Finding Rare Concurrent Programming Bugs

An Automatic, Symbolic, Randomized, and Parallelizable Approach

Gennaro Parlato
gennaro@ecs.soton.ac.uk
Concurrent programs

Concurrency is everywhere in computing
- Embedded systems
- multi-core architectures
- worldwide networks

Large concurrent computing resources are available
- clusters
- cloud computing

There is a big demand for concurrent software
- enterprise customer services (e.g., telecom companies)
- government services (e.g., tax payment services)
- social networks, cloud services, …
Developing concurrent programs is difficult

Programmers have to guarantee

- correctness of sequential execution of each individual thread
- under nondeterministic interferences from other threads (interleavings)
Developing concurrent programs is difficult.

What happens here...???

```c
int n=0; //atomic shared variable

int P(void) {
    int tmp, i=1;
    while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
    }
}

int main (void)
    id1 = thread_create(P);
    id2 = thread_create(P);
    join( id1 );
    join( id2 );
    assert(n == 20);
}
```

Can the assert fail?
int n=0; //atomic shared variable

int P(void) {
    int tmp, i=1;
    while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
    }
}

int main (void)
    id1 = thread_create(P);
    id2 = thread_create(P);
    join( id1 );
    join( id2 );
    assert(n > 2);
Scale of the challenge: #interleavings

2 threads with N LOC

#interleavings: \( \binom{2N}{N} \)

Scenario 1:
– N=40
– If 1 billion interleavings are simulated per second
  ➢ 3.4 million years

Scenario 2:
– N=150
  ➢ # interleavings > estimated # atoms in the known universe! \( \geq 10^{80} \)
Bug-finding: finding needles in a haystack

Set of interleavings

Testing is easy when many interleavings are buggy
Bug-finding: finding A needle in a haystack

Set of interleavings

\[
\begin{align*}
x &= *; & y &= *; \\
y &= 0; & x &= 0; \\
\text{if}(x \neq y) & \quad \text{if}(x \neq y) \\
x &= x - y; & y &= y - x; \\
x &= x + 1; & y &= y + 1; \\
z &= x \cdot y; & z &= x \cdot y;
\end{align*}
\]

Haystack

… but is hard when buggy

interleavings are rare

⇒ … needs to be complemented by automated analyses that handle interleavings symbolically
Bounded Model Checking (BMC) of concurrent programs
Testing vs Bounded Model Checking

Testing:
- checks some executions
- may miss errors
- fast

Bounded Model Checking (BMC)
- Exhaustively explores all executions
  - bounding loop iterations
  - bounding context-switchs, etc.
- Can be extremely resource-hungry
BMC for sequential C programs

tools
- BLITZ [ Cho, D'Silva, Song – ASE’13 ]
- CBMC [ Clarke, Kroening, Lerda – TACAS’04 ]
- LLBMC [ Falke, Merz, Sinz – ASE’13 ]
- ESBMC [ Cordeiro, Fischer, Marques-Silva – ASE’09 ]
BMC for concurrent C programs

SAT/SMT approach

• encode each thread as in the sequential case
• add a conjunct for shared memory operations
• all possible interleavings in the bounded program

\[ \varphi_{\text{threads}} \land \varphi_{\text{concurrency}} \]

papers

• [ Sinha, Wang – POPL’11 ]
• [ Alglave, Kroening, Tautschnig – CAV’13 ] — CBMC
Sequentialization targeting BMC
Sequentialization: motivations

Building verification tools for full-fledged concurrent languages is difficult and expensive...

... but scalable verification techniques exist for sequential languages

- Abstraction
- SAT/SMT techniques (i.e., bounded model checking)
- ...

