Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking

Lucas Cordeiro and Bernd Fischer
lucascordeiro@ufam.edu.br
Bounded Model Checking (BMC)

Basic Idea: check negation of given property up to given depth

\[ \neg \varphi_0 \lor \neg \varphi_1 \lor \neg \varphi_2 \lor \cdots \lor \neg \varphi_{k-1} \lor \neg \varphi_k \]

- transition system \( M \) unrolled \( k \) times
  - for programs: unroll loops, unfold arrays, ...
- translated into verification condition \( \psi \) such that
  \[ \psi \text{ satisfiable iff } \varphi \text{ has counterexample of max. depth } k \]
- has been applied successfully to verify (embedded) software
BMC of Multi-threaded Software

• concurrency bugs are tricky to **reproduce/debug** because they usually occur under specific thread interleavings
  
  – most common errors: **67% related to atomicity and order violations, 30% related to deadlock** [Lu et al.’08]

• problem: the number of interleavings grows exponentially with the number of threads and program statements
  
  – context switches among threads increase the number of possible executions

• two important observations help us:
  
  – concurrency bugs are shallow [Qadeer&Rehof’05]
  
  – SAT/SMT solvers produce unsatisfiable cores that allow us to remove logic that is not relevant
Objective of this work

Exploit SMT to improve BMC of multi-threaded software

• exploit SMT solvers to:
  – prune the *property and data dependent* search space (non-chronological backtracking and conflict clauses learning)
  – remove interleavings that are not relevant by analyzing the proof of unsatisfiability

• propose three approaches to SMT-based BMC:
  – *lazy exploration* of the interleavings
  – *schedule guards* to encode all interleavings
  – *underapproximation and widening (UW)* [Grumberg&et al.’05]

• implement these approaches in ESBMC and evaluate them using multi-threaded applications
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
7:  lock(m1);
8:  if (val1 == 0) {
9:    unlock(m1);
10:   return NULL;
11:  } else {
12:    t1 = val1;
13:    unlock(m1);
14:    lock(m2);
15:    t2 = val2;
16:    unlock(m2);
17:    assert(t2==(t1+1));

program counter: 0
mutexes:  m1 = 0  m2 = 0
globals:  val1 = 0  val2 = 0
locals:  t1 = 0  t2 = 0

val1 and val2 should be updated synchronously

program state; (value of program counter and program variables)
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1

Thread **twoStage**

1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**

7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**Program counter:** 1

**Mutexes:** m1 = 1, m2 = 0

**Globals:** val1 = 0, val2 = 0

**Locals:** t1 = 0, t2 = 0
Lazy exploration of interleavings

**Idea:** iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

**Thread reader**
7: lock(m1);
8: if (val1 == 0) {
9: unlock(m1);
10: return NULL;
}
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**Program counter:** 2
**Mutexes:** m1 = 1  m2 = 0
**Globals:** val1 = 1  val2 = 0
**Locals:** t1 = 0  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: **unlock(m1);**
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

**Thread reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**program counter:** 3
**mutexes:**  **m1 = 0     m2 = 0**
**globals:**  **val1 = 1     val2 = 0**
**locals:**  **t1 = 0     t2 = 0**
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

program counter: 7
mutexes: m1 = 1  m2 = 0
globals: val1 = 1  val2 = 0
locals:  t1 = 0  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));

program counter: 8
mutexes: m1 = 1  m2 = 0
globals: val1 = 1  val2 = 0
locals: t1 = 0  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11:  t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));

**program counter: 12**

**mutexes:** \( m1 = 0 \quad m2 = 0 \)

**globals:** \( \text{val1} = 1 \quad \text{val2} = 0 \)

**locals:** \( t1 = 1 \quad t2 = 0 \)
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11: }
12: t1 = val1;
13: unlock(m1);
14: lock(m2);
15: t2 = val2;
16: unlock(m2);
17: assert(t2 == (t1+1));

program counter: 4
mutexes:  m1 = 0  \textbf{m2 = 1}
globals:  val1 = 1  val2 = 0
locals:  t1 = 1  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: **val2 = val1 + 1;**
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**program counter: 5**
mutexes:  m1 = 0   m2 = 1
globals:  val1 = 1   **val2 = 2**
locals:   t1 = 1   t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4-5-6

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

program counter: 6
mutexes: m1 = 0  m2 = 0
globals: val1 = 1  val2 = 2
locals: t1 = 1  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4-5-6-13

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>Thread reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td>7: lock(m1);</td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>8: if (val1 == 0) {</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>9: unlock(m1);</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td>10: return NULL;</td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td>11: t1 = val1;</td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td>12: unlock(m1);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>program counter: 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutexes: m1 = 0 m2 = 1</td>
</tr>
<tr>
<td>globals: val1 = 1 val2 = 2</td>
</tr>
<tr>
<td>locals: t1 = 1 t2 = 0</td>
</tr>
</tbody>
</table>

CS1

CS2

CS3
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4-5-6-13-14

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));

**program counter: 14**
**mutexes: m1 = 1  m2 = 1**
**globals: val1 = 1  val2 = 2**
**locals: t1 = 1  **t2 = 2
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4-5-6-13-14-15

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9: unlock(m1);
10: return NULL;
}
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2 == (t1 + 1));

