Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions

Ermenegildo Tomasco  University of Southampton, UK
Truc Lam Nguyen  University of Southampton, UK
Omar Inverso  Gran Sasso Science Institute, Italy
Bernd Fischer  Stellenbosch University, South Africa
Salvatore La Torre  Università di Salerno, Italy
Gennaro Parlato  University of Southampton, UK
Relaxed Memory Consistency

**sequential consistency (SC)**
- memory operations executed in program order within each thread
- changes to the shared memory immediately visible to all threads
- relatively simple to reason about but not realistic

**weak memory models (WMMs)**
- memory operations may be reordered
- used in practice to fully exploit modern hardware
Relaxed Memory Consistency

**total store order (TSO)**
- writes executed in their order for each thread
- reads may overtake writes

**partial store order (PSO)**
- writes to the same location executed in their order for each thread
- writes to different locations may be reordered
- reads may overtake writes
Relaxed Memory Consistency

limitations of testing

• generally ineffective for rare concurrency errors
• cannot control additional nondeterminism introduced by WMMs
• need to be complemented with symbolic analysis
concurrency handling at formula level

- encode threads separately
- add $\varphi_c$ to capture thread interleaving

[Sinha, Wang – POPL 2011]
concurrency handling at formula level

- encode threads separately
- add $\varphi_c$ to capture thread interleaving
  
  [Sinha, Wang – POPL 2011]

extension to WMMs is natural

- change $\varphi_c$ to capture extra interactions due to weaker consistency
  
  [Alglave, Kroening, Tautschnig – CAV 2013]
Symbolic Bug Finding: Lazy Sequentialization + BMC

concurrency handling at code level
• reduction to sequential programs analysis
• implemented as source transformation
• lazy sequentialization tailored to BMC for effective in bug-hunting

[Inverso, Tomasco, Fischer, La Torre, Parlato – CAV 2014]
concurrency handling at code level

- reduction to sequential programs analysis
- implemented as source transformation
- lazy sequentialization tailored to BMC for effective in bug-hunting

[Inverso, Tomasco, Fischer, La Torre, Parlato – CAV 2014]

how to extend to WMMs?
how does it compare?
how to extend to WMMs?

- reduction to concurrent program analysis under SC
- again, implemented as source transformation
Extending Lazy Sequentialization to TSO and PSO

how to extend to WMMs?

- reduction to concurrent program analysis under SC
- again, implemented as source transformation
  - replace shared memory access with explicit function calls to SMA API:
    - `read(v,t)`, `write(v,val,t)`
    - `lock(m,t)`, `unlock(m,t)`, `fence(t)`, ...
  example: `x=y+3` is changed to `write(x,read(y)+3)`
Extending Lazy Sequentialization to TSO and PSO

how to extend to WMMs?

• reduction to concurrent program analysis under SC
• again, implemented as source transformation
  • replace shared memory access with explicit function calls to SMA API:
    `read(v,t), write(v,val,t)`
    `lock(m,t), unlock(m,t), fence(t), ...`
    example: `x=y+3` is changed to `write(x,read(y)+3)`
  • plug in implementation for specific semantics
    `TSO-SMA` - simple implementation
    `eTSO-SMA` - efficient implementation
    `PSO-SMA` - extension to PSO
simple simulation of the store buffer

- introduce one array for each thread
- \textbf{read}(v,t)
  - look up buffer for pending writes
  - fetch from memory
- \textbf{write}(v,val,t)
  - update store buffer
  - inject nondeterministic memory flush
simple simulation of the store buffer

- introduce one array for each thread
- \texttt{read(v,t)}
  - look up buffer for pending writes
  - fetch from memory
- \texttt{write(v,val,t)}
  - update store buffer
  - inject nondeterministic memory flush

TSO-SMA

\[ t_1 \quad \ldots \quad t_T \]
simple simulation of the store buffer

- introduce one array for each thread
- \textbf{read}(v,t)
  - look up buffer for pending writes
  - fetch from memory
- \textbf{write}(v,val,t)
  - update store buffer
  - inject nondeterministic memory flush

