<td>1,020,000,002</td>
<td>1,800.79</td>
<td>126.11</td>
</tr>
<tr>
<td>480,900,001</td>
<td>1,030,500,002</td>
<td>1,801.40</td>
<td>126.20</td>
</tr>
</tbody>
</table>
ProcessJ code generator that produces Java source.

JVMCSP runtime implemented.

ASM bytecode instrumentation.

Performs better than CPA’14 and JCSP.

Can handle approximately half a billion processes in 128GB.
Future Work

- Multi-core Scheduler
- Network distribution
- Libraries
- Mobile processes
- Alts & claims are `busy waits’ (remain ready to run and cycle through the run queue)
- More back ends
Omar and Austin won gold in the UNLV College of Engineering Senior Design Competition for a CCSP code generator for ProcessJ.