**program counter**: 15
**mutexes**: m1 = 1 \(m2 = 0\)
**globals**: val1 = 1 \(val2 = 2\)
**locals**: t1 = 1 \(t2 = 2\)
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3-7-8-11-12-4-5-6-13-14-15-16

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10: return NULL;
11: }
12: t1 = val1;
13: unlock(m1);
14: lock(m2);
15: t2 = val2;
16: unlock(m2);

program counter: 16
mutexes: m1 = 1, m2 = 0
globals: val1 = 1, val2 = 2
locals: t1 = 1, t2 = 2

CS1

interleaving completed, so call single-threaded BMC

QF formula is unsatisfiable, i.e., assertion holds

16: assert(t2 == (t1+1));

...so try next interleaving
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2:

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3

Thread \texttt{twoStage}
1: \texttt{lock(m1)};
2: \texttt{val1 = 1;}
3: \texttt{unlock(m1)};
4: \texttt{lock(m2)};
5: \texttt{val2 = val1 + 1;}
6: \texttt{unlock(m2)};

Thread \texttt{reader}
7: \texttt{lock(m1)};
8: \texttt{if (val1 == 0) \{}
9: \texttt{unlock(m1)};
10: \texttt{return NULL; }
11: \texttt{t1 = val1;}
12: \texttt{unlock(m1)};
13: \texttt{lock(m2)};
14: \texttt{t2 = val2;}
15: \texttt{unlock(m2)};
16: \texttt{assert(t2 == (t1+1));}

\textit{program counter: 3}
\textit{mutexes: m1 = 0 \hspace{1cm} m2 = 0}
\textit{globals: val1 = 1 \hspace{1cm} val2 = 0}
\textit{locals: t1 = 0 \hspace{1cm} t2 = 0}
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3-7

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**

7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**Program counter:** 7

**Mutexes:** m1 = 1  m2 = 0

**Globals:** val1 = 1  val2 = 0

**Locals:** t1 = 0  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3-7-8-11-12-13-14-15-16

Thread **twoStage**
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread **reader**
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

**program counter:** 16
**mutexes:** m1 = 0  m2 = 0
**globals:** val1 = 1  val2 = 0
**locals:** t1 = 1  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3-7-8-11-12-13-14-15-16-4

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11: } t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));

program counter: 4
mutexes: m1 = 0  m2 = 1
globals: val1 = 1  val2 = 0
locals: t1 = 1  t2 = 0
Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3-7-8-11-12-13-14-15-16-4-5-6

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);
7: lock(m1);
8: if (val1 == 0) {
9:   unlock(m1);
10:  return NULL;
11:  t1 = val1;
12:  unlock(m1);
13:  lock(m2);
14:  t2 = val2;
15:  unlock(m2);
16:  assert(t2 == (t1 + 1));
}

QF formula is satisfiable, i.e., assertion fails

...so found a bug for a specific interleaving

interleaving completed, so call single-threaded BMC (again)
Lazy exploration of interleavings

| Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving |

... combines

- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings
Lazy exploration of interleavings – Reachability Tree

Initial state:
- $\nu_0: t_{main}, 0, val1=0, val2=0, m1=0, m2=0, \ldots$
- Active thread, context bound
- Global and local variables

Expansion rules in paper:
- Initial state

Interleaving completed, so call single-threaded BMC
- $\nu_1: t_{twoStage}, 1, val1=0, val2=0, m1=1, m2=0, \ldots$
- CS1

Execution paths:
- $\nu_2: t_{twoStage}, 2, val1=1, val2=0, m1=1, m2=0, \ldots$
Lazy exploration of interleavings – Reachability Tree

Initial state:

- $\nu_0: t_{\text{main}}, 0$
- $\text{val1}=0$, $\text{val2}=0$
- $\text{m1}=0$, $\text{m2}=0$, ...

Global and local variables:

Active thread, context bound:

- backtrack to last unexpanded node and continue

Execution paths:

- execution paths
- blocked execution paths (eliminated)

Symbolic execution can statically determine that path is blocked (encoded in instrumented mutex-op)
Lazy exploration of interleavings – Reachability Tree

Initial state

- $v_0: t_{main}, val1=0, val2=0, m1=0, m2=0, \ldots$

Active thread, context bound

Global and local variables

Execution paths

- $v_1: t_{twoStage}, val1=0, val2=0, m1=1, m2=0, \ldots$

- $v_2: t_{twoStage}, val1=1, val2=0, m1=1, m2=0, \ldots$

- $v_3: t_{reader}, val1=0, val2=0, m1=1, m2=0, \ldots$

- $v_4: t_{reader}, val1=0, val2=0, m1=1, m2=0, \ldots$

- $v_5: t_{twoStage}, val1=0, val2=0, m1=1, m2=0, \ldots$

- $v_6: t_{reader}, val1=0, val2=0, m1=1, m2=0, \ldots$

Blocked execution paths (eliminated)
Lazy approach is naïve but useful

• bugs usually manifest in few context switches
  [Qadeer&Rehof’05]
• bound the number of context switches allowed per thread
  – number of executions: $O(n^c)$
• exploit which transitions are enabled in a given state
  – reduces number of executions
• keep in memory the parent nodes of all unexplored paths only
• each formula corresponds to one possible interleaving only,
  its size is relatively small
• ... but can suffer performance degradation:
  – in particular for correct programs where we need to invoke the
    SMT solver once for each possible execution path
Schedule Recording