TSO-SMA

formula size proportional to
no. memory accesses
no. of store buffers
max no. of elems in the buffer
efficient simulation of the store buffer

- introduce one list for each shared variable and thread
- use global clock and timestamp memory writes

- **read(v,t)**
  - buffer look up, return value from latest pending write
  - return value from latest expired write

- **write(v,val,t)**
  - guess timestamp, enforce non-decreasing order
  - update buffer
efficient simulation of the store buffer

- introduce one list for each shared variable and thread
- use global clock and timestamp memory writes
- read(v, t)
  - buffer look up, return value from latest pending write
  - return value from latest expired write
- write(v, val, t)
  - guess timestamp, enforce non-decreasing order
  - update buffer

**eTSO-SMA**
Variable Write Lists (T-CDLL)

MaxTimestamp = 100,  clock = 11

- store pairs \((value, timestamp)\)
- clock determines expired nodes
- expired nodes not removed

**special nodes**

- **sentinel node**
  has max \(timestamp\)
  does not correspond to any actual write

  - **head**
    only node to contain an expired write followed by a non-expired write
Variable Write Lists (T-CDLL)

MaxTimestamp = 100, clock = 11

- store pairs \((value, timestamp)\)
- clock determines expired nodes
- expired nodes not removed

**special nodes**

- **sentinel node**
  has max timestamp
  does not correspond to any actual write
- **head**
  only node to contain an expired write followed by a non-expired write
Variable Write Lists (T-CDLL)

- store pairs \((value, timestamp)\)
- clock determines expired nodes
- expired nodes not removed

**special nodes**

- **sentinel node**
  has max \(timestamp\)
  does not correspond to any actual write

- **head**
  only node to contain an expired write followed by a non-expired write

\[\text{MaxTimestamp} = 100, \quad \text{clock} = 11\]
Auxiliary Data Structures

parameters

T  max no. of threads
V  max no. of tracked locations (or write lists)
N  max no. of nodes for each variable write list
K  max timestamp

variables

int clock;

• variable write lists
  int value[V][N+1],
  tstamp[V][N+1],
  prev[V][N+1],
  next[V][N+1];

• last values and timestamps
  int last_value[V][T],
  last_tstamp[V][T];

• max timestamp so far
  int max_tstamp[T];
**eTSO-SMA: read operation**

```c
int clock_update() {
    int tmp = *;
    assume(clock <= tmp && tmp <= K);
    clock = tmp;
}

int read(int v, int t) {
    clock_update();

    if (last_tstamp[v][t] > clock)
        return last_value[v][t];

    int node = *;
    assume(node < N &&
            tstamp[v][node] <= clock &&
            tstamp[v][next[v][node]] > clock);
    return value[v][node];
}
```
eTSO-SMA: read operation

```c
int clock_update() {
    int tmp = *;
    assume(clock <= tmp && tmp <= K);
    clock = tmp;
}

int read(int v, int t) {
    clock_update();

    if (last_tstamp[v][t] > clock)
        return last_value[v][t];

    int node = *;
    assume(node < N &&
        tstamp[v][node] <= clock &&
        tstamp[v][next[v][node]] > clock);
    return value[v][node];
}
```

Clock follows non-decreasing order.
int clock_update() {
    int tmp = *;
    assume(clock <= tmp && tmp <= K);
    clock = tmp;
}

int read(int v, int t) {
    clock_update();

    if (last_tstamp[v][t] > clock)
        return last_value[v][t];

    int node = *;
    assume(node < N &&
        tstamp[v][node] <= clock &&
        tstamp[v][next[v][node]] > clock);
    return value[v][node];
}
eTSO-SMA: read operation

```c
int clock_update() {
    int tmp = *
    assume(clock <= tmp && tmp <= K);
    clock = tmp;
}

int read(int v, int t) {
    clock_update();
    if (last_tstamp[v][t] > clock)
        return last_value[v][t];
    int node = *
    assume(node < N &&
              tstamp[v][node] <= clock &&
              tstamp[v][next[v][node]] > clock);
    return value[v][node];
}
```

