Efficient Code Generation from SHIM Models

Stephen A. Edwards and Olivier Tardieu
Columbia University
**Definition**

**shim** \‘shim\  

1: a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

2: *Software/Hardware Integration Medium*, a model for describing hardware/software systems
The SHIM Model

Sequential processes
Unbuffered point-to-point communication channels exchange data tokens

Fixed topology
Asynchronous
Synchronous communication events
Delay-insensitive: sequence of data through any channel independent of scheduling policy (the Kahn principle)
“Kahn networks with rendezvous communication”
void main() {
    uint8 A, B, C;
    {
        // source: generate four values
        next A = 17;
        next A = 42;
        next A = 157;
        next A = 8;
    } par {
        // buf1: copy from input to output
        for (;;)
            next B = next A;
    } par {
        // buf2: copy, add 1 alternately
        for (;;) {
            next C = next B;
            next C = next B + 1;
        }
    } par {
        // sink
        for (;;)
            next C;
    }
}
Faking Concurrency in C

One function

```c
void run() {
    for (; ;) {
        switch (pc1) {
            case 0: block A
                pc1 = 1;
                break;
            case 1: block C
        }

        switch (pc2) {
            case 0: block B
                pc2 = 1;
                return;
            case 1: block D
        }
    }
}
```

Multiple Functions

```c
void run() {
    for (;;)
        run1(), run2();
}
```

```c
void run1() {
    static pc1;
    switch (pc1) {
        case 0:
            block A
            pc1 = 1;
        return;
        case 1:
            block C
    }
}
```

```c
void run2() {
    static pc2;
    switch (pc2) {
        case 0:
            block B
            pc2 = 1;
        return;
        case 1:
            block D
    }
}
```

Tail Recursion

```c
void run1a() {
    block A
    *(sp++) = run2a;
    (*(sp))(); return;
}
```

```c
void run2a() {
    block B
    *(sp++) = run1b;
    (*(sp))(); return;
}
```

```c
void run1b() {
    block C
    *(sp++) = run2b;
    (*(sp))(); return;
}
```

```c
void run2b() {
    block D
    (*(sp))(); return;
}
```
Faking Concurrency in C

One function

```c
void run() {
    for (;;) {
        switch (pc1) {
            case 0: block A
                pc1 = 1;
                break;
            case 1: block C
        }
        switch (pc2) {
            case 0: block B
                pc2 = 1;
                break;
            case 1: block D
        }
    }
}
```

Multiple Functions

```c
void run() {
    for (;;) {
        run1(), run2();
    }

    void run1() {
        static pc1;
        switch (pc1) {
            case 0: block A
                pc1 = 1;
                return;
            case 1: block C
        }
    }

    void run2() {
        static pc2;
        switch (pc2) {
            case 0: block B
                pc2 = 1;
                return;
            case 1: block D
        }
    }

    void run1a() {
        block A
        *(sp++) = run2a;
        (*(­­sp))(); return;
    }

    void run1b() {
        block C
        *(sp++) = run2b;
        (*(­­sp))(); return;
    }

    void run2a() {
        block B
        *(sp++) = run1b;
        (*(­­sp))(); return;
    }

    void run2b() {
        block D
        (*(­­sp))(); return;
    }
```
Faking Concurrency in C

One function

```c
void run() {
    for (;;) {
        switch (pc1) {
            case 0: block A
                pc1 = 1;
                break;
            case 1: block C
                break;
        }
    }
}
```

Multiple Functions

```c
void run() {
    for (;;) {
        run1(), run2();
    }
}

void run1() {
    static pc1;
    switch (pc1) {
        case 0: block A
            pc1 = 1;
            return;
        case 1: block C
            break;
    }
}

void run2() {
    static pc2;
    switch (pc2) {
        case 0: block B
            pc2 = 1;
            return;
        case 1: block D
            break;
    }
}
```