Idea: systematically encode all possible interleavings into one formula

- explore reachability tree in same way as lazy approach
- ... but call SMT solver only once
- add a schedule guard $ts_i$ for each context switch block $i$
  $0 < ts_i \leq \#\text{threads}$
  - record in which order the scheduler has executed the program
  - SMT solver determines the order in which threads are simulated
- add scheduler guards only to effective statements
  (assignments and assertions)
  - record effective context switches (ECS)
  - ECS block: sequence of program statements that are executed with no intervening ECS
Schedule Recording – Interleaving #1

statements:
twoStage-ECS:
reader-ECS:

Thread twoStage
1: lock(m1);
2: val1 = 1;
3: unlock(m1);
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

8: if (val1 == 0) {
9: unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));
Schedule Recording – Interleaving #1

statements: 1

twoStage-ECS: (1,1)

reader-ECS:

Thread twoStage
1: lock(m1); \[ts_1 = 1\]  
2: val1 = 1;  
3: unlock(m1);  
4: lock(m2);  
5: val2 = val1 + 1;  
6: unlock(m2);

Guarded statement can only be executed if statement 1 is scheduled in ECS block 1.

each program statement is then prefixed by a schedule guard $ts_i = j$, where:
- $i$ is the ECS block number
- $j$ is the thread identifier
Schedule Recording – Interleaving #1

statements: 1-2

twoStage-ECS: (1,1)-(2,2)

reader-ECS:

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>Thread reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td>7: lock(m1);</td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>8: if (val1 == 0) {</td>
</tr>
<tr>
<td></td>
<td>9: unlock(m1);</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>10: return NULL; }</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td>11: t1 = val1;</td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td>12: unlock(m1);</td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td>13: lock(m2);</td>
</tr>
<tr>
<td></td>
<td>14: t2 = val2;</td>
</tr>
<tr>
<td></td>
<td>15: unlock(m2);</td>
</tr>
<tr>
<td></td>
<td>16: assert(t2==(t1+1));</td>
</tr>
</tbody>
</table>
Schedule Recording – Interleaving #1

statements: 1-2-3

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS:

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>ts₁ == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td></td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>ts₂ == 1</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>ts₃ == 1</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td></td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td></td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread reader</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7: lock(m1);</td>
<td></td>
</tr>
<tr>
<td>8: if (val1 == 0) {</td>
<td></td>
</tr>
<tr>
<td>9: unlock(m1);</td>
<td></td>
</tr>
<tr>
<td>10: return NULL;</td>
<td></td>
</tr>
<tr>
<td>11: t1 = val1;</td>
<td></td>
</tr>
<tr>
<td>12: unlock(m1);</td>
<td></td>
</tr>
<tr>
<td>13: lock(m2);</td>
<td></td>
</tr>
<tr>
<td>14: t2 = val2;</td>
<td></td>
</tr>
<tr>
<td>15: unlock(m2);</td>
<td></td>
</tr>
<tr>
<td>16: assert(t2===(t1+1));</td>
<td></td>
</tr>
</tbody>
</table>
Schedule Recording – Interleaving #1

statements: 1-2-3-7

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)

Thread twoStage
1: lock(m1); \( ts_1 == 1 \)
2: val1 = 1; \( ts_2 == 1 \)
3: unlock(m1); \( ts_3 == 1 \)
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1); \( ts_4 == 2 \)
8: if (val1 == 0) {
9: unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)-(8,5)

Thread twoStage
1: lock(m1); \(ts_1 = 1\)
2: val1 = 1; \(ts_2 = 1\)
3: unlock(m1); \(ts_3 = 1\)
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

Thread reader
7: lock(m1); \(ts_4 = 2\)
8: if (val1 == 0) {
    unlock(m1); \(ts_5 = 2\)
9: unlock(m1);
10: return NULL;
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2 == (t1+1));
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11

twoStage-ECS: \((1,1)-(2,2)-(3,3)\)
reader-ECS: \((7,4)-(8,5)-(11,6)\)

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>Thread reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td>7: lock(m1);</td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>(ts_1 = 1)</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>(ts_2 = 1)</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td>(ts_3 = 1)</td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td></td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td>11: t1 = val1;</td>
</tr>
<tr>
<td></td>
<td>(ts_4 = 2)</td>
</tr>
<tr>
<td></td>
<td>12: unlock(m1);</td>
</tr>
<tr>
<td></td>
<td>13: lock(m2);</td>
</tr>
<tr>
<td></td>
<td>14: t2 = val2;</td>
</tr>
<tr>
<td></td>
<td>15: unlock(m2);</td>
</tr>
<tr>
<td></td>
<td>16: assert(t2==(t1+1));</td>
</tr>
</tbody>
</table>

\[ts_4 = 2\] \[ts_5 = 2\] \[ts_6 = 2\]
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12

twoStage-ECS: (1,1)-(2,2)-(3,3)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

---

Thread twoStage
1: lock(m1);  \( ts_1 == 1 \)
2: val1 = 1;  \( ts_2 == 1 \)
3: unlock(m1);  \( ts_3 == 1 \)
4: lock(m2);
5: val2 = val1 + 1;
6: unlock(m2);

