Efficient, Deterministic and Deadlock-free Concurrency

Nalini Vasudevan
Columbia University
Data Races

```c
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
}
bar(){
    int n;
    n = baz();
    x = x * n;
}
main() {
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
```

Diagram:
- **main** with:
  - `x = 2`
  - `x = 2`
  - `print x`

- **foo** with:
  - `m = qux()`
  - `x = x + m`

- **bar** with:
  - `n = baz()`
  - `x = x * n`
Eliminating Data Races

```c
int x;
foo()
{
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}
bar()
{
    int n;
    n = baz();
    lock(x);
    x = x * n;
    unlock(x);
}
main()
{
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
```

```
main

foo

x = 2

m = qux()

x = x + m

print x

bar

n = baz()

x = x * n

main
```
Eliminating Data Races

```c
int x;
void foo()
{
  int m;
  m = qux();
  lock(x);
  x = x + m;
  unlock(x);
}
void bar()
{
  int n;
  n = baz();
  lock(x);
  x = x * n;
  unlock(x);
}
main()
{
  x = 2;
  spawn foo();
  spawn bar();
  sync;
  print(x);
}
```

Diagram:

```
main

foo

x = 2
m = qux()
x = x + m

bar

n = baz()
x = x * n

print x

if m = n = 2
```
Eliminating Data Races

```c
int x;

foo() {
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}

bar() {
    int n;
    n = baz();
    lock(x);
    x = x * n;
    unlock(x);
}

main() {
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
```

```
if m = n = 2

x = (2 + 2) * 2 = 8
x = (2 * 2) + 2 = 6

Nondeterminism
```
Motivation

Parallel Computers
Library Support
Performance
Parallel Languages
Data Races
Non-Determinism
Hard-to-debug
Motivation

- Parallel Computers
- Library Support
- Parallel Languages
- Performance
- Data Races
- Non-Determinism
- Hard-to-debug
Determinism: The SHIM Model

- Stands for *Software Hardware Integration Medium*
- Race free, scheduling independent, concurrent model
- Blocking synchronous rendezvous communication
The SHIM Language

An imperative language with familiar C/Java-like syntax

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
```
Additional Constructs

\( stmt_1 \ par \ stmt_2 \) Run \( stmt_1 \) and \( stmt_2 \) concurrently

send \( var \) Send on channel \( var \)
recv \( var \) Receive on channel \( var \)
Communication

• Blocking: wait for all processes connected to c

```c
void f(chan int a) { // a is a copy of c
    a = 3; // change local copy
    recv a; // receive (wait for g)
    // a now 5
}
void g(chan int &b) { // b is an alias of c
    b = 5; // sets c
    send b; // send (wait for f)
    // b is 5
}
void main() {
    chan int c = 0;
    f(c); par g(c);
}
```
### Overview

**Determinism**
- What led to this thesis? [IPDPS Workshop 2008]
- How can we achieve determinism? [SES 2010]
- Is determinism efficient? [DATE 2008]
- Determinism: Language vs. Library? [IPDPS 2008]
- How do we solve the deadlock problem? [MEMOCODE 2008]

**Deadlock-freedom**
- Can we efficiently detect deadlocks? [EMSOFT 2009]
- Can we deterministically break deadlocks? [HOTPAR 2010]
- Can we reduce memory in deterministic programs? [MEMOCODE 2009]

**Efficiency**
- Can we optimize deterministic constructs? [CC 2009]
- Can we optimize locks? [PACT 2010]
- What are the limitations? [HIPG-SRS 2010]
- What next? [PLDI-FIT 2009]
Compiling to Quad-Core
Compiling to Quad-Core

- Intel quad-core machine
- Each task mapped to a pthread
- Example: JPEG decoder

<table>
<thead>
<tr>
<th>Cores</th>
<th>Tasks</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequential C</td>
<td>25s</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>16</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9.3</td>
<td>2.7</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>8.7</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8.2</td>
<td>3.05</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8.6</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Run on a 20 MB 21600 × 10800 image that expands to 668 MB.
Compiling to Cell
Compiling to Cell
Compiling to Cell [SAC 2009]

- Generated code for a heterogeneous multicore
- Computationally intensive tasks mapped on the SPEs
- Example: FFT
More Examples in SHIM

```c
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    }
    par {
        // Task q
        int c;
        recv a; // recv a
        recv b; // recv b
        c = a + b;
    }
}
```
The Problem

```c
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    } par {
        // Task q
        int c;
        recv b; // recv b
        recv a; // recv a
        c = a + b;
    }
}
```
The Problem

