Poster: Static Analysis of Concurrent Higher-Order Programs

Quentin Stievenart, Jens Nicolay, Wolfgang De Meuter, Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be

Abstract—Few static analyses support concurrent higher-order programs. Tools for detecting concurrency bugs such as deadlocks and race conditions are nonetheless invaluable to developers. Concurrency can be implemented using a variety of models, each supported by different synchronization primitives. Using this poster, we present an approach for analyzing concurrent higher-order programs in a precise manner through abstract interpretation. We instantiate the approach for two static analyses that are capable of detecting deadlocks and race conditions in programs that rely either on compare-and-swap (cas), or on conventional locks for synchronization. We observe few false positives and false negatives on a corpus of small concurrent programs, with better results for the lock-based analyses. We also observe that these programs lead to a smaller state space to be explored by the analyses. Our results show that the choice of synchronization primitives supported by an abstract interpreter has an important impact on the complexity of the static analyses performed with this abstract interpreter.

I. INTRODUCTION

It is difficult to find concurrency bugs such as deadlocks and race conditions by hand. An exponential number of thread interleavings needs to be explored. Higher-order programming features, such as functions taking others as arguments, exacerbate this difficulty. The need for tool support increases as these features become more widespread.

The AAM approach to abstract interpretation [9] provides a theoretical foundation for this support. It has already been adapted to concurrent programs [6], giving rise to the P(CEK) machine that features support for multiple threads of execution (i.e., through primitives spawn and join), as well as for their synchronization through a compare-and-swap primitive (i.e., cas). In addition to supporting higher-order features, its qualities include conciseness and extensibility.

We extend the P(CEK) machine with first-class support for locks as a synchronization primitive and demonstrate its applicability as the foundation for two client analyses that detect deadlocks and race conditions [8]. The corresponding implementations are publicly available1.

II. FOUNDATION

We observed that the P(CEK) machine only scales to small concurrent programs (around 60 lines of code with 3 threads). This is due to the state explosion problem, also shared by model checking techniques that support concurrent programs. Since every thread interleaving is computed, exploring the corresponding state space quickly becomes intractable. Research on this problem in the area of model checking (cf. [4]) has lead to techniques such as partial-order reduction [7], binary decision diagrams [3], and bounded model checking [1]. However, we leave incorporating these techniques in the P(CEK) machine as future work.

Instead, we orthogonally address another source of complexity in the generated state space: the synchronization primitives supported by the abstract interpreter. The original P(CEK) machine uses an atomic compare-and-swap synchronization primitive, cas. This primitive works in a active way: (cas v old new) will compare the content of the variable v with the value old, and only if the content matches, v will be updated to take the value of new. If the comparison fails (e.g., due to a bad thread interleaving, which would make the computation of new incorrect if based on the value old), the update has to be tried again. This is how a shared concurrent counter would be implemented using cas:

\[
\text{(letrec ((v 0)} \\
\text{ (inc (lambda ()} \\
\text{ (let ((old v) (new (+ old 1)))} \\
\text{ (if (cas v old new)} \\
\text{ #t ;; successful update} \\
\text{ ;; incorrect update (v != old),} \\
\text{ ;; try again} \\
\text{ (inc)))}}))
\]

The fact that a failing cas has to be retried over and over until it eventually succeeds, leads to an increase in the size of the state space where many states are generated only because a cas is retried, and will not lead to real progress in the program execution.

Another common synchronization mechanism is locking. Locks can be simulated with cas, and no first-class language support is necessary to use locks. The lock-based equivalent of the shared concurrent counter example is shown below.

Based on the observations that cas introduces an overhead in terms of number of states, and that most programs tend to use locks instead of cas, we add first-class support for locks to the P(CEK) machine. This way, the abstract interpreter no longer needs to analyze a low-level library implementation of locking in terms of cas. A lock can be either #locked or #unlocked. It can be acquired with the acquire primitive,

1https://github.com/acieroid/pcesk
which blocks the current thread until the lock is acquired. Once acquired, a lock can be released with the \texttt{release} primitive.

III. CLIENT ANALYSES

One can design static analyses for concurrent programs as an exploration of the state space generated by the P(CEK)S machine. We have formulated analyses for detecting deadlocks and race conditions, each for programs using \texttt{cas} and for programs using first-class locks [8]. We have observed that analyses for programs making use of locks are more straightforward to formulate. For example, the race condition analysis with locks is equivalent to a conflict analysis. With \texttt{cas}, the conflict analysis has to be augmented by another analysis to catch a source of race conditions not present with locks.

IV. EVALUATION

We tested our client analyses on a small number of concurrent programs exhibiting various situations of deadlocks and race conditions. The results are given in Table I. Lock-based analyses tend to be able to handle programs with more threads and locks, and have fewer false positives when dealing with deadlocks. The only false positive found by both race condition analyses is a benign race condition which has no effect on the outcome of the programs, and it is arguable whether this instance of race condition should be detected.