---

Thread reader
7: lock(m1);  \( ts_4 == 2 \)
8: if (val1 == 0) {  \( ts_5 == 2 \)
9:  unlock(m1);
10: return NULL; }
11: t1 = val1;
12: unlock(m1);  \( ts_6 == 2 \)
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4-5

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4-5-6

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)
reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)

Thread twoStage
1: lock(m1); \( ts_1 == 1 \)
2: val1 = 1; \( ts_2 == 1 \)
3: unlock(m1); \( ts_3 == 1 \)
4: lock(m2); \( ts_8 == 1 \)
5: val2 = val1 + 1; \( ts_9 == 1 \)
6: unlock(m2); \( ts_{10} == 1 \)

Thread reader
7: lock(m1); \( ts_4 == 2 \)
8: if (val1 == 0) \{ \( ts_5 == 2 \)
9: unlock(m1);
10: return NULL; \}
11: t1 = val1; \( ts_6 == 2 \)
12: unlock(m1);
13: lock(m2); \( ts_7 == 2 \)
14: t2 = val2;
15: unlock(m2);
16: assert(t2===(t1+1));
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4-5-6-13

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)
reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)
**Schedule Recording – Interleaving #1**

**statements**: 1-2-3-7-8-11-12-4-5-6-13-14

**twoStage-ECS**: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

**reader-ECS**: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>Thread reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td>7: lock(m1);</td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>8: if (val1 == 0) {</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>9: unlock(m1);</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td>10: return NULL;</td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td>11: t1 = val1;</td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td>12: unlock(m1);</td>
</tr>
<tr>
<td></td>
<td>13: lock(m2);</td>
</tr>
<tr>
<td></td>
<td>14: t2 = val2;</td>
</tr>
<tr>
<td></td>
<td>15: unlock(m2);</td>
</tr>
<tr>
<td></td>
<td>16: assert(t2==(t1+1));</td>
</tr>
</tbody>
</table>

\(ts_1 == 1\) \(ts_2 == 1\) \(ts_3 == 1\) \(ts_4 == 2\) \(ts_5 == 2\) \(ts_6 == 2\) \(ts_7 == 2\) \(ts_8 == 1\) \(ts_9 == 1\) \(ts_{10} == 1\) \(ts_{11} == 2\) \(ts_{12} == 2\)
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4-5-6-13-14-15

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)-(15,13)
Schedule Recording – Interleaving #1

statements: 1-2-3-7-8-11-12-4-5-6-13-14-15-16

twoStage-ECS: (1,1)-(2,2)-(3,3)-(4,8)-(5,9)-(6,10)

reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,11)-(14,12)-(15,13)-(16,14)

interleaving completed, so build constraints for interleaving (but do not call SMT solver)
Schedule Recording – Interleaving #2

statements: 1-2-3-7-8-11-12-13-14-15-16-4-5-6

twoStage-ECS: (1,1)-(2,3)-(3,4)-(4,12)-(5,13)-(6,14)
reader-ECS: (7,4)-(8,5)-(11,6)-(12,7)-(13,8)-(14,9)-(15,10)-(16,11)

Thread twoStage
1: lock(m1); \hspace{1cm} ts_1 == 1
2: val1 = 1; \hspace{1cm} ts_2 == 1
3: unlock(m1); \hspace{1cm} ts_3 == 1
4: lock(m2); \hspace{1cm} ts_{12} == 1
5: val2 = val1 + 1; ts_{13} == 1
6: unlock(m2); \hspace{1cm} ts_{14} == 1

Thread reader
7: lock(m1); \hspace{1cm} ts_4 == 2
8: if (val1 == 0) \{ \hspace{1cm} ts_5 == 2
9: unlock(m1);
10: return NULL; \}
11: t1 = val1; \hspace{1cm} ts_6 == 2
12: unlock(m1); \hspace{1cm} ts_7 == 2
13: lock(m2); \hspace{1cm} ts_8 == 2
14: t2 = val2; \hspace{1cm} ts_9 == 2
15: unlock(m2); \hspace{1cm} ts_{10} == 2
16: assert(t2==(t1+1)); ts_{11} == 2
Schedule Recording: Execution Paths

SMT solver instantiates ts to evaluate all possible interleavings

If the guard of the parent node is false then the guard of the child node is false as well
Observations about the schedule recoding approach

• systematically explore the thread interleavings as before, but:
  – add schedule guards to record in which order the scheduler has executed the program
  – encode all execution paths into one formula
    ▶ bound the number of context switches
    ▶ exploit which transitions are enabled in a given state
• number of threads and context switches grows very large quickly, and easily “blow-up” the solver:
  – there is a clear trade-off between usage of time and memory resources
Under-approximation and Widening

Idea: check models with an increased set of allowed interleavings [Grumberg&et al.’05]

• start from a single interleaving (under-approximation) and widen the model by adding more interleavings incrementally

Main steps of the algorithm:

1. encode control literals ($c_{i,j}$) into the verification condition $\psi$
   
   ▷ $c_{i,j}$ where $i$ is the ECS block number and $j$ is the thread identifier

2. check the satisfiability of $\psi$ (stop if $\psi$ is satisfiable)