- **clock follows non-decreasing order**
- **if the last write by t on v has not expired, return the corresponding value**
- **return the value from the latest expired write, which is guaranteed to exist and correspond to the value of v in the memory**
eTSO-SMA: read operation

```c
int clock_update() {
    int tmp = *;
    assume(clock <= tmp && tmp <= K);
    clock = tmp;
}

int read(int v, int t) {
    clock_update();

    if (last_tstamp[v][t] > clock)
        return last_value[v][t];

    int node = *;
    assume(node < N &&
        tstamp[v][node] <= clock &&
        tstamp[v][next[v][node]] > clock);
    return value[v][node];
}
```

- The clock follows non-decreasing order.
- If the last write by `t` on `v` has not expired, return the corresponding value.
- Return the value from the latest expired write, which is guaranteed to correspond to the value of `v` in the memory.
- No longer needed representation of the memory.
eTSO-SMA: write operation

```c
int write(int v, int t) {
    clock_update();

    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;

    int succ = *
    assume(succ <= N && tstamp[v][succ] > clock);
    int pred = prev[v][succ];

    int ts = *
    assume(ts >= clock && ts >= max_tstamp[t]);
    assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

    value[v][node] = val;
    tstamp[v][t] = ts;
    ...
    last_tstamp[v][t] = ts;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
```

select expired node with min timestamp for the new write
int write(int v, int t) {
    clock_update();

    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;

    int succ = *;
    assume(succ <= N && tstamp[v][succ] > clock);
    int pred = prev[v][succ];

    int ts = *;
    assume(ts >= clock && ts >= max_tstamp[t]);
    assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

    value[v][node] = val;
    tstamp[v][t] = ts;
    ...
    last_tstamp[v][t] = ts;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
eTSO-SMA: write operation

```c
int write(int v, int t) {
    clock_update();

    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;

    int succ = *
    assume(succ <= N && tstamp[v][succ] > clock);
    int pred = prev[v][succ];

    int ts = *
    assume(ts >= clock && ts >= max_tstamp[t]);
    assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

    value[v][node] = val;
    tstamp[v][t] = ts;

    last_tstamp[v][t] = ts;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
```

- Position the new node by nondeterministically selecting its successor among the non-expired nodes.
- Select expired node with min timestamp for the new write.
- Guess suitable timestamp, must respect non-decreasing order.
eTSO-SMA: write operation

```c
int write(int v, int t) {
    clock_update();
    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;

    int succ = *
    assume(succ <= N && tstamp[v][succ] > clock);
    int pred = prev[v][succ];

    int ts = *
    assume(ts >= clock && ts >= max_tstamp[t]);
    assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

    value[v][node] = val;
    tstamp[v][t] = ts;
    ...
    last_tstamp[v][t] = ts;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
```

- select expired node with min timestamp for the new write
- position the new node by nondeterministically selecting its successor among the non-expired nodes
- guess suitable timestamp, must respect non-decreasing order
- update variable write list and auxiliary variables
extension to PSO

```c
int write(int v, int t) {
    clock_update();
    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;

    int succ = *;
    assume(succ <= N && tstamp[v][succ] <= clock);
    int pred = prev[v][succ];

    int ts = *;
    assume(ts >= clock && ts >= max_tstamp[t]);
    assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

    value[v][node] = val;
    tstamp[v][t] = ts;
    ...
    last_tstamp[v][t] = ts;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
```

write to different variables may be reordered, guessed timestamps no longer need to be the maximum over all variables, but the maximum for the relevant variable:

```plaintext
    ts >= last_tstamp[t][v]
```
extension to PSO

```c
int write(int v, int t) {
    clock_update();
    int node = next[v][N];
    assume(tstamp[v][next[v][node]] <= clock);
    next[v][N] = next[v][node];
    prev[v][next[v][N]] = N;
    int succ = *;
    assume(succ <= N && tstamp[v][succ] <= clock);
    int pred = prev[v][succ];
    int ts = *
    assume(ts >= clock && ts >= max_tstamp[t] && ts >= tstamp[v][pred] && ts < tstamp[v][succ]);
    value[v][node] = val;
    tstamp[v][t] = ts;
    ...
    last_tstamp[v][t] = t;
    last_value[v][t] = val;
    max_tstamp[t] = ts;
}
```

write to different variables may be reordered, guessed timestamps no longer need to be the maximum over all variables, but the maximum for the relevant variable:

\[
\text{ts} \geq \text{last_tstamp}[t][v]
\]

guessed timestamps may be smaller than the max timestamp:

\[
\text{max_tstamp}[t] = \max(\text{max_tstamp}[t], ts)
\]
### Experimental Evaluation: common benchmarks

