CSCI-GA.3033-017

Special Topics:

Multicore Programming

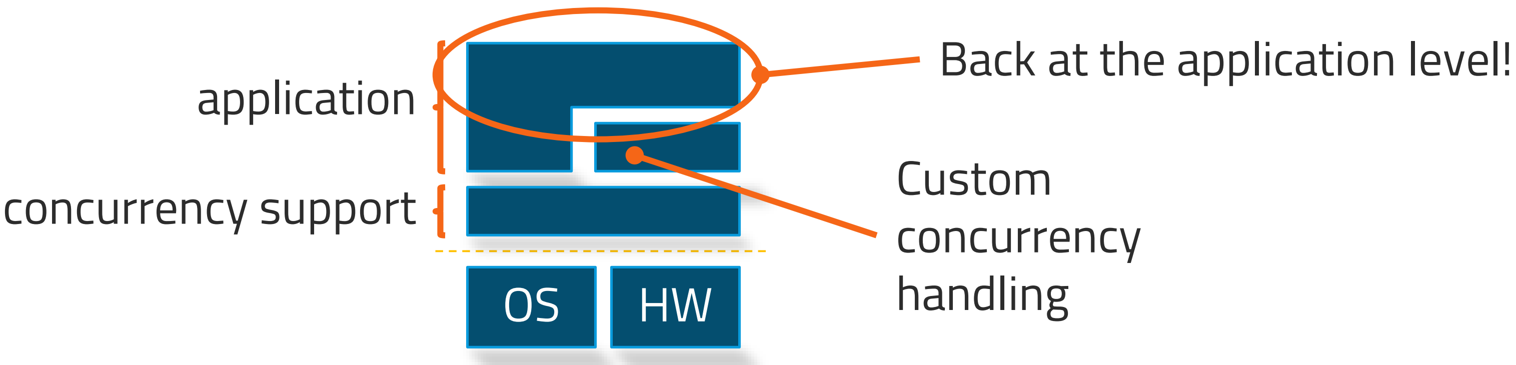
Lecture 9

Multicore Correctness

Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me

Context

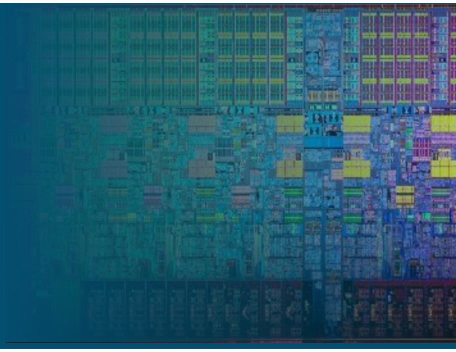

application
concurrency support
OS
HW
Back at the application level!
Custom
concurrency
handling

Outline

- Taxonomy of Real-World Bugs
- Detecting and Reproducing Bugs
- Advanced Thread Interleaving
- Eliminating Non-Determinism?

A Taxonomy of Real-World Bugs

Types of Bugs



- Race conditions
- Deadlocks
- Atomicity Violations
- Ordering Violations
- Group Coordination Violations
- Timing Dependencies

The Problem with Threads

"[Threads] discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programming becomes one of pruning non-determinism"

-- Edward A. Lee, "The Problem with Threads", 2006

A Simple Race Condition

- Accessing a shared variable outside a lock

```
Thread 1
{
    std::scoped_lock(m);
    i++;
}
```

```
Thread 2
    //…
    i--;
    //…
```

A Simple Deadlock

- Inconsistent lock ordering

```
Thread 1

m1.lock();
m2.lock();
i++;
m2.unlock();
m1.unlock();
```

```
Thread 2

m2.lock();
m1.lock();
i--;
m1.unlock();
m2.unlock();
```

More Subtle Bugs

- Some thread interleavings...
  1. Break implicit atomicity assumptions
  2. Break implicit order assumptions
  3. Break time interval guarantees
- Not every problem can be fixed with locks

Atomicity Violation

- Find more examples in “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)

```
Thread 1

if (thr->proc_info) {
    fputs(thr->proc_info);
}
```

```
Thread 2

//…
thr->proc_info = nullptr;
//…
```

Atomicity Violation

- Find more examples in “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)

```
Thread 1

if (thr->proc_info) {
    fputs(thr->proc_info);
}
```

```
Thread 2

//…
thr->proc_info = nullptr;
//…
```

- Assumes 3 will not be interposed between 1 and 2

Order Violation

- Implicit programmer expectations

```
Thread 1

void init() {
    //…
    thread_ = create(&main_);
    //…
}
```

```
Thread 2

void main_() {
    state_ = thread_->state;
    //…
}
```

Order Violation

- Implicit programmer expectations

```
Thread 1

void init() {
    //…
    thread_ = create(&main_);
    //…
}
```

```
Thread 2

void main_() {
    state_ = thread_->state;
    //…
}
```

- What if `create()` doesn't return until `main_` runs for a while?

