Effect Summaries for Thread-Modular Analysis

Lukáš Holík[1], Roland Meyer[2], Tomáš Vojnar[1], and Sebastian Wolff[2]

[1] Brno University of Technology [2] TU Braunschweig

Goal

Automated verification of:

- lock-free data structures
    - ➡ complex, low-level concurrency
- libraries
    - ➡ (arbitrarily) many client threads
- explicit memory management
    - ➡ subtle memory bugs (ABA)

Thread-Modular Verification [Flanagan et al. SPIN'03]

- *View abstraction* splits states into set of views
  - ➡ capturing the system as seen by a single thread
  - ➡ abstracting away correlation among threads
- State space exploration as fixed point

$$X = X \cup sequential(X) \cup interference(X)$$

Lets every view in X perform a step of *its own* thread.

Applies to views in X possible influence by other threads.

Thread-Modular Interference

Learning approach [Vafeiadis VMCAI'10]

- Update patterns
  - ➡ symbolic representation of modifications performed by the threads
  - ➡ collected from sequential steps
- Interference
  - ➡ apply update patterns to the views from X
  - ➡ requires matching to check applicability of update pattern

Thread-Modular Interference

Learning approach [Vafeiadis VMCAI'10] ✗ **successful, but only for GC**

- Update patterns
  - ➡ symbolic representation of modifications performed by the threads
  - ➡ collected from sequential steps
- Interference
  - ➡ apply update patterns to the views from X
  - ➡ requires matching to check applicability of update pattern

Thread-Modular Interference cont.

Merge-and-project approach [Abdulla et al. TACAS'13]

for every pair of views v_1 and v_2 from X

1. a merged view is created
   - ➡ requires matching to check compatibility
   - ➡ relates thread-local state
2. the thread from v_2 executes a step
3. the result is projected to the thread of v_1

Thread-Modular Interference cont.

Merge-and-project approach [Abdulla et al. TACAS'13]

for every pair of views v_1 and v_2 from X

1. a merged view is created
   - ➡ requires matching to check compatibility
   - ➡ relates thread-local state
2. the thread from v_2 executes a step
3. the result is projected to the thread of v_1

For explicit memory management requires:

- two threads per view [Abdulla et al. TACAS'13]
- tailored ownership [Haziza et al. VMCAI'16]

Thread-Modular Interference cont.

Merge-and-project approach [Abdulla et al. TACAS'13]

✘ **successful, but scales poorly**

for every pair of views v_1 and v_2 from X

1. a merged view is created
   - ➡ requires matching to check compatibility
   - ➡ relates thread-local state
2. the thread from v_2 executes a step
3. the result is projected to the thread of v_1

For explicit memory management requires:

- two threads per view [Abdulla et al. TACAS'13]
- tailored ownership [Haziza et al. VMCAI'16]

Contribution

- Interference by an *effect summary*
  - ➡ *linear* in X
  - ➡ *no* matching/merging
- *Effect* = update of the shared heap
- *Effect summary*
  - ➡ *stateless* sequential program
  - ➡ *over-approximation* of the effects of the program to be verified

Contribution

- Interference by an *effect summary*
  - ➡ *linear* in X
  - ➡ *no* matching/merging
- *Effect* = update of the shared heap
- *Effect summary*
  - ➡ *stateless* sequential program
  - ➡ *over-approximation* of the effects of the program to be verified

Road Map:

- Statelessness
- Interference in more detail
- How to compute a summary

Statelessness

- Atomic execution
- Absence of local state
  - starts with empty local state
    - ➡ independent of execution history
    - ➡ behavior determined solely by shared heap
  - terminates with empty local state
    - ➡ disposes local state