<table>
<thead>
<tr>
<th></th>
<th>parameters</th>
<th>TSO runtime (s)</th>
<th>PSO runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LazySMA</td>
<td>CBMC</td>
</tr>
<tr>
<td>dekker</td>
<td>·</td>
<td>1 2 0 0 4 2 2</td>
<td>0.77 0.29 0.04</td>
</tr>
<tr>
<td>lamport</td>
<td>·</td>
<td>1 2 0 0 4 2 2</td>
<td>0.88 0.31 0.05</td>
</tr>
<tr>
<td>peterson</td>
<td>·</td>
<td>1 3 0 0 4 2 2</td>
<td>0.66 0.26 0.04</td>
</tr>
<tr>
<td>szymanski</td>
<td>·</td>
<td>1 3 0 0 4 2 3</td>
<td>0.81 0.34 0.07</td>
</tr>
<tr>
<td>fib_longer_unsafe</td>
<td>·</td>
<td>6 2 0 0 10 6 2</td>
<td>6.47 8.19 94.84</td>
</tr>
<tr>
<td>fib_longer_safe</td>
<td>6 2 0 0 10 6 2</td>
<td>9.78 22.5 t.o.</td>
<td>8.82 31.8 t.o.</td>
</tr>
<tr>
<td>parker</td>
<td>·</td>
<td>1 2 0 0 4 2 3</td>
<td>1.68 0.31 0.05</td>
</tr>
<tr>
<td>stack_unsafe</td>
<td>·</td>
<td>2 2 1 2 5 2 2</td>
<td>1.50 0.41 0.05</td>
</tr>
<tr>
<td>litmus_safe (avg)</td>
<td>5 2 0 0 10 2 20</td>
<td>1.26 0.17 2.35</td>
<td>1.22 0.15 6.65</td>
</tr>
<tr>
<td>litmusUnsafe (avg)</td>
<td>5 2 0 0 10 2 20</td>
<td>1.27 0.16 3.86</td>
<td>1.26 0.12 1.58</td>
</tr>
</tbody>
</table>

**timeout = 600s**

transformation overhead shows on small programs
## Experimental Evaluation: common benchmarks

<table>
<thead>
<tr>
<th></th>
<th>bug?</th>
<th>unwind</th>
<th>qsize (N)</th>
<th>naddr</th>
<th>nmalloc</th>
<th>bitwidth</th>
<th>rounds</th>
<th>maxclock (K)</th>
<th>TSO runtime (s)</th>
<th>PSO runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LazySM</td>
<td>CBMC</td>
<td>NIDHUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LazySM</td>
<td>CBMC</td>
</tr>
<tr>
<td>dekker</td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.77</td>
<td>0.29</td>
</tr>
<tr>
<td>lamport</td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.88</td>
<td>0.31</td>
</tr>
<tr>
<td>peterson</td>
<td>●</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.66</td>
<td>0.26</td>
</tr>
<tr>
<td>szymanski</td>
<td>●</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0.81</td>
<td>0.34</td>
</tr>
<tr>
<td><strong>fib_longer_unsafe</strong></td>
<td>●</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td><strong>6.47</strong></td>
<td>8.19</td>
</tr>
<tr>
<td><strong>fib_longer_safe</strong></td>
<td>●</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td><strong>9.78</strong></td>
<td>22.5</td>
</tr>
<tr>
<td>parker</td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1.68</td>
<td>0.31</td>
</tr>
<tr>
<td>stack_unsafe</td>
<td>●</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1.50</td>
<td>0.41</td>
</tr>
<tr>
<td>litmus_safe (avg)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.26</td>
<td><strong>0.17</strong></td>
</tr>
<tr>
<td>litmus_unsafe (avg)</td>
<td>●</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>1.27</td>
<td><strong>0.16</strong></td>
</tr>
</tbody>
</table>