```c
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    }
    par {
        // Task q
        int c;
        recv b; // recv b
        recv a; // recv a
        c = a + b;
    }
}
```

Deadlock
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    } par {
        // Task q
        int c;
        recv b; // recv b
        recv a; // recv a
        c = a + b;
    }
}
Runtime Deadlock Detection [HotPar 2010]

• Generally, cycle detection algorithm is expensive
• SHIM’s semantics makes it simpler

A possible SHIM network

An impossible SHIM network
Static Deadlock Detection

Scheduling independence

Data sharing through rendezvous communication

Asynchronous parts are independent

Reduces state space

Just pick one schedule
## Deadlock Detection [MEMOCODE 2008]

- **Using NuSMV**

<table>
<thead>
<tr>
<th>Example</th>
<th>Lines</th>
<th>Channels</th>
<th>Tasks</th>
<th>Result</th>
<th>Runtime</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source-Sink</td>
<td>35</td>
<td>2</td>
<td>11</td>
<td>No Deadlock</td>
<td>0.2 s</td>
<td>3.9 MB</td>
</tr>
<tr>
<td>Pipeline</td>
<td>30</td>
<td>7</td>
<td>13</td>
<td>No Deadlock</td>
<td>0.1</td>
<td>2.0</td>
</tr>
<tr>
<td>Prime Sieve</td>
<td>35</td>
<td>51</td>
<td>45</td>
<td>No Deadlock</td>
<td>1.7</td>
<td>25.4</td>
</tr>
<tr>
<td>Berkeley</td>
<td>40</td>
<td>3</td>
<td>11</td>
<td>No Deadlock</td>
<td>0.2</td>
<td>7.2</td>
</tr>
<tr>
<td>FIR Filter</td>
<td>100</td>
<td>28</td>
<td>28</td>
<td>No Deadlock</td>
<td>0.4</td>
<td>13.4</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>130</td>
<td>65</td>
<td>167</td>
<td>No Deadlock</td>
<td>8.5</td>
<td>63.8</td>
</tr>
<tr>
<td>Framebuffer</td>
<td>220</td>
<td>11</td>
<td>12</td>
<td>No Deadlock</td>
<td>1.7</td>
<td>11.6</td>
</tr>
<tr>
<td>JPEG Decoder</td>
<td>1025</td>
<td>7</td>
<td>15</td>
<td>No Deadlock</td>
<td>0.9</td>
<td>85.6</td>
</tr>
</tbody>
</table>
Deadlocks in SHIM

- Why SHIM? No data races.
- Deadlocks in SHIM are deterministic (always reproducible).
- SHIM’s philosophy: It prefers deadlocks to races.
• Compositional deadlock detection
More Verification [CC 2009, MEMOCODE 2009, TCAD 2010]

- Buffer sharing
  - Can two channels be active simultaneously?
- Analysis of clocks in X10
  - E.g.: A clock is used by just two tasks.
  - Specialization based on analysis
More Verification [CC 2009, MEMOCODE 2009, TCAD 2010]

- Buffer sharing
  - Can two channels be active simultaneously?
- Analysis of clocks in X10
  - E.g.: A clock is used by just two tasks.
  - Specialization based on analysis

Determinism simplifies verification
SHIM as a Library [IPDPS 2009]

- Implemented in Haskell
- APIs that mimic `par`, `send` and `recv`
- Programmer’s job to use the library correctly
- Example: RGB Histogram

![Graph showing execution time vs number of processors for SHIM Concurrent Library compared to ideal.]
Comparison with other models

- Esterel: Deterministic
- StreamIt: Deterministic
- CSP: Nondeterministic
- Cilk: Nondeterministic
- X10: Nondeterministic
The $D^2C$ Model

Limitations of SHIM

- Single writes
- Susceptible to deadlocks

Addressed by the $D^2C$ model

- Stands for *Deterministic Deadlock-free Concurrent Model*
The $D^2C$ Model [HIPC-SRS 2010]

```c
void f(shared int &a) {
    /* a is 0 */
    a = 3;
    /* a is 3 , x is still 0 */
    next; /* Apply reduction operator */
    /* a is now 8, x is 8 */
}

void g(shared int &b) {
    /* b is 0 */
    b = 5;
    /* b is 5, x is still 0 */
    next; /* Apply reduction operator */
    /* b is now 8, x is 8 */
}

void h (shared int &c) {
    /* c is 0 , x is still 0 */
    next;
    /* c is now 8, x is 8 */
}

main() {
    shared int (+) x = 0;
    /* If there are multiple writers, reduce using the + reduction operator */
    spawn f(x);
    spawn g(x);
    spawn h(x);
    sync;
    /* x is 8 */
}
```
The $D^2C$ Model

- Histogram example

```c
void histogram(int a[], int n) {
    int b[10];
    for (int i = 0; i < n; i++) {
        spawn {
            int index = a[i];
            b[index]++;
        }
    }
    sync;
}
```
The $D^2C$ Model

- Histogram example

```c
void histogram(int a[], int n) {
    shared int (+) b[10];
    for (int i = 0; i < n; i++) {
        spawn {
            int index = a[i];
            b[index] = 1;
            next;
        }
    }
    sync;
}
```
The $D^2C$ Model

- Determinism ✓
- Deadlock Freedom ✓
- Efficiency ?
The $D^2C$ model

![Relative Speed Graph](image-url)
Future Work  [PLDI’09 Fun Ideas and Thoughts]
The Example

```c
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
}
bar(){
    int n;
    n = baz();
    x = x * n;
}
main() {
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
```

```
x = 2
m = qux()
x = x + m
n = baz()
x = x * n
print x
```
int x;
foo()
    int m;
    m = qux();
    x = x + m;
    sync(x);
}
bar()
    int n;
    n = baz();
    sync(x);
    x = x * n;
}
main() {
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
The Determinizing Compiler’s Role

```c
int x;

foo()
{
    int m;
    m = qux();
    x = x + m;
    sync(x);
}

bar()
{
    int n;
    n = baz();
    sync(x);
    x = x * n;
}

main()
{
    x = 2;
    spawn foo();
    spawn bar();
    sync;
    print(x);
}
```

if m = n = 2
x = (2 + 2) * 2 = 8
Always
The Ultimate Goal

- Determinism ✓
- Deadlock Freedom ✓
- Efficiency ✓