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.
Concurrency & Parallelism

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

![Diagram showing sequential processes and parallel processes]

Sequential
Concurrent & 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
Concurrenty & 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 \textit{critical path} or \textit{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

\[
\text{Given } p \text{ processors, } \frac{Time_1}{p} \leq Time_p \leq \frac{Time_1}{p} + Time_\infty
\]
Using Parallelism

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

- 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

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

- Identifying good opportunities for effective parallelism is open to research
  - 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
  – ...
Data Races

• A data race occurs when:
Data Races

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

T1  T2

x++

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

```
tmp1 = x
tmp1 = tmp1+1
x = tmp1
```

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

T1  T2

T1  T2

x++

tmp = x
tmp = tmp+1
x = tmp

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
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

```java
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;
```

```c
if (!init) {
    lock();
    if (!init) {
        init = true;
        data = create();
    }
    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);
}
“Benign” Data Races

Data race freedom allows extra reads.

```c
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

\[ c = a + 10 \]
\[ \ldots \]
\[ b = a + 10 \]
\[ c = 1 \]

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

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

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can jump forward and backward in time

```java
a = 1
flag = true
```
```java
flag = true
f = flag
b = a
c = a
```

[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

2 can be read after 1 even in the same thread!

[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()

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
unlock()

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

T1  T2

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

T1  T2

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

T1  T2

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

T1 \quad T2

lock(a) \quad write x \quad unlock(a)\quad write y \quad lock(a) \quad read x \quad unlock(a)

Clocks

\begin{align*}
T1 & \mapsto [0, 0] \\
T2 & \mapsto [0, 0]
\end{align*}

Shadow Memory

\begin{align*}
x & \mapsto R[0, 0] \\
W & \mapsto [0, 0] \\
y & \mapsto R[0, 0] \\
W & \mapsto [0, 0]
\end{align*}
Logical Time & Vector Clocks

T1  T2

lock(a)
write x
unlock(a)

write x

lock(a)
read x
unlock(a)

write y

Clocks

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

Shadow Memory

x\mapsto R[0,0]
W[0,0]
y\mapsto R[0,0]
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↦[1,0]
T2↦[0,0]

Shadow Memory

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

Clocks

$T1 \mapsto [2, 0]$

$T2 \mapsto [0, 0]$

Shadow Memory

$X \mapsto R[0, 0]$

$W[1, 0]$

$Y \mapsto R[0, 0]$

$W[0, 0]$
Logical Time & Vector Clocks

Clocks

$T1 \mapsto [2, 0]$

$T2 \mapsto [2, 1]$

Shadow Memory

$x \mapsto R[0,0]$

$y \mapsto R[0,0]$

$W[0,0]$
Logical Time & Vector Clocks

T1  T2

lock(a)
write x
unlock(a)

write x

lock(a)
read x
unlock(a)

T1↦[2,0]
T2↦[2,1]

Clocks

Shadow Memory

write y

x↦R[2,1]
W[1,0]
y↦R[0,0]
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

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

T1  T2

lock(a)  write x  unlock(a)

write x

write y

lock(a)  read x  unlock(a)

Clocks

T1↦[2,0]  T2↦[2,2]

Shadow Memory

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

T1  T2

**Logical Time**
- T1 → [2, 0]
- T2 → [2, 2]

**Vector Clocks**
- Lock(a)
- Write x
- Unlock(a)
- Write y
- Lock(a)
- Read x
- Unlock(a)
- Write x

**Shadow Memory**
- X → R[2, 1]
- W[2, 0]
- Y → R[0, 0]
- W[0, 0]

**Consistency Check**
- \( C(R, X) \not\subseteq C(W, X) \rightarrow \text{race!} \)
  (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!

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

\[
\begin{align*}
\text{lock}(b) \\
\text{write } x \\
\text{unlock}(b)
\end{align*}
\]

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

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

T1  T2

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↦a
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!

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

T1 T2

Shadow Memory

```
y↦a, b
x↦{}
```

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
```

```
x->datum
```

wait/notify or condition variables can fix these
Atomicity Violations

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

```plaintext
tmp = x
tmp = tmp + 1
x = tmp
```
Atomicity Violations

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

\[
\begin{align*}
\text{tmp} &= x \\
\text{tmp} &= \text{tmp} + 1 \\
\text{x} &= \text{tmp}
\end{align*}
\]

\[
\begin{align*}
\text{lock}() \\
\text{tmp} &= x \\
\text{unlock}() \\
\text{tmp} &= \text{tmp} + 1 \\
\text{lock}() \\
\text{x} &= \text{tmp} \\
\text{unlock}()
\end{align*}
\]

No race, similar effect!
Atomicity Violations

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

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

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

What do we really want?
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]

```plaintext
Acq(m)  Rd(y,0)  Rd(x,0)  Wr(y,1)  Wr(x,1)  Wr(y,2)  Rel(m)
```

Right Mover  Both Movers  Left Mover
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>
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]

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

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x</td>
<td>c = x</td>
</tr>
<tr>
<td>e = d</td>
<td>d = c</td>
</tr>
</tbody>
</table>
```
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]

```
a = x

c = x
d = c

e = d
```
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]

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

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-emptions 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.