Statelessness

- Atomic execution
- Absence of local state
  - starts with empty local state
    - ➡ independent of execution history
    - ➡ behavior determined solely by shared heap
  - terminates with empty local state
    - ➡ disposes local state

$$Q \ \ stateless$$

$$\Longrightarrow \ \prod^{\infty} Q = Q^*$$

New Interference

- Thread-modular

$$X = X \cup sequential(X) \cup interference(X)$$

- Interference by summary
    - on every view in X execute the summary
        - ➡ corresponds to analyzing $T \parallel Q^*$
    - no matching/merging required
        - ➡ summary has no state which needs to be related

New Interference

- Thread-modular

$$X = X \cup sequential(X) \cup interference(X)$$

- Interference by summary ✓ **linear in $\mathbf{X}$**
  - on every view in X execute the summary
    - ➡ corresponds to analyzing $T \parallel Q^*$
  - no matching/merging required
    - ➡ summary has no state which needs to be related

Computing an Effect Summary

Copy-and-check blocks

- ➡ widespread programming pattern
- ➡ updates a shared value
  1. copy the shared value
  2. perform computation over it
  3. update the shared value if unchanged since copy, retry otherwise

Computing an Effect Summary

Copy-and-check blocks

- ➡ widespread programming pattern
- ➡ updates a shared value

1. copy the shared value
2. perform computation over it
3. update the shared value if unchanged since copy, retry otherwise

Typical implementation

```
while (true)
    x = X;
    n = ...;
    if (CAS(X, x, n))
        break;
```

Computing an Effect Summary cont.

- Assuming *atomicity* of copy-and-check blocks
  - ➡ potentially unsound
  - ➡ a good heuristic (the programmers intent)
- Effect summary = choice over all copy-and-check blocks in the program
- Ensure soundness by a check *on top* of thread-modular fixed point

Soundness Check

- For every view v_1 in X

  (a) perform a sequential step for v_1

  (b) apply the summary to v_1

- Check that

  ➡ effects from (a) are included in the effects from (b)

  ➡ in (b) summary disposes local state

Soundness Check

- For every view v_1 in X

  (a) perform a sequential step for v_1

  (b) apply the summary to v_1

- Check that

  ➡ effects from (a) are included in the effects from (b)

  ➡ in (b) summary disposes local state

Check works on top of potentially unsound fixed point solution X

Soundness Check

- For every view v_1 in X ✔ **linear in $\mathbf{X}$**

  (a) perform a sequential step for v_1

  (b) apply the summary to v_1

Check works on top of potentially unsound fixed point solution X

- Check that
  - ➡ effects from (a) are included in the effects from (b)
  - ➡ in (b) summary disposes local state

Summary of our Approach

Guess&Check framework

1. guess effect summary of program
2. state space exploration
   - ➡ thread-modular fixed point
   - ➡ interference by summary
3. soundness check

Experiments

- Implemented C++ prototype
  - ➡ Abdulla et al. [TACAS'13]
  - ➡ Haziza et al. [VMCAI'16]
  - ➡ guess&check analysis
- Check linearizability of lock-free data structures
- Analyses for GC and MM
- Open source

Experiments: GC

	classical		summaries
Coarse Stack	0.29s	↓ :10	0.03s
Coarse Queue	0.49s	↓ :10	0.05s
Treiber's stack	1.99s	↓ :33	0.06s
Michael&Scott's queue	11.0s	↓ :28	0.39s
DGLM queue	9.56s	↓ :25	0.37s

Experiments: MM

	classical		summaries
Coarse Stack	1.89s	↓ :10	0.19s
Coarse Queue	2.34s	↓ :2	0.98s
Treiber's stack	25.5s	↓ :15	1.64s
Michael&Scott's queue	11700s	↓ :114	102s
DGLM queue	false-positive	✗	violation

Explicit Memory Management

- Problem: explicit frees
    - ➡ target memory unreachable from shared variables
    - ➡ cannot be mimicked by stateless summary
- Solution: ownership transfer
    - ➡ breaking reachability from shared variables grants ownership
    - ➡ stateless summary can free immediately after gaining ownership
- Future work: relax statelessness

Thanks.