Tail Recursion

```c
void run1a() {
    block A
    *(sp++) = run2a;
    (*((--sp)))(); return;
}

void run1b() {
    block C
    *(sp++) = run2b;
    (*((--sp)))(); return;
}

void run2a() {
    block B
    *(sp++) = run1b;
    (*((--sp)))(); return;
}

void run2b() {
    block D
    (*((--sp)))(); return;
}
```
void source(int32 &C) {
    bool b = 0;
    for (int32 a = 0 ; a < 42 ; ) {
        if (b) {
            next C = a;
        } else {
            for (int32 d = 0 ; d < 10 ; ++d)
                a = a + 1;
        }
        b = ~b;
    }

    Extended basic blocks...
}
void (*stack[3])(void);
void (**sp)(void);

struct channel {
    void (*reader)(void);
    void (*writer)(void);
};

struct channel A_ch = { 0, 0 };;
struct channel B_ch = { 0, 0 };;
struct channel C_ch = { 0, 0 };;
unsigned char A, B, C;

struct {
    char blocked;
} source = { 0 };

struct {
    char blocked;
    unsigned char tmp;
} buf1 = { 0 };

struct {
    char blocked;
    unsigned char tmp;
} buf2 = { 0 };

struct {
    char blocked;
} sink = { 0 };

• Stack of pointers to runnable functions
• Each channel holds pointer to function (process) to resume after communication

Blocked flag for each process
• Each process broken into tail-recursive atomic functions
void source_0() {
    A = 17;
    if (buf1.blocked && A_ch.reader) {
        buf1.blocked = 0;
        *(sp++) = A_ch.reader;
        A_ch.reader = 0;
    }
    source.blocked = 1;
    A_ch.writer = source_1;
    (*(--sp))(); return;
}

void source_1() {
    /* ... */
}
void buf1_0() {
    if (source.blocked && A_ch.writer) { // If source is blocked,
        buf1_1(); return;                // “goto” buf1_1
    }
    buf1.blocked = 1;                  // Block us
    A_ch.reader = buf1_1;              // to continue below
    (*(­­sp))(); return;               // Run next process
}

void buf1_1() {                    // Read from the channel
    buf1.tmp = A;
    source.blocked = 0;              // Unblock the source,
    *(sp++) = A_ch.writer;           // schedule it, and
    A_ch.writer = 0;                 // clear the channel.
}
Compiling Processes Together

Build an automaton through abstract simulation

State signature:

- Running.blocked status of each process
- Blocked on reading.writing status of each channel

*Trick: does not include control or data state of each process*
Abstract Simulation

```c
{ // buf1
    for (;;) {
        next B = next A;
    }
} par {
    // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```
Abstract Simulation

```c
{ // buf1
    for (;;)
        next B = next A;
}
par {
    // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```

buf1 ready
buf2 blocked

buf1 PCs
buf2 PCs

A clear
C waiting for writer
B waiting for reader
Abstract Simulation

```c
{  // buf1
    for (;;) {
        next B = next A;
    }
} par {
    // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```

buf1 ready
buf2 blocked
A clear
B waiting for reader
C waiting for writer
Abstract Simulation

```c
{ // buf1
    for (;;) { // buf2
        next B = next A;
    } par {
        for (;;) {
            next C = next B;
            next C = next B + 1;
        }
    }
}
```

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs
A clear
C waiting for writer
B waiting for reader
Abstract Simulation

```c
{ // buf1
  for (;;)
    next B = next A;
} par {
  // buf2
  for (;;)
    next C = next B;
    next C = next B + 1;
}
```

- Buf1 ready
- Buf2 blocked
- Buf1 PCs
- Buf2 PCs
- A clear
- C waiting for writer
- B waiting for reader

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

{ // buf1
  1 for (;;)
  2 next B = 3 next A;
} par { // buf2
  4 for (;;) {
    5 next C = 6 next B;
    7 next C = 8 next B + 1;
  }
}

buf1 ready
buf2 blocked
A clear
C waiting for writer
B waiting for reader

buf1 PCs
buf2 PCs

{1, 2} {3}

{1} {4}
buf2

{1} {6}
buf1

{3} {6}
receive A

{3} {6}
buf1

{2} {6}
buf2

{2} {5}

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