Another Order Violation

- Subtle write-write race

```
Thread 1

int readWriteProc() {
    //…
    ReadAsync(&p);
    io_pending = true;
    while(io_pending) {
        // Wait for done
        //…
    }
}
```

```
Thread 2

void doneWaiting();
    // Callback called from
    // ReadAsync()
    io_pending = false;
}
```

Another Order Violation

- Subtle write-write race

```
Thread 1

int readWriteProc() {
    //…
    ReadAsync(&p);
1   io_pending = true;
    while(io_pending) {
        // Wait for done
        //…   2
    }
}
```

```
Thread 2

void doneWaiting();
    // Callback called from
    // ReadAsync()
3   io_pending = false;
}
```

- Programmer assumes 1 must run before Thread 2 manages to get to 3

Group Coordination Bugs

```
Thread 1

void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```

```
Thread 2

void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```

Group Coordination Bugs

```
Thread 1

void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```

```
Thread 2

void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```

- Imagine ordering of 1, 2, 3, 4: both Thread 1 and Thread 2 will try to free the resource
- Type of race condition

Timing Dependencies

- Many threads may cause timeout to spuriously trigger

```
Thread i

//…
rw_lock(m);
//…
```

```
Thread i

//…
{
    try_lock_for(m);
}

// timeout
```

