Domain-Independent Multi-threaded Software Model Checking

Dirk Beyer and Karlheinz Friedberger

LMU Munich, Germany

25 Sept 2018, Moscow
Software Verification

C Program

```c
int main() {
    int a = foo();
    int b = bar(a);
    assert(a == b);
}
```

Verification Tool

TRUE
i.e., specification is satisfied

FALSE
i.e., bug found

General method:
Create an overapproximation of the program states /
compute program invariants
Basic Challenges with Software Verification

- more domains
- larger tasks
- bigger hardware
Basic Challenges with Software Verification

configurable program analysis

more domains

larger tasks

bigger hardware
Basic Challenges with Software Verification

- More domains
- Bigger hardware
- Larger tasks
- Configurable program analysis
- Abstraction
- Caching
- Block summaries
Basic Challenges with Software Verification

- Configurable program analysis
- More domains
- Abstraction
- Larger tasks
- Caching
- Block summaries
- Bigger hardware
- Portfolio
- Parallel analysis?
- Scheduling?
- Synchronization?
Configurable Program Analysis (CPA)
[Beyer/Henzinger/Théoduloz, 2007]

- CPA algorithm computes a fixed-point based on two sets of abstract states
  - *reached*: already analyzed abstract states
  - *waitlist*: frontier states to be analyzed
Configurable Program Analysis (CPA)
[Beyer/Henzinger/Théoduloz, 2007]

- CPA algorithm computes a fixed-point based on two sets of abstract states
  - *reached*: already analyzed abstract states
  - *waitlist*: frontier states to be analyzed
- Operators defined for specific domain:
  - *transfer*: successor computation
  - *merge*: combination of two abstract states
  - *stop*: coverage of abstract states
Configurable Program Analysis (CPA)
[Beyer/Henzinger/Théoduloz, 2007]

- CPA algorithm computes a fixed-point based on two sets of abstract states
  - reached: already analyzed abstract states
  - waitlist: frontier states to be analyzed
- Operators defined for specific domain:
  - transfer: successor computation
  - merge: combination of two abstract states
  - stop: coverage of abstract states

✓ Independent of used domain
Configurable Program Analysis (CPA)
[Beyer/Henzinger/Théoduloz, 2007]

- CPA algorithm computes a fixed-point based on two sets of abstract states
  - *reached*: already analyzed abstract states
  - *waitlist*: frontier states to be analyzed
- Operators defined for specific domain:
  - *transfer*: successor computation
  - *merge*: combination of two abstract states
  - *stop*: coverage of abstract states

✓ Independent of used domain
X Operators strictly sequential (per analysis!)
Schematic Example of an Analysis
Schematic Example of an Analysis

A plain analysis
Block Summarization

- Block-abstraction memoization (BAM) defined as CPA [Wonisch/Wehrheim, 2012]
- Split large verification task into smaller problems and solve them separately
- Use CPA algorithm for a domain-specific analysis
- Cache intermediate analysis results
Block Summarization

- Block-abstraction memoization (BAM) defined as CPA [Wonisch/Wehrheim, 2012]
- Split large verification task into smaller problems and solve them separately
- Use CPA algorithm for a domain-specific analysis
- Cache intermediate analysis results

☑ Independent of domain-specific analysis
✚ Nearly independent analyses for blocks
Schematic Example of an Analysis

A plain analysis

Time
Schematic Example of an Analysis

plain analysis

A

time
Schematic Example of an Analysis

plain analysis

BAM
Parallel Block-Abstraction Memoization

Our contribution: Parallel BAM [Proc. ASE 2018]

- Continue with CPA algorithm (non-empty waitlist!) while asynchronously computing block abstractions
- Lazy application of computed block abstractions
- Simple scheduler
Parallel Block-Abstraction Memoization

Our contribution: Parallel BAM [Proc. ASE 2018]

▶ Continue with CPA algorithm (non-empty waitlist!) while asynchronously computing block abstractions
▶ Lazy application of computed block abstractions
▶ Simple scheduler

