A Review/Tour of Concurrency & Parallelism

CMPT 886
Automated Software Analysis & Security
Nick Sumner
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism

- As processor speeds have slowed increasing, much focus has been placed on the last two
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrenty & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

![Diagram](sequential-diagram.png)
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

- The best possible running time is determined by the *critical path* or *span* of dependent tasks through the program.
Using Parallelism

• Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

• The best possible running time is determined by the **critical path** or **span** of dependent tasks through the program.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the best possible running time is determined by the critical path or span of dependent tasks through the program.

- This is too optimistic! Why?
Using Parallelism

- There are often more tasks than compute resources
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

Given $p$ processors, \[ \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty \]
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits
    
    \[
    \text{Given } p \text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
    \]

- Identifying good opportunities for effective parallelism is open to research
Using Parallelism

• There are often more tasks than compute resources
  – Brent’s Theorem describes the time accounting for limits

\[ \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty \]

• Identifying good opportunities for effective parallelism is open to research
  – Profiling for tasks to extract
  – Understanding the effect of speeding specific tasks
  – ...
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others

- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - ...

Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  
  97% of real world concurrency bugs

[Lu, ASPLOS 2008]
Data Races

- A data race occurs when:
Data Races

- A data race occurs when:
  1) two threads access the same location
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write (WAW, WAR, RAR)
A data race occurs when:
1) two threads access the same location
2) the accesses are *logically simultaneous*
3) at least one access is a write (WAW, WAR, RAR)
Data Races

x++

TMP = x
TMP = TMP + 1
X = TMP

T1  T2

tmp₁ = x
tmp₁ = tmp₁ + 1
x = tmp₁

tmp₂ = x
tmp₂ = tmp₂ + 1
x = tmp₂
Data Races

T1  T2

x++

tmp = x
tmp = tmp+1
x = tmp

T1  T2

tmp1 = x
tmp1 = tmp1+1
x = tmp1

tmp2 = x
tmp2 = tmp2+1
x = tmp2

tmp1 = x
tmp1 = tmp1+1

x = tmp1

tmp2 = x
tmp2 = tmp2+1
x = tmp2

x = tmp
Data Races

Synchronization discipline prevents data races.
“Benign” Data Races

• Sometimes a developer will make use of a data race
  – Avoid expensive synchronization
  – The race looks “benign” or harmless
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is really okay
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is really okay
  - A memory model determines what values may be read by a given memory access, esp. w.r.t. previous writes

[CACM 2010, PLDI 2018]
“Benign” Data Races

```c
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}

tmp = data;

[Boehm, Hotpar 2011]
```
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```c
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;

[Boehm, Hotpar 2011]
```
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```c
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```java
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;

[Boehm, Hotpar 2011]`
“Benign” Data Races

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}

[Boehm, Hotpar 2011]
"Benign" Data Races

Data race freedom allows extra reads.

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (counter > localMax) {
    handler(update);
}

[Boehm, Hotpar 2011]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

\[ c = a + 10 \]
\[ \ldots \]
\[ b = a + 10 \]

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

\[
\begin{align*}
c & = a + 10 \\
... \\
b & = a + 10 \\
c & = 1
\end{align*}
\]

[Dolan, PLDI 2018]
"Benign" Data Races

- Races can jump forward and backward in time

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can jump forward and backward in time

This can happen in Java when flag is volatile & b is a complex reference

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can jump forward and backward in time

This can happen in Java when flag is volatile & b is a complex reference

[Dolan, PLDI 2018]
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*).
  - defined behavior occurs when writes & reads are ordered.
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: simultaneously)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps per variable order
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: simultaneously)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps per variable order

- Happens-Before ordering of a specific execution can be tracked to identify bugs
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()  

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()
Happens-Before Ordering

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
x = tmp  
unlock()  
lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()
Happens-Before Ordering

T1  T2

lock()

tmp = x

tmp = tmp + 1

x = tmp

unlock()
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

T1  T2

tmp = x
tmp = tmp+1
x = tmp

No ordering!
Simultaneous!
Race detected!

tmp = x
tmp = tmp+1
x = tmp
Happens-Before Ordering

• Note, this only detects races in the current execution!
  – *Sound predictive* data race detection can extend it across other executions [PLDI 2017/2018]
Happens-Before Ordering

- Note, this only detects races in the current execution!
  - *Sound predictive* data race detection can extend it across other executions [PLDI 2017/2018]

- Requires careful tracking of dependences
  - Careful construction of logical time using *vector clocks* [JVM 2001, PLDI 2009]
Logical Time & Vector Clocks

Clocks

Shadow Memory

T1 \rightarrow [0, 0]
T2 \rightarrow [0, 0]

x \rightarrow R[0, 0]
W[0, 0]
y \rightarrow R[0, 0]
W[0, 0]
Logical Time & Vector Clocks

T1   T2

**Clocks**

T1\rightarrow [1, 0]
T2\rightarrow [0, 0]

**Shadow Memory**

x\rightarrow R[0, 0]
W[0, 0]
y\rightarrow R[0, 0]
W[0, 0]
Logical Time & Vector Clocks

T1  T2

Clocks

T1→[1,0]
T2→[0,0]

Shadow Memory

x→ R[0,0]
W[1,0]
y→ R[0,0]
W[0,0]

lock(a)
write x
unlock(a)

write x

lock(a)
read x
unlock(a)

write y
Logical Time & Vector Clocks

T1  T2

lock(a)
write x
unlock(a)

write x

lock(a)
read x
unlock(a)

write y

Clocks

T1↦[2,0]
T2↦[0,0]

Shadow Memory

x↦R[0,0]
W[1,0]
y↦R[0,0]
W[0,0]
Logical Time & Vector Clocks

T1  T2

lock(a)  write x  unlock(a)  write y

Clocks

T1↦[2,0]  T2↦[2,1]

Shadow Memory

x↦R[0,0]  W[1,0]
y↦R[0,0]  W[0,0]
Logical Time & Vector Clocks

T1  T2

lock(a)  write x  unlock(a)

write x

Clocks

lock(a)  read x  unlock(a)

write y

write y

Shadow Memory

x$\mapsto$ $R[2,1]$

T1$\mapsto$ $[2,0]$

T2$\mapsto$ $[2,1]$

y$\mapsto$ $R[0,0]$

W[1,0]

W[0,0]
Logical Time & Vector Clocks

T1 \ T2

lock(a)
write x
unlock(a)

write x

lock(a)
read x
unlock(a)

write y

Clocks

\[ T1 \mapsto [2,0] \]
\[ T2 \mapsto [2,2] \]

Shadow Memory

\[ x \mapsto R[2,1] \]
\[ W[1,0] \]
\[ y \mapsto R[0,0] \]
\[ W[0,0] \]
Logical Time & Vector Clocks