```
{  // buf1
    for (; ;) {
        next B = next A;
    }
} par {
    // buf2
    for (; ;) {
        next C = next B;
        next C = next B + 1;
    }
}
```

- `buf1 ready`
- `buf2 blocked`
- `buf1 PCs`
- `buf2 PCs`
- `{1, 2} {3}`
- `A clear`
- `C waiting for writer`
- `B waiting for reader`
Abstract Simulation

```
{ // buf1
    for (;;) {
        next B = next A;
    }
} par {
    // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```
Abstract Simulation

```plaintext
{ // buf1
  for (;;)
    next B = next A;
} par {
  // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
}
```
Abstract Simulation

```c
{  // buf1
  for (;;)
  {  // buf2
    for (;;)
    {  // buf1
        next B = next A;
        next C = next B + 1;
    }
  }
}
```

buf1 ready
buf2 blocked
A clear
B waiting for reader
C waiting for writer
send C
receive A

buf1 PCs
buf2 PCs
{1, 2} {3}

{1} {4}
buf2

{1} {6}
buf1

{3} {6}
receive A

{3} {6}
buf1

{2} {5, 6}
buf2

{2} {5}
buf1

{3} {5}
buf1

{3} {5}

Abstract Simulation

```
{ // buf1
    for (;;)
        next B = next A;
} par {
    { // buf2
        for (;;) {
            next C = next B;
            next C = next B + 1;
        }
    }
}
```
Abstract Simulation

```c
{ // buf1
    for (;;)
        next B = next A;
    } par {
        // buf2
        for (;;) {
            next C = next B;
            next C = next B + 1;
        }
    }
}
```

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

```c
{ // buf1
  for (;;) {
    next B = next A;
  }
} par {
  // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
}
```
Abstract Simulation

```plaintext
{ // buf1
  ① for (;;) {
    ② next B = ③ next A;
  }
} par {
  // buf2
  ④ for (;;) {
    ⑤ next C = ⑥ next B;
    ⑦ next C = ⑧ next B + 1;
  }
}
```

buf1 ready

buf2 blocked

buf1 PCs

buf2 PCs

A clear

C waiting for writer

B waiting for reader

send C

receive A

receive A

send C

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

```c
for (;;) {
    next B = next A;
}

par {
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```
Abstract Simulation

```c
{ // buf1
    for (;;)
        next B = next A;
} par { // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```

- buf1 ready
- buf2 blocked
- buf1 PCs
- buf2 PCs
- A clear
- B waiting for reader
- C waiting for writer
- send C
- receive A
### Abstract Simulation

```c
{// buf1
  for (;;)
    next B = next A;
} par {
  // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
}
```

---

- **buf1 ready**
- **buf2 blocked**
- **buf1 PCs**
- **buf2 PCs**
- **buf1 PCs**
- **buf2 PCs**
- **buf1 PCs**
- **buf2 PCs**
- **buf1 PCs**
- **buf2 PCs**

A clear

B waiting for reader

C waiting for writer

---

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

```c
{ // buf1
  for (;;)
  next B = next A;
} par {
  // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
}
```
Abstract Simulation

```
for (;;)
next B = next A;
```

```
for (;;)
next C = next B;
next C = next B + 1;
```

Buffer 1 (buf1)

- A clear
- B waiting for reader
- C waiting for writer
- buf1 ready

Buffer 2 (buf2)

- buf2 blocked
- buf1 PCs
- buf2 PCs

```
buf1 PCs
{1, 2} {3}
```

```
buf2 PCs
{2} {5, 7}
```

```
send C
{3} {5, 7}
```

```
receive A
{3} {5, 7}
```

```
buf1
{2} {5, 6, 7}
```

```
buf2
{2} {5, 7}
```

```
buf2
{1} {6}
```

```
buf1
{1} {4}
```

```
buf2
{2} {5, 7}
```
Abstract Simulation

```c
{ // buf1
    for (;;)
        next B = next A;
} par {
    // buf2
    for (;;) {
        next C = next B;
        next C = next B + 1;
    }
}
```

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

```c
{ // buf1
    for (;;) { // buf1 PCs
        next B = next A;
    }
} par {
    // buf2
    for (;;) { // buf2 PCs
        for (;;) { // buf2 PCs
            next C = next B;
            next C = next B + 1;
        }
    }
}
```