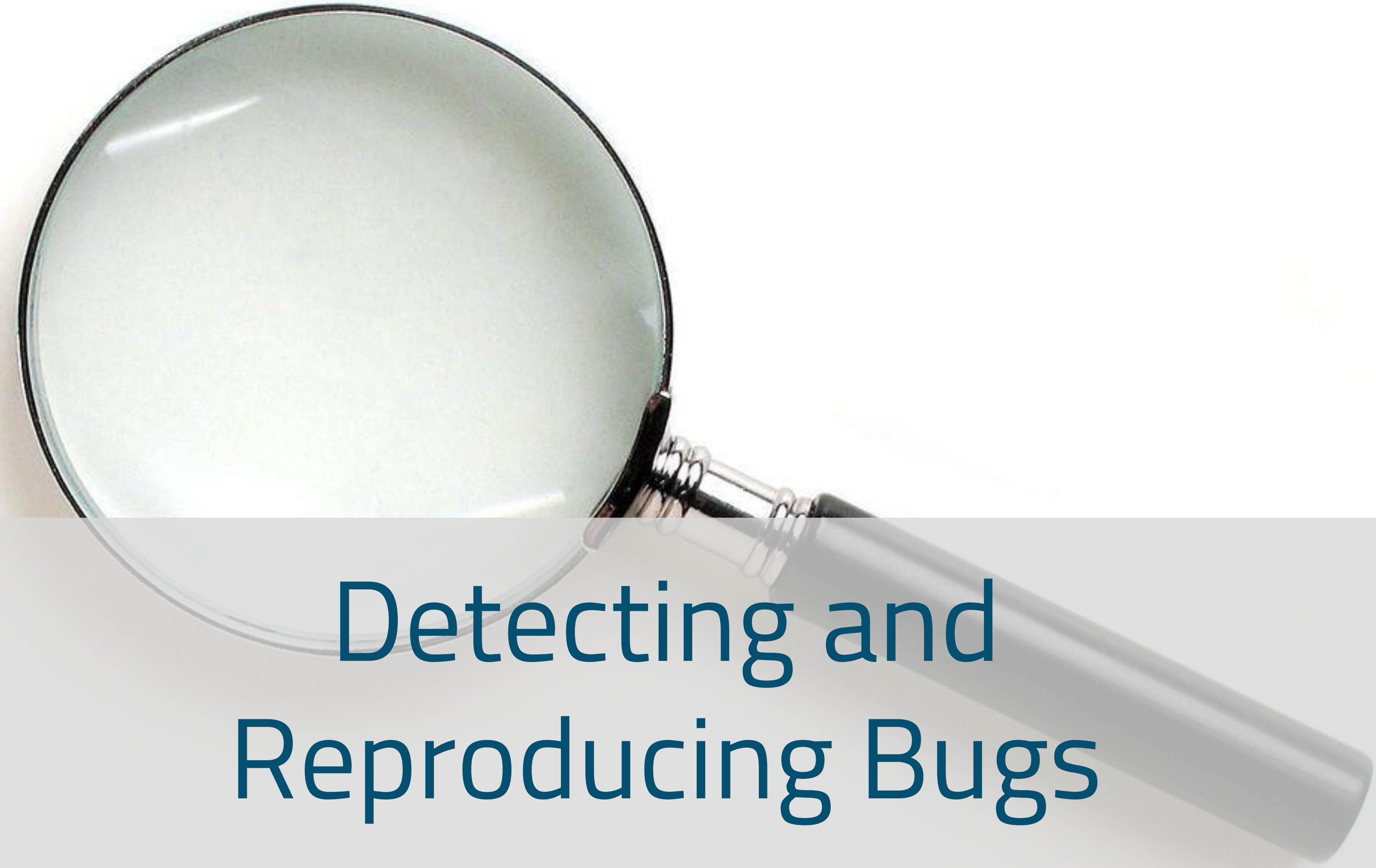
Timing Dependencies

- Many threads may cause timeout to spuriously trigger

```
Thread i

//…
rw_lock(m);
//…
```

```
Thread i

//…
rw_lock(m);
//…
```

```
Thread i

//…
{
    try_lock_for(m);
}

// timeout
```



Detecting and Reproducing Bugs

Finding a Bug-Fix Strategy

- First, we must detect bugs. How?

“Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques other than adding/changing locks. Programmers need to consider correctness, performance and other issues to decide the most appropriate fix strategy.”

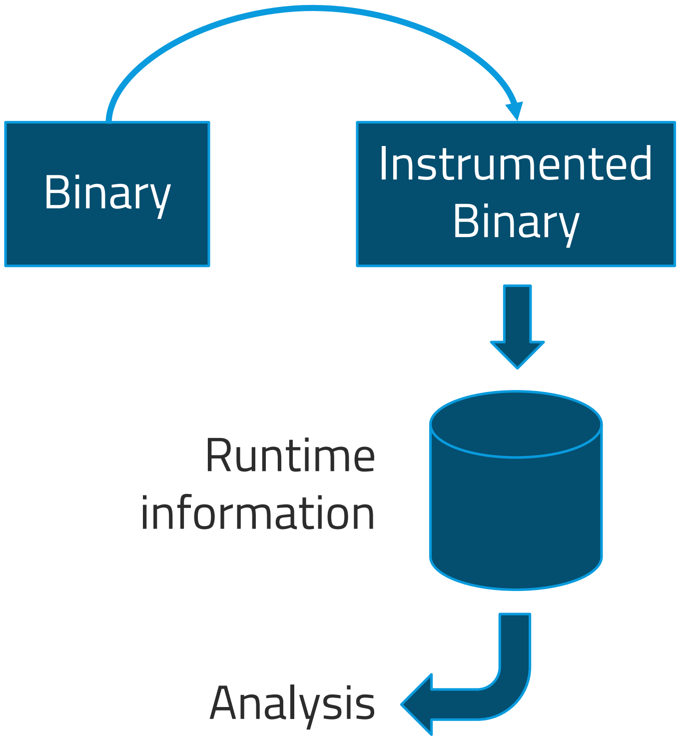
-- “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)

Software-Based Detection

- Static analysis: inspect during compilation
- Dynamic analysis: inspect during runtime
  - Catches more than static checking
  - Shared variables may not always be static (e.g.: pointers)
  - Subtleties of shared variable protection that cannot be captured by static analysis
    - Anything involving non-deterministic input
- Binary instrumentation
- Dynamic binary translation