⇒ Can we leverage these?
Sequentialization as a code-to-code translation

Code-to-code translation from multithreaded recursive programs to sequential programs that preserves reachability

Use existing automatic verification techniques designed for sequential programs to analyze concurrent programs
Lazy-CSeq: Schema Overview
(a sequentialization for BMC)

[ Inverso–Tomasco–Fischer–La Torre–Parlato, CAV’14 ]
Lazy-CSeq approach

We have designed new sequentializations targeting BMC scalable analyses + surprisingly simple

Lazy-CSeq
Bounded Concurrent Programs

- no loops
- no function calls
- control flow only forward
- one procedure for each thread
Round Robin Schedule

Lazy-Cseq sequentialization:
- captures all bounded Round-Robin computations for a given bound
- error manifest themselves within very few rounds

[ Musuvathi, Qadeer – PLDI’07 ]
Schema Overview

bounded concurrent program

sequentialization
(code-to-code translation)

“equivalent” sequential program with non determinism

Sequentialized functions

Main Driver
Naïve Lazy Sequentialization

main driver

pc₀=0; ... pcₙ=0;
local₀; ... localₖ;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Tᵢ
      if (activeᵢ)
        Fᵢ();
}
Naïve Lazy Sequentialization

```
main() {
    for (r=0; r<K; r++)
        for (i=0; i<N; i++)
            // simulate T_i
            if (active_i)
                F_i();
}

for each round
    for each thread T_i
        simulate T_i
```
Naïve Lazy Sequentialization

main driver

```c
pc_0 = 0; ... pc_N = 0;
local_0; ... local_k;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate T_i
      if (active_i)
        F_i();
}
```

switch(pc_k) {
  case 0: goto 0;
  case 1: goto 1;
  case 2: goto 2;
  ... 
  case M: goto M;
}

0: stmt0;
1: stmt1;
2: stmt2;
   ... 
M: stmt_M;
Naïve Lazy Sequentialization

main driver

\[
\text{pc}_0 = 0; \ldots \text{pc}_N = 0;
\]
\[
\text{local}_0; \ldots \text{local}_k;
\]

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      \[
      // \text{simulate } T_i
      \]
      if (active\textsubscript{i})
        \[
        F_i();
        \]
  }

switch(pc\textsubscript{i}) {
  case 0: goto 0;
  case 1: goto 1;
  case 2: goto 2;
  \[
  \ldots
  \]
  case M: goto M;
}

0: \[
\text{stmt0};
\]
1: \[
\text{stmt1};
\]
2: \[
\text{stmt2};
\]
\[
\ldots
\]
M: \[
\text{stmt}_M;
\]
Naïve Lazy Sequentialization

main driver

```
pc_0=0; ... pc_N=0;
local_0; ... local_k;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate T_i
      if (active_i)
        F_i();
}
```

```
switch(pc_i) {
  case 0: goto 0;
  case 1: goto 1;
  case 2: goto 2;
  ... 
  case M: goto M;
}
```

```
0: CS(0); stmt0;
1: CS(1); stmt1;
2: CS(2); stmt2;
  ... 
  ... 
M: CS(M); stmt_M;
```

Context-switch mechanism:

```
#define CS(j)
if (*) { pc_i=j; return; }
```
Naïve Lazy Sequentialization

**Formula encoding:**

goto statement to formula

add a guard for each crossing control-flow edge

= $O(M^2)$ guards

Context-switch mechanism:

```c
#define CS(j) 
if (*) { pc_i=j; return; }
```

```c
switch(pc_i) { 
  case 0: goto 0;
  case 1: goto 1;
  case 2: goto 2;
  ...
  case M: goto M;
}
```

0: CS(0); stmt0;
1: CS(1); stmt1;
2: CS(2); stmt2;
    ...
    ...
    ...
M: CS(M); stmt_M;
```
Lazy-CSeq sequentialization

main driver

pc_0 = 0; ... pc_N = 0;
local_0; ... local_k;
nextCS;
main()

for (r=0; r<K; r++)
    for (i=0; i<N; i++)
        // simulate T_i
        if (active_i)
            nextCS = nondet;
            assume(nextCS >= pc_i)
            F_i();
            pc_i = nextCS;
Lazy-CSeq sequentialization

main driver

\[
\begin{align*}
&pc_0=0; \ldots pc_N=0; \\
&local_0; \ldots local_k; \\
&nextCS; \\
&main() \\
&\quad \textbf{for} (r=0; r<K; r++) \\
&\quad \quad \textbf{for} (i=0; i<N; i++) \\
&\quad \quad \quad \text{// simulate } T_i \\
&\quad \quad \quad \textbf{if} (active_i) \\
&\quad \quad \quad \quad \textbf{nextCS} = \text{ nondet}; \\
&\quad \quad \quad \quad \text{assume}(\text{nextCS}>=pc_i) \\
&\quad \quad \quad F_i(); \\
&\quad \quad \quad pc_i = \text{nextCS}; \\
\end{align*}
\]

\[
\begin{align*}
0: J(0); \text{stmt}_0; \\
1: J(1); \text{stmt}_1; \\
2: J(2); \text{stmt}_2; \\
\ldots \\
M: J(M); \text{stmt}_M; \\
\end{align*}
\]

\#define \textbf{J}(j)
\begin{align*}
&\text{ if } (j<pc_i || j>=\text{nextCS}) \text{ goto } j+1;
\end{align*}
Lazy-CSeq sequentialization