Efficient Code Generation from SHIM Models – p. 11/16
Abstract Simulation

{ // buf1
  for (; ;) // buf1 PCs
    next B = next A;
  } par { // buf2
    for (; ;) { // buf2 PCs
      next C = next B;
      next C = next B + 1;
    }
  }
}

buf1 ready
buf2 blocked
A clear
B waiting for reader
C waiting for writer

buf1
send C
receive A

buf2
send C
receive A

{1} {4} buf2
{1} {6} buf1
{3} {6, 8} receive A
{3} {6, 8} receive A

{2} {5, 6, 7, 8} buf2
{2} {5, 7} buf1
{3} {5, 7} send C
{3} {5, 7} send C
Abstract Simulation

```
{ // buf1
  1 for (;;) {
  2   next B = 3 next A;
  } par {
    // buf2
    4 for (;;) {
  5     next C = 6 next B;
  6     next C = 8 next B + 1;
  }
}
```
## Benchmarks

<table>
<thead>
<tr>
<th>Example</th>
<th>Lines</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>Buffer2</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Buffer3</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>Buffer10</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>Esterel1</td>
<td>144</td>
<td>5</td>
</tr>
<tr>
<td>Esterel2</td>
<td>127</td>
<td>5</td>
</tr>
<tr>
<td>FIR5</td>
<td>78</td>
<td>19</td>
</tr>
<tr>
<td>FIR19</td>
<td>190</td>
<td>75</td>
</tr>
<tr>
<td>Example</td>
<td>Switch</td>
<td>Tail-Recursive</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size</td>
</tr>
<tr>
<td>Berkeley</td>
<td>860</td>
<td>1299</td>
</tr>
<tr>
<td>Buffer2</td>
<td>832</td>
<td>1345</td>
</tr>
<tr>
<td>Buffer3</td>
<td>996</td>
<td>1579</td>
</tr>
<tr>
<td>Buffer10</td>
<td>2128</td>
<td>3249</td>
</tr>
<tr>
<td>Esterel1</td>
<td>3640</td>
<td>5971</td>
</tr>
<tr>
<td>Esterel2</td>
<td>4620</td>
<td>7303</td>
</tr>
<tr>
<td>FIR5</td>
<td>4420</td>
<td>6863</td>
</tr>
<tr>
<td>FIR19</td>
<td>17052</td>
<td>25967</td>
</tr>
</tbody>
</table>
## Speedups vs. Switch

<table>
<thead>
<tr>
<th>Example</th>
<th>Tail-Recursive</th>
<th>Static (partial)</th>
<th>Static (full)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>$2.9 \times$</td>
<td>2.6</td>
<td>7.8</td>
</tr>
<tr>
<td>Buffer2</td>
<td>2.0</td>
<td>2.4</td>
<td>11</td>
</tr>
<tr>
<td>Buffer3</td>
<td>2.1</td>
<td>2.6</td>
<td>10</td>
</tr>
<tr>
<td>Buffer10</td>
<td>1.7</td>
<td>4.8</td>
<td>12</td>
</tr>
<tr>
<td>Esterel1</td>
<td>1.9</td>
<td>2.9</td>
<td>5.9</td>
</tr>
<tr>
<td>Esterel2</td>
<td>2.0</td>
<td>2.5</td>
<td>5.2</td>
</tr>
<tr>
<td>FIR5</td>
<td>0.92</td>
<td>4.8</td>
<td>7</td>
</tr>
<tr>
<td>FIR19</td>
<td>0.90</td>
<td>5.9</td>
<td>7.1</td>
</tr>
</tbody>
</table>
Conclusions

- The SHIM Model: Sequential processes communicating through rendezvous
- Tail-recursion: postponed `goto` in C
- Dynamic code maintains stack of function pointers to runnable processes
- Each process broken into one function per extended basic block
- Compiling processes together through abstract simulation
- Main trick: do not consider PC values part of state
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
Future Work

• Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

• Synthesis for multicore processors
  Compile together the processes on each core
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

- Synthesis for multicore processors
  Compile together the processes on each core

- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
- Convince world: deterministic concurrency is good