3. extract proof objects generated by the SMT solver

4. check whether the proof depends on the control literals
   (stop if the proof does not depend on the control literals)

5. remove literals that participated in the proof and go to step 2
UW Approach: Running Example

- use the same guards as in the schedule recording approach as control literals
  - but here the schedule is updated based on the information extracted from the proof

<table>
<thead>
<tr>
<th>Thread twoStage</th>
<th>cl_1,_twoStage \rightarrow ts_1 == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(m1);</td>
<td>cl_2,_twoStage \rightarrow ts_2 == 1</td>
</tr>
<tr>
<td>2: val1 = 1;</td>
<td>cl_3,_twoStage \rightarrow ts_3 == 1</td>
</tr>
<tr>
<td>3: unlock(m1);</td>
<td>cl_8,_twoStage \rightarrow ts_8 == 1</td>
</tr>
<tr>
<td>4: lock(m2);</td>
<td>cl_9,_twoStage \rightarrow ts_9 == 1</td>
</tr>
<tr>
<td>5: val2 = val1 + 1;</td>
<td>cl_10,_twoStage \rightarrow ts_10 == 1</td>
</tr>
<tr>
<td>6: unlock(m2);</td>
<td></td>
</tr>
</tbody>
</table>

- reduce the number of control points from $m \times n$ to $e \times n$
  - $m$ is the number of program statements; $n$ is the number of threads, and $e$ is the number of ECS blocks
Evaluation
Comparison of the Approaches

• Goal: compare efficiency of the proposed approaches
  – lazy exploration
  – schedule recording
  – underapproximation and widening

• Set-up:
  – ESBMC v1.15.1 together with the SMT solver Z3 v2.11
  – support the logics $QF\_AUFBV$ and $QF\_AUFLIRA$
  – standard desktop PC, time-out 3600 seconds
## About the benchmarks

<table>
<thead>
<tr>
<th>Module</th>
<th>#L</th>
<th>#T</th>
<th>#P</th>
<th>B</th>
<th>#C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsbench_ok</td>
<td>81</td>
<td>26</td>
<td>47</td>
<td>26</td>
<td>2</td>
<td>Frangipani file system</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td>80</td>
<td>27</td>
<td>48</td>
<td>27</td>
<td>2</td>
<td>Frangipani file system with array out of bounds</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>77</td>
<td>13</td>
<td>21</td>
<td>129</td>
<td>4</td>
<td>Insert messages into a hash table concurrently</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>1233</td>
<td>3</td>
<td>279</td>
<td>200</td>
<td>2</td>
<td>Multi-threaded download accelerator</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>6366</td>
<td>3</td>
<td>8568</td>
<td>1</td>
<td>9</td>
<td>Data compressor</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>84</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>Data race</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>128</td>
<td>100</td>
<td>13</td>
<td>100</td>
<td>4</td>
<td>Atomicity violation</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>110</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Wrong lock acquisition order</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td>1060</td>
<td>2</td>
<td>24</td>
<td>16</td>
<td>20</td>
<td>Configures the HDMI device</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td>425</td>
<td>2</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>Front panel LED display</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>1109</td>
<td>2</td>
<td>249</td>
<td>2</td>
<td>1</td>
<td>Demonstrate how thumbnail images can be manipulated</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>1171</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>synthetic micro-benchmark</td>
</tr>
</tbody>
</table>
### About the benchmarks

<table>
<thead>
<tr>
<th>Module</th>
<th># of processes</th>
<th># of threads</th>
<th># of total calls</th>
<th># of calls</th>
<th>C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 fsbench_ok</td>
<td>81</td>
<td>26</td>
<td>47</td>
<td>26</td>
<td>2</td>
<td>Frangipani file system</td>
</tr>
<tr>
<td>2 fsbench_bad</td>
<td>80</td>
<td>27</td>
<td>48</td>
<td>27</td>
<td>2</td>
<td>Frangipani file system with array out of bounds</td>
</tr>
<tr>
<td>3 indexer_ok</td>
<td>77</td>
<td>13</td>
<td>21</td>
<td>129</td>
<td>4</td>
<td>Insert messages into a hash table concurrently</td>
</tr>
<tr>
<td>4 aget-0.4_bad</td>
<td>1233</td>
<td>3</td>
<td>279</td>
<td>200</td>
<td>2</td>
<td>Multi-threaded download accelerator</td>
</tr>
<tr>
<td>5 bzip2smp_ok</td>
<td>6366</td>
<td>3</td>
<td>8568</td>
<td>1</td>
<td>9</td>
<td>Data compressor</td>
</tr>
<tr>
<td>6 reorder_bad</td>
<td>84</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>Data race</td>
</tr>
<tr>
<td>7 twostage_bad</td>
<td>128</td>
<td>100</td>
<td>13</td>
<td>100</td>
<td>4</td>
<td>Atomicity violation</td>
</tr>
<tr>
<td>8 wronglock_bad</td>
<td>110</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Wrong lock acquisition order</td>
</tr>
<tr>
<td>9 exStbHDMI_ok</td>
<td>1060</td>
<td>2</td>
<td>24</td>
<td>16</td>
<td>20</td>
<td>Configures the HDMI device</td>
</tr>
<tr>
<td>10 exStbLED_ok</td>
<td>425</td>
<td>2</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>Front panel LED display</td>
</tr>
<tr>
<td>11 exStbThumb_bad</td>
<td>1109</td>
<td>2</td>
<td>249</td>
<td>2</td>
<td>1</td>
<td>Demonstrate how thumbnail images can be manipulated</td>
</tr>
<tr>
<td>12 micro_10_ok</td>
<td>1171</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>synthetic micro-benchmark</td>
</tr>
</tbody>
</table>
## About the benchmarks