```
main driver

pc0=0; ... pcN=0;
local0; ... localk;
nextCS;
main()
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
        nextCS = nondet;
        assume(nextCS>=pci)
        Fi();
      pc_i = nextCS;

#define J(j)
  if (j<pci || j>=nextCS) goto j+1;

EXECUTE

0: J(0); stmt0;
1: J(1); stmt1;
2: J(2); stmt2;
...
M: J(M); stmtM;

resuming + context-switch
```
Lazy-CSeq sequentialization

Formula encoding:

goto statement to formula

add a guard for each crossing control-flow edge

= $O(M)$ guards

#define $J(j)$
if ($j < pc_i$ || $j >= nextCS$) goto $j+1;$
Lazy-CSeq sequentialization

0: J(0); stmt0;
1: J(1); stmt1;
2: J(2); stmt2;
    .       . 
    .       . 
EXECUTE
    .       . 
    .       . 
M: J(M); stmt_M;

#define J(j)
if (j<pc_i || j>=nextCS) goto j+1;

inject light-weight, non-invasive control code
• no non-determinism
• no pc assignments
• no return

resuming + context-switch
Lazy-CSeq tool is a framework that simplifies code-to-code translations
- for C programs + Pthread
- comprises several code-to-code translation modules
- supports several sequential analysis back-end tools

Sequentialisations
- Memory-Unwinding
- Lazy-CSeq, UL-CSeq
- LR-CSeq

Internal modules
- unrolling
- function inlining
- counter-example

Concolic testing
- Klee

bounded model-checking
- BLITZ
- CBMC
- ESBMC
- LLBM

abstraction
- CPA-checker
- Frama-C
- SATABS
- Seahorn
SV-COMP concurrency (2014-17)
Experiments on lock-free data structures
(hard benchmarks)

**Safestack** [Concurrency Testing Using Controlled Schedulers: An Empirical Study, Thomson, Donaldson, Betts, PPoPP’14, TOPC’16]
- ABA problem: requires context bound of 5 for exposure
- **Lazy-CSeq** can find bug in ~7h and 6.5GB
  - #unwind=3, #rounds=4, #threads=4, size=152 visible stmts
- all other tools fail

**Eliminationstack** [Bouajjani, Emmi, Enea, Hamza--POPL’15]
- ABA problem: requires 7 threads for exposure
- **Lazy-CSeq** can find bug in ~13h and 4GB
  - #unwind=1, #rounds=2, #threads=8, size=52 visible stmts
- all other tools fail
State of affairs

BMC

Dream 😊

Testing
How can we get the bales?

How can we partition a task into independent smaller tasks?

How can we partition a task into **independent smaller** tasks?
Tiling threads

Assumption: bounded concurrent programs
- control can only go forward
- same # of stmts, e.g. 1000

\[ T_0 \]
stmt;

\[ T_1 \]
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;

\[ T_2 \]
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;

\[ T_3 \]
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
stmt;
Tiling threads

Tasks as variants of the original program by splitting the code of each thread into fragments (tiles) and allowing context-switches only in some of them.
Tiling threads

- **tile**: (contiguous) subset of visible statements
- **tiling**: partition of program into tiles
- **uniform window tiling**: all tiles have same size
**Tiling threads**

**Observation:** For a $k$-round execution at most $k$ tiles per thread are involved in context-switching!

**Example:** $k=2$

![Diagram showing tiling threads]

---

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
<tr>
<td></td>
<td>stmt;</td>
<td>stmt;</td>
<td>stmt;</td>
</tr>
</tbody>
</table>

---

**Diagram:**

- Tiling threads
- For a $k$-round execution at most $k$ tiles per thread are involved in context-switching.
- Example: $k=2$
**Tiling threads**

**Observation:** For a $k$-round execution at most $k$ tiles per thread are involved in context-switching!

**Example:** $k=2$
**Tiling threads**

**Observation:** For a $k$-round execution at most $k$ tiles per thread are involved in context-switching!

**Example:** $k=2$
**k-selections & program variants**

- **k-selection**: subset of $k$ tiles for each thread