```
lock(a)
write x
unlock(a)

lock(a)
read x
unlock(a)

T1 ↦ [2, 0]
T2 ↦ [2, 2]
```

Clocks

Shadow Memory

```
write x
write y

x ↦ R[2, 1]
W[2, 0]
y ↦ R[0, 0]
W[0, 0]
```
Logical Time & Vector Clocks

T1 \[\rightarrow [2, 0] \]
T2 \[\rightarrow [2, 2] \]

Clocks

Shadow Memory

C(R,X) \not\subseteq C(W,X) \rightarrow \text{race!} 
(simplified for this case)
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

\[
\begin{align*}
\text{T1:} & \quad \text{lock}(a); \quad \text{lock}(b) \\
& \quad \text{read y} \\
& \quad \text{write x} \\
& \quad \text{unlock}(b) \\
& \quad \text{write x} \\
& \quad \text{unlock}(a) \\
\text{T2:} & \quad \text{lock}(b) \\
& \quad \text{unlock}(b) \\
\end{align*}
\]

Shadow Memory:
\[
\begin{align*}
y & \mapsto a, b \\
x & \mapsto a, b
\end{align*}
\]
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Shadow Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(a); lock(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write x</td>
<td></td>
<td>y↦a, b</td>
</tr>
<tr>
<td>unlock(a)</td>
<td></td>
<td>x↦a</td>
</tr>
</tbody>
</table>

lock(b)
write x
unlock(b)
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

```
lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)
```

```
lock(b)
write x
unlock(b)
```

Shadow Memory

```
y↦a, b
x↦{}
```
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking.
- We can dynamically track the locks guarding an address!

```
lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)
```

```
lock(b)
write x
unlock(b)
```

**Shadow Memory**

- \( y \mapsto a, b \)
- \( x \mapsto \{\} \)

**Note:** Both \( x \) and \( y \) are always protected by locks. \( x \) still races.
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs

- Tends to have many false positives
Order Violations

- Some accesses are wrongly assumed to occur before others

x = new Data

- wait/notify or condition variables can fix these

T1  T2

x->datum
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```plaintext
tmp = x
tmp = tmp+1
x = tmp

lock()
unlock()

tmp = x
unlock()
tmp = tmp+1
lock()
x = tmp
unlock()

vs

No race, similar effect!
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

What do we really want?

```
lock()
tmp = x
unlock()

lock()
tmp = tmp+1
unlock()
x = tmp
```

```
lock()
tmp = x
unlock()

lock()
tmp = tmp+1
unlock()
x = tmp
unlock()
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.
- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is \textit{atomic} if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
  - This also takes care of data races
  - Similar to notions from databases (serializability & linearizability)
Atomicity Violations

- How can we find atomicity violations?
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

How can we find atomicity violations (or correctness)?

- Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
- 2 thread atomicity patterns [Lu ASPLOS ‘06]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  – 2 thread atomicity patterns [Lu ASPLOS ‘06]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>W</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>W</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

Only some patterns are unserializable. Detect unlikely issues via training.
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction \([\text{CACM '75, POPL '04}]\)
  – 2 thread atomicity patterns \([\text{Lu ASPLOS '06}]\)
  – Conflict graphs \([\text{PLDI ‘08, RV ‘11}]\)
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

\[
\begin{align*}
a &= x \\
e &= d \\
c &= x \\
d &= c
\end{align*}
\]

Cycles are unserializable!
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

- How do we know what regions should be atomic?
Concurrent Test Generation

- What if we don’t already have a buggy execution?
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emption finds most bugs
- Careful schedule generation & selection
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
- Generate API unit tests targeting concurrency
  - Small enough for exhaustive schedule exploration
Other Directions

- Shepherding toward good behaviors
- Tolerating & avoiding on the fly
- Static analysis
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard

And the hard problems are interesting to study.