<table>
<thead>
<tr>
<th>Module</th>
<th>#L</th>
<th>#T</th>
<th>#P</th>
<th>B</th>
<th>#C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsbench_ok</td>
<td>81</td>
<td>26</td>
<td>47</td>
<td>26</td>
<td>2</td>
<td>Frangipani file system</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td>80</td>
<td>27</td>
<td>48</td>
<td>27</td>
<td>2</td>
<td>Frangipani file system with array out of bounds</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>77</td>
<td>13</td>
<td>21</td>
<td>129</td>
<td>4</td>
<td>Insert messages into a hash table concurrently</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>1233</td>
<td>3</td>
<td>279</td>
<td>200</td>
<td>2</td>
<td>Multi-threaded download accelerator</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>6366</td>
<td>3</td>
<td>8568</td>
<td>1</td>
<td>9</td>
<td>Data compressor</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>84</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>Data race</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>128</td>
<td>100</td>
<td>13</td>
<td>100</td>
<td>4</td>
<td>Atomicity violation</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>110</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Wrong lock acquisition order</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td>1060</td>
<td>2</td>
<td>24</td>
<td>16</td>
<td>20</td>
<td>Configures the HDMI device</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td>425</td>
<td>2</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>Front panel LED display</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>1109</td>
<td>2</td>
<td>249</td>
<td>2</td>
<td>1</td>
<td>Demonstrate how thumbnail images can be manipulated</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>1171</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>Synthetic micro-benchmark</td>
</tr>
</tbody>
</table>
## About the benchmarks

<table>
<thead>
<tr>
<th>Module</th>
<th>#L</th>
<th>#T</th>
<th>#P</th>
<th>B</th>
<th>#C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsbench_ok</td>
<td>81</td>
<td>26</td>
<td>47</td>
<td>26</td>
<td>2</td>
<td>Frangipani file system</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td>80</td>
<td>27</td>
<td>48</td>
<td>27</td>
<td>2</td>
<td>Frangipani file system with array out of bounds</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>77</td>
<td>13</td>
<td>21</td>
<td>129</td>
<td>4</td>
<td>Insert messages into a hash table concurrently</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>1233</td>
<td>3</td>
<td>279</td>
<td>200</td>
<td>2</td>
<td>Multi-threaded download accelerator</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>6366</td>
<td>3</td>
<td>8568</td>
<td>1</td>
<td>9</td>
<td>Data compressor</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>84</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>Data race</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>128</td>
<td>100</td>
<td>13</td>
<td>100</td>
<td>4</td>
<td>Atomicity violation</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>1109</td>
<td>2</td>
<td>249</td>
<td>2</td>
<td>1</td>
<td>Wrong lock acquisition order</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td>1060</td>
<td>2</td>
<td>24</td>
<td>16</td>
<td>20</td>
<td>Configures the HDMI device</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td>425</td>
<td>2</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>Front panel LED display</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>1109</td>
<td>2</td>
<td>249</td>
<td>2</td>
<td>1</td>
<td>Demonstrate how thumbnail images can be manipulated</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>1171</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>Synthetic micro-benchmark</td>
</tr>
</tbody>
</table>

*Set-top box applications from NXP semiconductors*
## About the benchmarks

<table>
<thead>
<tr>
<th>Module</th>
<th>#L</th>
<th>#T</th>
<th>#P</th>
<th>B</th>
<th>#C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsbench_ok</td>
<td>81</td>
<td>26</td>
<td>47</td>
<td>26</td>
<td>2</td>
<td>Frangipani file system</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td>80</td>
<td>27</td>
<td>48</td>
<td>27</td>
<td>2</td>
<td>Frangipani file system with array out of bounds</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>77</td>
<td>13</td>
<td>21</td>
<td>129</td>
<td>4</td>
<td>Insert messages into a hash table concurrently</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>1233</td>
<td>3</td>
<td>279</td>
<td>200</td>
<td>2</td>
<td>Multi-threaded download accelerator</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>6366</td>
<td>3</td>
<td>8568</td>
<td>1</td>
<td>8</td>
<td>Data compressor</td>
</tr>
<tr>
<td>reorder_bad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data race</td>
</tr>
<tr>
<td>twostage_bad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Atomicity violation</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wrong lock acquisition order</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Configures the HDMI device</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Front panel LED display</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Demonstrate how thumbnail images can be manipulated</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>1171</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>17</td>
<td>synthetic micro-benchmark</td>
</tr>
</tbody>
</table>

It is used to check the scalability of multi-threaded software verification tools [Ghafari 2010]
### Comparison of the approaches