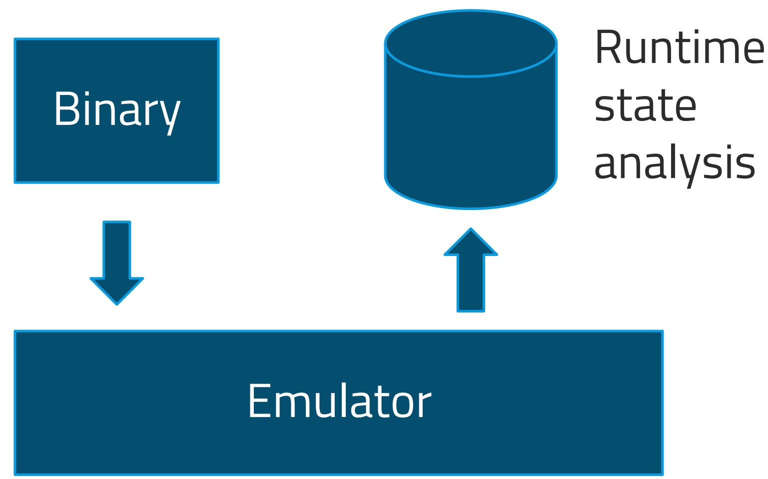
Software-Based Detection

- Binary Instrumentation



- Eg: helgrind

- Dynamic binary translation



“Happens-Before” Graphs

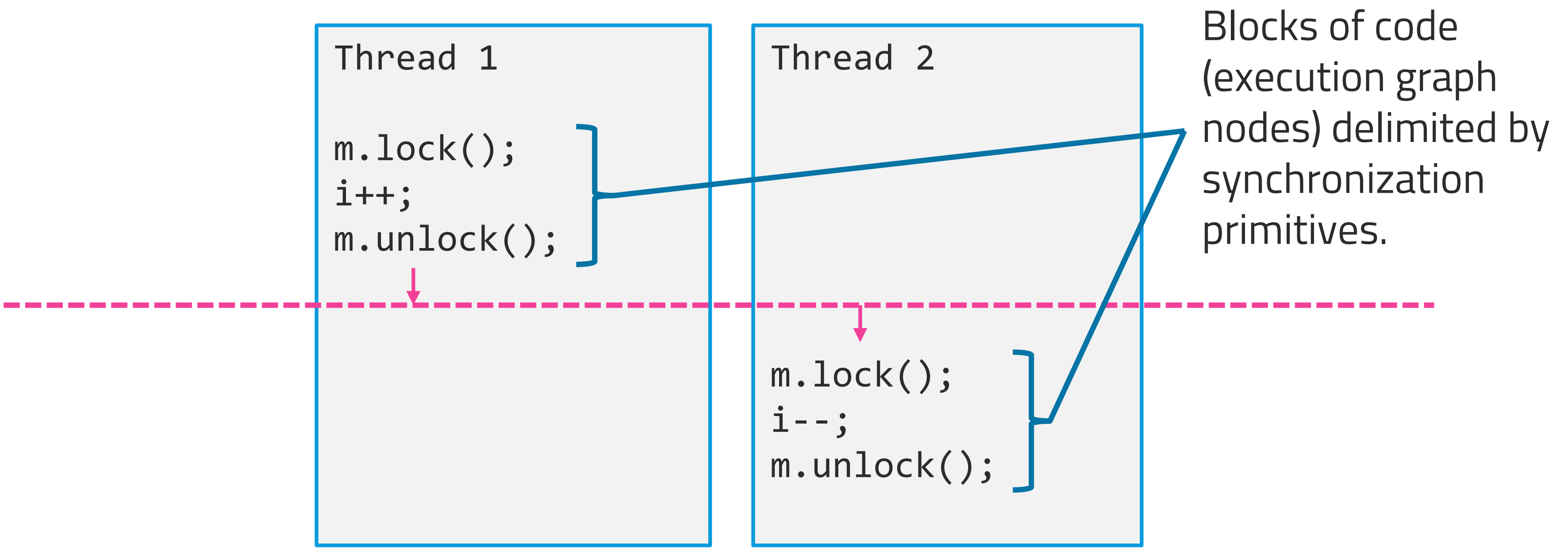
- Basic concept also used in software analysis

```
Thread 1

m.lock();
i++;
m.unlock();
```

```
Thread 2

m.lock();
i--;
m.unlock();
```

“Happens-Before” Graphs

- Basic concept also used in software analysis



Race Detection

- `i` accessed by both nodes, but they do not have a "happens before" relationship

```
Thread 1

{
    scoped_lock(m);
    i++;
}
```

```
Thread 2

//…
i--;
//…
```

- Other examples: "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", Savage et al., 1997.