**timeout = 600s**

competitive on twisted interleavings
## Experimental Evaluation: common benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Parameters</th>
<th>TSO runtime (s)</th>
<th>PSO runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LazySMA</td>
<td>CBMC</td>
</tr>
<tr>
<td>dekker</td>
<td>●</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>lamport</td>
<td>●</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>peterson</td>
<td>●</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>szymanski</td>
<td>●</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>fib_longer_unsafe</td>
<td>●</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>fib_longer_safe</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>parker</td>
<td>●</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>stack_unsafe</td>
<td>●</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>litmus_safe (avg)</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>litmus_unsafe (avg)</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

timeout = 600s

slower
## Experimental Evaluation: common benchmarks

<table>
<thead>
<tr>
<th></th>
<th>bug?</th>
<th>unwind</th>
<th>qsize (N)</th>
<th>naddr</th>
<th>nmalloc</th>
<th>bitwidth</th>
<th>rounds</th>
<th>maxclock (K)</th>
<th>TSO runtime (s)</th>
<th>PSO runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dekker</strong></td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.77</td>
<td>0.75</td>
</tr>
<tr>
<td><strong>lamport</strong></td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td><strong>peterson</strong></td>
<td>●</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.66</td>
<td>0.65</td>
</tr>
<tr>
<td><strong>szymanski</strong></td>
<td>●</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0.81</td>
<td>0.80</td>
</tr>
<tr>
<td><strong>fib_longer_unsafe</strong></td>
<td>●</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>6.47</td>
<td>6.51</td>
</tr>
<tr>
<td><strong>fib_longer_safe</strong></td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td></td>
<td>9.78</td>
<td>8.82</td>
</tr>
<tr>
<td><strong>parker</strong></td>
<td>●</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1.68</td>
<td>2.19</td>
</tr>
<tr>
<td><strong>stackUnsafe</strong></td>
<td>●</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1.50</td>
<td>1.49</td>
</tr>
<tr>
<td><strong>litmusSafe (avg)</strong></td>
<td>●</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>1.26</td>
<td>1.22</td>
</tr>
<tr>
<td><strong>litmusUnsafe (avg)</strong></td>
<td>●</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>1.27</td>
<td>1.26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LazySMA</th>
<th>CBMC</th>
<th>NIDHUGG</th>
<th>LazySMA</th>
<th>CBMC</th>
<th>NIDHUGG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dekker</strong></td>
<td>0.77</td>
<td>0.29</td>
<td><strong>0.04</strong></td>
<td>0.75</td>
<td>0.25</td>
<td><strong>0.05</strong></td>
</tr>
<tr>
<td><strong>lamport</strong></td>
<td>0.88</td>
<td>0.31</td>
<td><strong>0.05</strong></td>
<td>0.88</td>
<td>0.29</td>
<td><strong>0.05</strong></td>
</tr>
<tr>
<td><strong>peterson</strong></td>
<td>0.66</td>
<td>0.26</td>
<td><strong>0.04</strong></td>
<td>0.65</td>
<td>0.25</td>
<td><strong>0.04</strong></td>
</tr>
<tr>
<td><strong>szymanski</strong></td>
<td>0.81</td>
<td>0.34</td>
<td><strong>0.07</strong></td>
<td>0.80</td>
<td>0.32</td>
<td><strong>0.04</strong></td>
</tr>
<tr>
<td><strong>fib_longer_unsafe</strong></td>
<td>6.47</td>
<td>8.19</td>
<td>94.84</td>
<td>6.51</td>
<td>1.69</td>
<td>135.45</td>
</tr>
<tr>
<td><strong>fib_longer_safe</strong></td>
<td>9.78</td>
<td>22.5</td>
<td>t.o.</td>
<td>8.82</td>
<td>31.8</td>
<td>t.o.</td>
</tr>
<tr>
<td><strong>parker</strong></td>
<td>1.68</td>
<td>0.31</td>
<td><strong>0.05</strong></td>
<td>2.19</td>
<td>0.28</td>
<td><strong>0.05</strong></td>
</tr>
<tr>
<td><strong>stackUnsafe</strong></td>
<td>1.50</td>
<td>0.41</td>
<td><strong>0.05</strong></td>
<td>1.49</td>
<td>0.35</td>
<td><strong>0.05</strong></td>
</tr>
<tr>
<td><strong>litmusSafe (avg)</strong></td>
<td>1.26</td>
<td><strong>0.17</strong></td>
<td>2.35</td>
<td>1.22</td>
<td><strong>0.15</strong></td>
<td>6.65</td>
</tr>
<tr>
<td><strong>litmusUnsafe (avg)</strong></td>
<td>1.27</td>
<td><strong>0.16</strong></td>
<td>3.86</td>
<td>1.26</td>
<td><strong>0.12</strong></td>
<td>1.58</td>
</tr>
</tbody>
</table>