<table>
<thead>
<tr>
<th>Module</th>
<th>Lazy Schedule UW</th>
<th>Time</th>
<th>Result</th>
<th>#FI/#I</th>
<th>Time</th>
<th>Result</th>
<th>Time</th>
<th>Result</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsbench_ok</td>
<td></td>
<td>282</td>
<td>+</td>
<td>0/676</td>
<td>304</td>
<td>+</td>
<td>301</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td></td>
<td>&lt;1</td>
<td>+</td>
<td>1/1</td>
<td>729</td>
<td>+</td>
<td>360</td>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>indexer_ok</td>
<td></td>
<td>595</td>
<td>+</td>
<td>0/17160</td>
<td>220</td>
<td>+</td>
<td>218</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td></td>
<td>137</td>
<td>+</td>
<td>1/1</td>
<td>127</td>
<td>+</td>
<td>125</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td></td>
<td>1800</td>
<td>+</td>
<td>0/1294</td>
<td>MO</td>
<td>-</td>
<td>MO</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>reorder_bad</td>
<td></td>
<td>&lt;1</td>
<td>+</td>
<td>1/154574</td>
<td>MO</td>
<td>-</td>
<td>MO</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>twostage_bad</td>
<td></td>
<td>88</td>
<td>+</td>
<td>1/139</td>
<td>93</td>
<td>+</td>
<td>195</td>
<td>+</td>
<td>5</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td></td>
<td>90</td>
<td>+</td>
<td>6/104015</td>
<td>MO</td>
<td>-</td>
<td>MO</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td></td>
<td>229</td>
<td>+</td>
<td>0/1</td>
<td>226</td>
<td>+</td>
<td>213</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td></td>
<td>73</td>
<td>+</td>
<td>0/11</td>
<td>73</td>
<td>+</td>
<td>787</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td></td>
<td>95</td>
<td>+</td>
<td>3/3</td>
<td>14</td>
<td>+</td>
<td>12</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td></td>
<td>254</td>
<td>+</td>
<td>0/29260</td>
<td>MO</td>
<td>-</td>
<td>MO</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Number of generated and failed interleavings**
- **Number of iterations**
- **Encoding and solver time**
- **Good thing**: error detected in module “+”
- **Bad thing**: error occurred in tool “–”
### Comparison of the approaches (1)

<table>
<thead>
<tr>
<th>Module</th>
<th>Lazy</th>
<th>Schedule</th>
<th>UW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Result</td>
<td>#FI/#I</td>
</tr>
<tr>
<td>fsbench_good</td>
<td>282</td>
<td>+</td>
<td>0/676</td>
</tr>
<tr>
<td>fsbench_bad</td>
<td>&lt;1</td>
<td>+</td>
<td>729/729</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>595</td>
<td>+</td>
<td>0/17160</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>137</td>
<td>+</td>
<td>1/1</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>1800</td>
<td>+</td>
<td>0/1294</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>&lt;1</td>
<td>+</td>
<td>1/154574</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>88</td>
<td>+</td>
<td>1/139</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>90</td>
<td>+</td>
<td>6/104015</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td>229</td>
<td>+</td>
<td>0/1</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td>73</td>
<td>+</td>
<td>0/11</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>95</td>
<td>+</td>
<td>3/3</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>254</td>
<td>+</td>
<td>0/29260</td>
</tr>
</tbody>
</table>

*lazy encoding often more efficient than schedule recording and UW*
## Comparison of the approaches (2)

<table>
<thead>
<tr>
<th></th>
<th>Lazy Encoding</th>
<th>Schedule Recording</th>
<th>UW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta</td>
<td>#FI/#I</td>
<td>Time</td>
<td>Result</td>
</tr>
<tr>
<td>fsbench_ok</td>
<td>+ 0/676</td>
<td>304 +</td>
<td>301 + 1</td>
</tr>
<tr>
<td>indexer_ok</td>
<td>595 + 0/17160</td>
<td>220 +</td>
<td>218 + 1</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>137 + 1/1</td>
<td>127 +</td>
<td>125 + 1</td>
</tr>
<tr>
<td>bzip2smp_ok</td>
<td>1800 + 0/1294</td>
<td>MO -</td>
<td>MO - 1</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>&lt;1 + 1/154574</td>
<td>MO -</td>
<td>MO - 1</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>88 + 1/139</td>
<td>93 +</td>
<td>195 + 5</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>90 + 6/104015</td>
<td>MO -</td>
<td>MO - 1</td>
</tr>
<tr>
<td>exStbHDMI_ok</td>
<td>229 + 0/1</td>
<td>226 +</td>
<td>213 + 1</td>
</tr>
<tr>
<td>exStbLED_ok</td>
<td>73 + 0/11</td>
<td>73 +</td>
<td>787 + 1</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>95 + 3/3</td>
<td>14 +</td>
<td>12 + 1</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>254 + 0/29260</td>
<td>MO -</td>
<td>MO - 1</td>
</tr>
</tbody>
</table>

*lazy encoding often more efficient than schedule recording and UW, **but not always**

lazy encoding often more efficient than schedule recording and UW, but not always
## Comparison of the approaches (3)

