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);
}
```
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);
}
```
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
```

Efficient, Deterministic Concurrency – p. 3/31
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
```
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
```

Non-determinism
Motivation

- Parallel Computers
- Library Support
- Parallel Languages
- Performance
- 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

\[ stmt_1 \ par \ stmt_2 \quad \text{Run } stmt_1 \text{ and } stmt_2 \text{ concurrently} \]

\[ \text{send } \var \quad \text{Send on channel } \var \]

\[ \text{recv } \var \quad \text{Receive on channel } \var \]
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 now 5
}
void main() {
    chan int c = 0;
    f(c); par g(c);
}
```
## Overview

<table>
<thead>
<tr>
<th>How efficient is determinism?</th>
<th>Compiling SHIM to Shared Memory Multicores</th>
<th>DATE 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiling SHIM to Heterogeneous Multicores</td>
<td>SAC 2009</td>
</tr>
<tr>
<td></td>
<td>Compilation techniques for SHIM</td>
<td>SES 2010</td>
</tr>
<tr>
<td></td>
<td>Simple and Fast Biased Locks</td>
<td>PACT 2010</td>
</tr>
<tr>
<td></td>
<td>A SHIM-like Library in Haskell</td>
<td>IPDPS 2008</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Does determinism guarantee deadlock-freedom?</th>
<th>Static Deadlock Detection for SHIM</th>
<th>MEMOCODE 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compositional Deadlock Detection</td>
<td>EMSOFT 2009</td>
</tr>
<tr>
<td></td>
<td>Run-time deadlock detection in SHIM</td>
<td>HotPar 2010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Does determinism simplify verification?</th>
<th>Buffer Sharing in SHIM Programs</th>
<th>MEMOCODE 2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Buffer Sharing in Rendezvous Programs</td>
<td>TCAD 2010</td>
</tr>
<tr>
<td></td>
<td>Analysis and Specialization of Clocks in X10</td>
<td>CC 2009</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What next?</th>
<th>Overview and Shortcomings of SHIM</th>
<th>IPDPS Proc. 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A Determinizing Compiler</td>
<td>PLDI WACI 2009</td>
</tr>
</tbody>
</table>
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</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 [SAC 2009]

- Generated Code for a Heterogeneous Multicore
- Computationally intensive tasks mapped on the SPUs
- Example: FFT

![Graph showing execution time vs number of SPE tasks]

- Observed
- Ideal
```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;
    }
}
```
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

- 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.
More Verification [EMSOFT 2009, MEMOCODE 2009, TCAD 2010]

- Compositional Deadlock Detection

![Graph showing verification time vs. number of IDCT processes]

- Buffer Sharing
  - Can two channels be active simultaneously?
More Verification [CC 2009]

- Analysis of clocks in X10
  - E.g.: A clock is used by just two tasks.
  - Specialization based on analysis
More Verification [CC 2009]

- 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: Systolic Filter

![Graph showing execution time vs number of processors]
void f(shared int &a) {
    /* a is 1 */
    a = 3;
    /* a is 3 , x is still 1 */
    next; /* Apply reduction operator */
    /* a is now 8, x is 8 */
}

void g(shared int &b) {
    /* b is 1 */
    b = 5;
    /* b is 5, x is still 1 */
    next; /* Apply reduction operator */
    /* b is now 8, x is 8 */
}

void h (shared int &c) {
    /* c is 1 , x is still 1 */
    next;
    /* c is now 8, x is 8 */
}

main() {
    shared int (+) x = 1;
    /* If there are multiple writers, reduce using the + reduction operator */
    spawn f(x);
    spawn g(x);
    spawn h(x);
    sync;
    /* x is 8 */
}
Deterministic, Deadlock-free 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]++;
        }
    }
    print (b);
}
```
Deterministic, Deadlock-free 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;
        }
    }
    print (b);
}
```
Deterministic, Deadlock-free Model

Determinism ✓

Deadlock Freedom ✓ Efficiency ?
Deterministic, Deadlock-free Model

The diagram represents the relative speed of various applications when deterministic and deadlock-free concurrency is applied. The x-axis lists different applications such as AllReduceParallel, Pipeline, Convolve, NQueensPar, MontyPiParallel, KMeansScalar, Histogram, MergeSort, Stream, Prefix, UTS, IDEA, Stencil, SOR, Series, RayTrace, LUFact, and SparseMatMul. The y-axis shows the Relative Speed range.

The chart compares the Determinized version (solid line) with the Original version (dotted line) for each application. The Determinized versions show improved performance in many cases, indicating the benefits of deterministic concurrency for these applications.
Future Work [PLDI’09 Fun Ideas and Thoughts]

Parallel Computers
Library Support
Parallel Languages
Performance

A Determinizing Compiler!
```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);
}
```

```
main
    x = 2

foo
    m = qux()
    x = x + m

bar
    n = baz()
    x = x * n

print x
```

Efficient, Deterministic Concurrency – p. 29/31
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);
}
```

Efficient, Deterministic Concurrency – p. 30/31
The Determinizing Compiler’s Role

```
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 ✓