✓ Combines benefits of existing approaches
Parallel Block-Abstraction Memoization

Our contribution: Parallel BAM [Proc. ASE 2018]

- Continue with CPA algorithm (non-empty waitlist!) while asynchronously computing block abstractions
- Lazy application of computed block abstractions
- Simple scheduler

✔ Combines benefits of existing approaches
✔ Small synchronization overhead (depends on block size)
Schematic Example of an Analysis

plain analysis

BAM
Schematic Example of an Analysis

plain analysis

BAM

parallel BAM
Evaluation

- **Configuration**
  - CPAchecker r28809
  - Explicit Value Domain

- **Environment**
  - Intel Xeon E3-1230 v5 CPU with 4 physical cores
  - 5400 tasks from SV-COMP benchmark set

- **Limitations**
  - 15 GB RAM
  - 15 minutes
Evaluation

![Graph showing wall time in seconds against the n-th fastest result for different numbers of threads (1, 2, 4, 8).]
Evaluation

The graphs show the relationship between the number of threads and the wall time (s) for the n-th fastest result. The graphs are labeled as follows:

- **1 (number of threads)**
- **2**
- **4**
- **8**

The x-axis represents the n-th fastest result, while the y-axis represents the wall time in seconds. The graphs indicate how the wall time changes as the number of threads increases. The inset graph provides a closer view of the trend at higher numbers of threads.

Dirk Beyer and Karlheinz Friedberger, LMU Munich, Germany
Conclusion

- Small overhead for synchronization in parallel analysis
- Elegant integration into the framework CPAchecker
- No changes necessary to existing analyses and concepts
  - Small refactoring on implementation if necessary
  - CEGAR, proof and counterexample witnesses
Hints for developers

CPA operators are applied in parallel (on different reached sets and waitlists)

- CPA operators should be *stateless*
- Caches should allow shared access or only be used in *one instance* of an operator
Hints for developers

CPA operators are applied in parallel
(on different reached sets and waitlists)

▶ CPA operators should be *stateless*
▶ Caches should allow shared access or
  only be used in *one instance* of an operator

Abstract states should be immutable after construction
▶ Guarantee for the developer: no side effects
Hints for developers

CPA operators are applied in parallel (on different reached sets and waitlists)

▶ CPA operators should be *stateless*
▶ Caches should allow shared access or only be used in *one instance* of an operator

Abstract states should be immutable after construction

▶ Guarantee for the developer: no side effects

Statistics are *data*!

▶ Often shared across several components
▶ Allow concurrent access!
Future work

- Scheduling/iteration order:
  - Prefer parts deeper in the program
  - Depending on machine load?
- Processes instead of threads
  - Cluster instead multi-core machine
  - Easier handling of external libraries
- Support more domains
  - Mostly simple refactoring, only a few *hard* changes
  - Dependencies, e.g., on external libraries like SMT solvers
Questions? Discussion?
CPAchecker Framework

Source Code $\rightarrow$ Parser & CFA Builder $\rightarrow$ CPA Algorithm $\rightarrow$ Results

Spec $\rightarrow$ Spec CPA $\rightarrow$ Location CPA $\rightarrow$ Callstack CPA $\rightarrow$ Predicate CPA

CPA Location CPA

CPA Callstack CPA

CPA Predicate CPA
BAM in CPAchecker

Source Code → Parser & CFA Builder → CEGAR Algorithm → Results

Spec → Spec CPA, Location CPA, Callstack CPA, Predicate CPA

CPA Algorithm

wait for nested analysis

BAM CPA

Dirk Beyer and Karlheinz Friedberger
LMU Munich, Germany
Parallel BAM in CPAchecker

- Source Code
- Parser & CFA Builder
- CEGAR Algorithm
- Parallel BAM Algorithm
- CPA Algorithm
- Parallel BAM CPA
  - Spec CPA
  - Location CPA
  - Callstack CPA
  - Value CPA
- Results

enqueue missing block abstraction