<table>
<thead>
<tr>
<th>Module</th>
<th>Lazy</th>
<th>Schedule</th>
<th>UW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Result</td>
<td>#FI/#I</td>
</tr>
<tr>
<td>ffsbench.ok</td>
<td>282</td>
<td>+</td>
<td>0/676</td>
</tr>
<tr>
<td>ffsbench_bad</td>
<td>&lt;1</td>
<td>+</td>
<td>729/729</td>
</tr>
<tr>
<td>indexer.ok</td>
<td>595</td>
<td>+</td>
<td>0/17160</td>
</tr>
<tr>
<td>aget-0.4_bad</td>
<td>137</td>
<td>+</td>
<td>1/1</td>
</tr>
<tr>
<td>bzip2smp.ok</td>
<td>1800</td>
<td>+</td>
<td>0/1294</td>
</tr>
<tr>
<td>reorder_bad</td>
<td>&lt;1</td>
<td>+</td>
<td>1/154574</td>
</tr>
<tr>
<td>twostage_bad</td>
<td>88</td>
<td>+</td>
<td>1/139</td>
</tr>
<tr>
<td>wronglock_bad</td>
<td>90</td>
<td>+</td>
<td>6/104015</td>
</tr>
<tr>
<td>exStbHDMI.ok</td>
<td>229</td>
<td>+</td>
<td>0/1</td>
</tr>
<tr>
<td>exStbLED.ok</td>
<td>73</td>
<td>+</td>
<td>0/11</td>
</tr>
<tr>
<td>exStbThumb_bad</td>
<td>95</td>
<td>+</td>
<td>3/3</td>
</tr>
<tr>
<td>micro_10_ok</td>
<td>254</td>
<td>+</td>
<td>0/29260</td>
</tr>
</tbody>
</table>

*lazy encoding is extremely fast for satisfiable instances*
Comparison to CHESS [Musuvathi and Qadeer]

- CHESS (v0.1.30626.0) is a concurrency testing tool for C# programs; also works for C/C++ (Windows API).
  - implements iterative context-bounding
  - requires unit tests that it repeatedly executes in a loop, exploring a different interleaving on each iteration
  - performs state hashing based on a happens-before graph
    - avoids exploring the same state repeatedly

- Goal: compare efficiency of the approaches
  - on identical verification problems taken from standard benchmark suites of multi-threaded software
CHESS is effective for programs where there are a small number of threads

<table>
<thead>
<tr>
<th>Function</th>
<th>B</th>
<th>C</th>
<th>Time</th>
<th>Tests</th>
<th>Lazy</th>
<th>#FI/#I</th>
</tr>
</thead>
<tbody>
<tr>
<td>reorder_4_bad (3,1)</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>98</td>
<td>130000</td>
<td>&lt;1</td>
</tr>
<tr>
<td>reorder_5_bad (4,1)</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>TO</td>
<td>429000</td>
<td>&lt;1</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>TO</td>
<td>396000</td>
<td>&lt;1</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>TO</td>
<td>371000</td>
<td>&lt;1</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>TO</td>
<td>367000</td>
<td>&lt;1</td>
</tr>
<tr>
<td>twostage_4_bad (3,1)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>215</td>
<td>27000</td>
<td>2</td>
</tr>
<tr>
<td>twostage_5_bad (4,1)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>TO</td>
<td>384000</td>
<td>2</td>
</tr>
<tr>
<td>twostage_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>TO</td>
<td>366000</td>
<td>2</td>
</tr>
<tr>
<td>wronglock_4_bad (1,3)</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>21</td>
<td>3000</td>
<td>5</td>
</tr>
<tr>
<td>wronglock_5_bad (1,4)</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>724</td>
<td>93000</td>
<td>10</td>
</tr>
<tr>
<td>wronglock_6_bad (1,5)</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>TO</td>
<td>356000</td>
<td>18</td>
</tr>
<tr>
<td>micro_2_ok (100)</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>316</td>
<td>35855</td>
<td>&lt;1</td>
</tr>
<tr>
<td>micro_2_ok (100)</td>
<td>2</td>
<td>1</td>
<td>17</td>
<td>TO</td>
<td>40000</td>
<td>1095</td>
</tr>
</tbody>
</table>
CHESS is effective for programs where there are a small number of threads, but it does not scale that well and consistently runs out of time when we increase the number of threads.

<table>
<thead>
<tr>
<th></th>
<th>CHESS</th>
<th>Lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Tests</td>
</tr>
<tr>
<td>reorder_4_bad (3,1)</td>
<td>5</td>
<td>98</td>
</tr>
<tr>
<td>reorder_5_bad (4,1)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>reorder_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>twostage_4_bad (3,1)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>twostage_5_bad (4,1)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>twostage_6_bad (5,1)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>wronglock_4_bad (1,3)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>wronglock_5_bad (1,4)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>wronglock_6_bad (1,5)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>micro_2_ok (100)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>micro_2_ok (100)</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Results

• lazy, schedule recording, and UW algorithms
  – lazy: check constraints lazily is fast for satisfiable instances and to a lesser extent even for safe programs
    ⇒ it has not been described or evaluated in the literature
  – schedule recording: the number of threads and context switches can grow quickly (and easily “blow-up” the model checker)
    ⇒ combines symbolic with explicit state space exploration
  – UW: memory overhead and slowdowns to extract the unsat core
    ⇒ it has not been used for BMC of multi-threaded software

Future Work

• fault localization in multi-threaded C programs
• verify real-time software using SMT techniques
• interpolants to prove non-interference of context switches