  - *context switches* are only allowed from *selected tiles*
- each $k$-selection specifies a *reduced interleaving instance*
Tiling threads

- **k-selection**: subset of \( k \) tiles for each thread
  - **context switches** are **only** allowed from **selected tiles**
- each \( k \)-selection specifies a **reduced interleaving instance**
How can we get the bales?

How can we partition a task into independent smaller tasks?

How can we partition a task into independent smaller tasks?
How can we get the bales?

Answer:

– fix a tiling and $k$
– generate the program variants for all $k$-selections

# threads

# tiles

$\binom{\text{# tiles}}{k}$

# pgrm variants
How can we get the bales?

Answer:

– fix a tiling
– generate the program variants for all \( k \)-selections

Why does this work?

– each prgm variant captures a subset of \( k \)-round executions of \( P \)
– each execution is captured by a prgm variant
VERISMART architecture
Eliminationstack: results

Eliminationstack

- ABA problem: requires 7 threads for exposure
- Lazy-CSeq can find bug in ~13h and 4GB
  - #unwind=1, #rounds=2, #threads=8, size=52 visible stmts

- each experiment: 8,000 instances chosen randomly

<table>
<thead>
<tr>
<th>#1: tile size 12, t_max 1.5hrs</th>
<th>#2: tile size 14, t_max 2.5hrs</th>
<th>#3: tile size 18, t_max 3hrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification</td>
<td>Time</td>
<td>Memory</td>
</tr>
<tr>
<td>Min</td>
<td>34.9</td>
<td>945.2</td>
</tr>
<tr>
<td>Max</td>
<td>4753.6</td>
<td>1199.1</td>
</tr>
<tr>
<td>Average</td>
<td>1116.3</td>
<td>1017.8</td>
</tr>
</tbody>
</table>

- fastest instances very fast – 1000x
- average still very fast – 40x
- reduced memory consumption – 4x
- high fraction of bug-exposing instances
- some slowdown for larger tile sizes – 10x

Each experiment: 8,000 instances chosen randomly.
Eliminationstack: expected bug-finding time

100x speed-up!

bug found with 99% probability, 5 cores, < 500sec
**Safestack: experiments**

**Safestack**
- ABA problem: requires context bound of 5 for exposure
- **Lazy-CSeq** can find bug in \(~7h\) and **6.5GB**
  - \#unwind=3, \#rounds=4, \#threads=4, size=152 visible stmts

---

<table>
<thead>
<tr>
<th>#1: tile size 11, t_max 1hr</th>
<th>#2: tile size 14, t_max 1hr</th>
<th>#3: tile size 20, t_max 4hrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification    Time</td>
<td>Memory</td>
<td>Verification Time</td>
</tr>
<tr>
<td>Min</td>
<td>195.6</td>
<td>774.5</td>
</tr>
<tr>
<td>Max</td>
<td>2662.6</td>
<td>1265.7</td>
</tr>
<tr>
<td>Average</td>
<td>1172.2</td>
<td>928.8</td>
</tr>
</tbody>
</table>

instances with bug: **1.26%**

instances with bug: **2.14%**

instances with bug: **10.20%**

- lower fraction of bug-exposing instances than eliminationstack
- ...but boosted with larger tile sizes
Safestack: expected bug-finding time

- bug found with 95% probability,
  ~32 cores, ~1300sec
- smaller tiles take longer

25x speed-up!
Conclusions

Lazy-CSeq
BMC: fully symbolic
VERISMArt
Testing

Lazy-CSeq
BMC: fully symbolic

PROBABILITY
PERFORMANCE
Current & Future Work

• Fast over-approximations to filter out safe instances
  – abstract interpretation based on BMC?

• BBD-based analysis + VERISMArt
  – Safestack: bug found < 1 min

• Weak Memory Models
  – Efficient encoding / Lazy-CSeq
    ▶ Memory shadowing
  – VERISMArt
Thank You