<table>
<thead>
<tr>
<th></th>
<th>Deadlock Analysis</th>
<th>Race Condition Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{cas} locks</td>
<td>\texttt{cas} locks</td>
</tr>
<tr>
<td>Input programs</td>
<td>6 7</td>
<td>8 6</td>
</tr>
<tr>
<td>Max. LOC</td>
<td>65 58</td>
<td>45 52</td>
</tr>
<tr>
<td>Max. threads</td>
<td>3 4</td>
<td>3 4</td>
</tr>
<tr>
<td>Max. locks</td>
<td>2 3</td>
<td>- 1</td>
</tr>
<tr>
<td>Max. states</td>
<td>23 4623</td>
<td>38 29</td>
</tr>
<tr>
<td>Defects</td>
<td>7 5</td>
<td>7 2</td>
</tr>
<tr>
<td>False positives</td>
<td>2 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

V. CONCLUSION

Our poster presents the P(CEK)S machine and its design based on the CESK machine. The different synchronization mechanisms supported (\texttt{cas} and locks) are introduced through examples. We show that switching from \texttt{cas} to first-class locks in the analyzed programs can lead to a reduction of the state space of around one order of magnitude, and up to three orders of magnitude. We also introduce our deadlock and race condition analyses.

This approach of performing static analysis for concurrent programs has the advantage of supporting higher-order programs, unlike most existing concurrent static analyses. It also outputs a precise state graph that over-approximates executions of the program. One downside of this approach is the size of the state graph (up to $10^6$ states in our examples), which is subject to the state explosion problem, as concurrent model checkers are. However, it allows a relatively straightforward formulation of static analyses, such as our deadlock and race condition analyses.

Future work includes tackling the state explosion problem to improve the analysis’ scalability to support programs of a complexity closer to real-world applications. We aim to reduce the size of the state space further, by adapting techniques that have been proven useful in the context of model checking (e.g., to verify models of up to $2^{120}$ states [2]). One such technique is cartesian partial-order reduction [5], which is formulated in a manner close to abstract interpretation.

Adapting the P(CEK)S machine to more precise abstract interpretation models (e.g., pushdown-based models) could improve the precision of the derived states, but requires a non straightforward adaptation. Finally, adapting the P(CEK)S machine to other concurrency models and other synchronization primitives could lead to further improvements and new insights.

REFERENCES


Static analysis of concurrent higher-order programs

Quentin Stievenart, Jens Nicolay, Wolfgang De Meuter, Coen De Roover
{qstieven, jnicolay, wdemeuter, cderoove}@vub.ac.be
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Context: concurrent higher-order languages

Multi-threaded execution in a Scheme variant:

```
; Creating a thread
(let ((t (spawn (fib 12))))
[...])
; Waiting for a thread to finish
(join t)
```

Synchronization primitives:

- Concurrent counter implemented with cas
- (atomic compare-and-swap)
- (a-lock)
- (if (inc (lambda ()
    ;; (atomic compare-and-swap)
    ;; Waiting for a thread to finish
    [...]))

Example state graph for a very small multithreaded program.

Approach: the P(CEK)S abstract machine

The P(CEK)S machine [1] is a static analysis technique:

- based on abstract interpretation
- that supports higher-order programs
- that supports concurrent programs.

```
P(c E) E S K set of threads control: C environment: E store: S continuation: K shared store
```

- models a single-threaded machine
- supports the base language (Scheme)
- first-class support for spawn, join, cas
- not yet for locks (acquire, release)
- transition function \( \Rightarrow \)
- transition function \( \Rightarrow_s \), defined in terms of \( \Rightarrow \)

From \( \Rightarrow_s \), we compute a state graph over-approximating every feasible path

Client analyses: cas vs. locks

Contribution: adding first-class support for locks to the P(CEK)S machine simplifies formulating client analyses [2].

Deadlock detection

A deadlock occurs when the computation does not make any progress.

**With cas:**

Search for a loop within the state graph that has no success branch out of cas:

```
1: (let ((t (spawn (fib 12))))
2: (a-lock)
3: (if (inc (lambda ()
    (atomic compare-and-swap)
    ;; Waiting for a thread to finish
    [...])))
```

**With locks:**

Search for a state (grouping multiple threads) simultaneously evaluating acquire, release, and joins that has no successors.

```
1: (join t)
2: (join t)
3: (acquire 1)
4: (release 1)
```

Race condition detection

A race condition occurs when the result of a computation depends on the thread interleaving.

With cas:

```
1: (set! v 1)
2: (set! v 1)
```

With locks:

```
1: (acquire 1)
2: (release 1)
```

**With cas:**

Same as for locks, except that a subgraph has to be found in which there is no back edge from the failure state to the cas itself.

**With locks:**

Detect read-write conflicts by searching for a state (grouping multiple threads) that contains simultaneous read and write

```
1: (join t)
2: (a-lock)
```

Results

<table>
<thead>
<tr>
<th>Complexity.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contribution:</strong> switching from cas to locks reduces state space by up to 3 orders of magnitude [2].</td>
</tr>
<tr>
<td><strong>Cause:</strong></td>
</tr>
<tr>
<td>- cas is an active way of performing synchronization, generates many states,</td>
</tr>
<tr>
<td>- acquiring a lock blocks a thread, preventing it from generating further states.</td>
</tr>
<tr>
<td>- Still only scales to small concurrent programs (future work: adapt state space reduction techniques from model checking)</td>
</tr>
</tbody>
</table>

![Graph showing results]

Precision and scalability.

**Contribution:** adding first-class support for locks improves accuracy [2].

### Contribution

- Precision:
  - Adding first-class support for locks improves accuracy [2].

### Implementation

Available at [https://github.com/acieroid/pcesk](https://github.com/aceroid/pcesk)

References