**timeout = 600s**

faster than Nidhugg
### Experimental Evaluation: Safestack

<table>
<thead>
<tr>
<th>parameters</th>
<th>TSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
<th>PSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>N rounds</td>
<td>Time</td>
<td>Mem.</td>
<td>Reach?</td>
</tr>
<tr>
<td>1</td>
<td>2 4</td>
<td>10m18s</td>
<td>0.8GB</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>2 3</td>
<td>12m2s</td>
<td>0.6GB</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3 4</td>
<td>13m45s</td>
<td>1.2GB</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>3 3</td>
<td>12m50s</td>
<td>0.9GB</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>2 4</td>
<td>26m55s</td>
<td>1.4GB</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>2 3</td>
<td>24m34s</td>
<td>1.0GB</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>3 4</td>
<td>74m22s</td>
<td>3.4GB</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>3 3</td>
<td>62m22s</td>
<td>1.0GB</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>3 2</td>
<td>12m14s</td>
<td>0.6GB</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>2 4</td>
<td>47m17s</td>
<td>2.4GB</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>2 3</td>
<td>35m7s</td>
<td>1.3GB</td>
<td>No</td>
</tr>
</tbody>
</table>

maxclock=K=1 forces SC analysis,
TSO puts 3x-4x overhead on lazy schema (SC times not shown in table)
Experimental Evaluation: Safestack

<table>
<thead>
<tr>
<th>parameters</th>
<th>TSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
<th>PSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4</td>
<td>10m18s</td>
<td>0.8GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1 2 3</td>
<td>12m2s</td>
<td>0.6GB</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>1 3 4</td>
<td>13m45s</td>
<td>1.2GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1 3 3</td>
<td>12m50s</td>
<td>0.9GB</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>3 2 4</td>
<td>26m55s</td>
<td>1.4GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3 2 3</td>
<td>24m34s</td>
<td>1.0GB</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>3 3 4</td>
<td>74m22s</td>
<td>3.4GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3 3 3</td>
<td>62m22s</td>
<td>1.0GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3 3 2</td>
<td>12m14s</td>
<td>0.6GB</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>7 2 4</td>
<td>47m17s</td>
<td>2.4GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7 2 3</td>
<td>35m7s</td>
<td>1.3GB</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

quicker to spot the bug under PSO
as it requires a smaller number of thread interactions;
performance comparable when no bugs are found
## Experimental Evaluation: Safestack

<table>
<thead>
<tr>
<th>parameters</th>
<th>TSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
<th>PSO analysis (3 bits)</th>
<th>CEX check (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>10m18s</td>
<td>0.8GB</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>12m2s</td>
<td>0.6GB</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>13m45s</td>
<td>1.2GB</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>12m50s</td>
<td>0.9GB</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>26m55s</td>
<td>1.4GB</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>24m34s</td>
<td>1.0GB</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>74m22s</td>
<td>3.4GB</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>62m22s</td>
<td>1.0GB</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>12m14s</td>
<td>0.6GB</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>4</td>
<td>47m17s</td>
<td>2.4GB</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>35m7s</td>
<td>1.3GB</td>
</tr>
</tbody>
</table>

Increase maxlock to cover more reorderings, more resource demanding..
Thank You

users.ecs.soton.ac.uk/gp4/cseq