A More Subtle Hazard

Thread 1

```
count++;
m.lock();
i++;
m.unlock();
```

Thread 2

```
m.lock();
i--;
count++;
m.unlock();
```

Locksets

- A record for each variable read/written
  - Performed under which lock(s)?
  - Performed in which block?
- Managing lockset size

'count' Lockset

'i' Lockset

```
Thread 1

count++;
m.lock();
i++;
m.unlock();
```

```
Thread 2

m.lock();
i--;
count++;
m.unlock();
```

'count' Lockset

'i' Lockset

Further Detection Needed

- How to know 3 is at fault for a crash at 2?

```
Thread 1

1 if (thr->proc_info) {
2   fputs(thr->proc_info);
}
```

```
Thread 2

//…
3 thr->proc_info = nullptr;
//…
```

- Locksets can be refined to capture situations that are races, but are harmless
- Some bugs involve atomicity violations or order violations, not incorrect lock use

Record/Replay

- Trace backwards from where bug manifested to track root cause (eg, what set `thd->proc_info` to nullptr)
- Requires storing huge amounts of state

A Software Approach

- Using threading API (e.g., pthreads) to explore potential hazards

```
Thread 1

void init() {
    //…
    thread_ = create(&main_);
    //…
}
```

Alternative 2

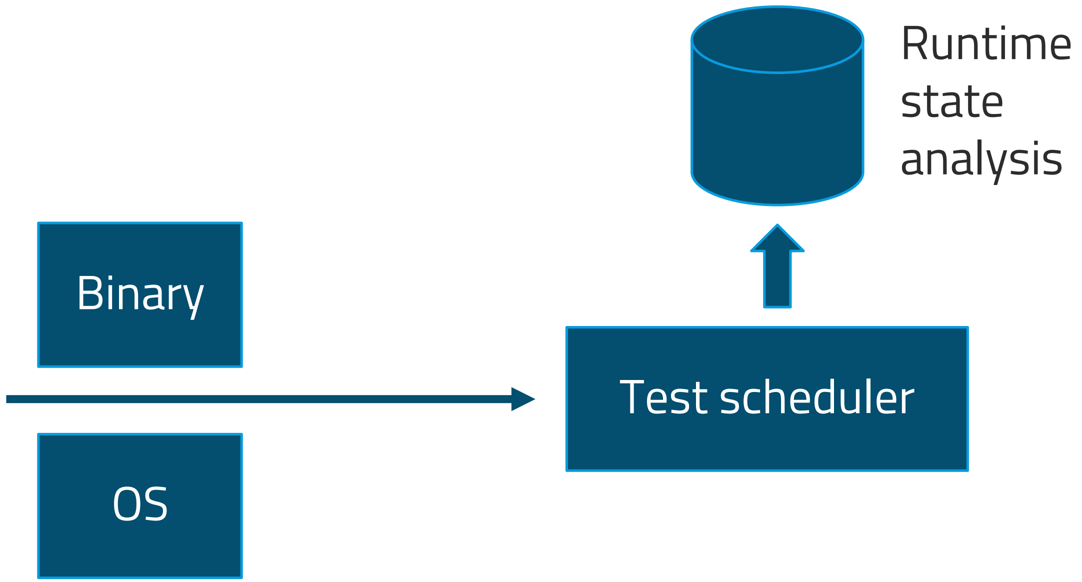
Alternative 1

```
Thread 2

void main_() {
    state_ = thread_->state;
    //…
}
```

Descheduling point

Scheduler-Based Framework

- See Musuvathi et al, “Finding and Reproducing Heisenbugs”, 2008.



Taming State Explosion

- A program with ***n*** threads that execute ***k*** atomic steps has n^k possible interleavings
- If we reduce the number of preemptions, ***k*** decreases sharply.
  - Tradeoff of coverage and analysis time
- Empirical evidence: few threads necessary to expose atomic and order violations.

Simplifying Parallel Programming

- Two of many efforts
  - Hardware Transactional Memory (future lecture): Research dates back to 1993
  - Deterministic execution: guaranteeing deterministic semantics in parallel software
- More efforts: to be mentioned in your presentations!

Deterministic Execution

- Recently, arguments for exploring deterministic ways to express parallelism
  - “Parallel Programming Must Be Deterministic By Default”, Bocchino et al, 2009.
- Language itself would have constructs for compile-time enforcements of sharing constraints
- Ongoing effort, with many recent publications

Conclusion

- At some level (ideally as low as possible), threads must exist
  - Hardware primitive: multiple cores
- Continuous, wide effort to expose different model to higher-level programmer
  - Programmer still wants parallel view of the world
- Main challenge: Taming non-determinism inherent in pure thread model

References

- Edward A. Lee, “The Problem with Threads”, 2006
- Lu, Park, Seo, Zhou, “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics”, 2009
- Savage et al, “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”, 1997
- Musuvathi et al, “Finding and Reproducing Heisenbugs”, 2008.
- Bocchino Jr. et al, “Parallel Programming Must Be Deterministic By Default”, 2009.
- Helgrind: http://valgrind.org/docs/manual/hg-manual.html