A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker

ThreadingCPA

Dirk Beyer[1] Karlheinz Friedberger[2]

[1]LMU Munich, Germany

[2]University of Passau, Germany

Multi-Threaded Programs on the Rise

Why do we need multi-threaded programs? Where do we use them?

Multi-threaded programs appear everywhere!

- several threads per CPU core
- multi-core CPUs
- Linux kernel, device drivers
- internet, web and cloud services, IoT

  ...
- SV-Comp: special category for concurrent programs

Several approaches available:

- direct analysis of all thread interleavings
- program sequentialization
- formula-based encoding of threads

A Very Old Problem

Several approaches available:

- direct analysis of all thread interleavings
- program sequentialization
- formula-based encoding of threads

Combined with some optimization:

- partial order reduction (ample sets, ...)
- iteration order for state-space exploration
- bounded model checking (bounded number of threads, ...)

Several approaches already available in CPACHECKER:
(all of them are based on the *pthreads* library)

- formula-based encoding with predicate analysis
  $\rightarrow$ very old orphaned branch
- sequentialization of the CFA
  $\rightarrow$ student's thesis, needs some work
- ThreadingCPA: handles program locations for multiple threads
  $\rightarrow$ replaces LocationCPA
  $\rightarrow$ everything else should work out-of-the-box (really?)

CFA control flow automaton with location nodes (program counter) and edges (statements and assumptions),
one CFA per function,
all function connected into super-graph of program

Basics

What every developer of CPACHECKER already knows

CFA
: control flow automaton with location nodes (program counter) and edges (statements and assumptions),
one CFA per function,
all function connected into super-graph of program

CPA
: abstract domain: how does an abstract state look alike?
transfer relation: how to handle a single edge?
merge and stop operator: how are abstract states related?

ThreadingCPA
... just another CPA

LocationCPA: one program location per abstract state

ThreadingCPA

... just another CPA

LocationCPA: one program location per abstract state

Basic idea: track *many* instead of *one* program locations

ThreadingCPA
... just another CPA

LocationCPA: one program location per abstract state

Basic idea: track *many* instead of *one* program locations

abstract state: $\{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, \ldots\}$
transfer relation: $s \overset{g}{\rightsquigarrow} s'$ depends on the edge g:

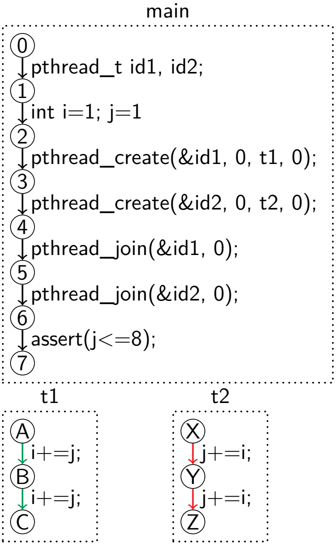
1. *pthread_create*: add a new location for the new thread
2. *pthread_join*: remove the exit location of the joined thread
3. otherwise: just analyze the edge (like LocationCPA, with additional handling of pthread locks)

merge and stop operator: based on equality of abstract states ($merge_{sep}$ and $stop_{sep}$)

→ can be combined with other CPAs

Example

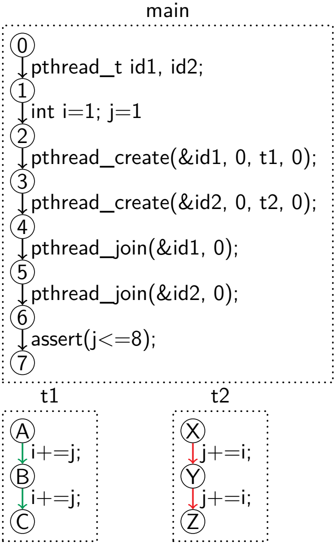
Program with CFA

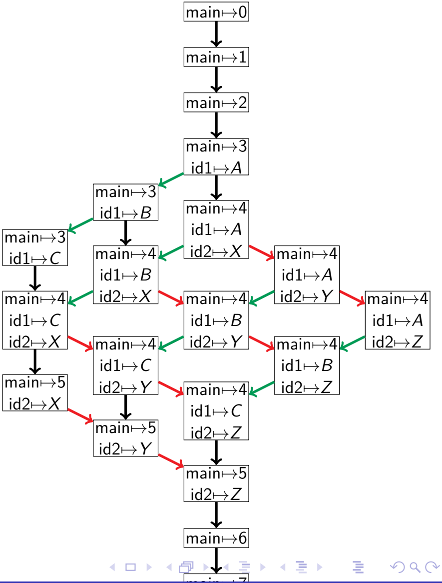



```
pthread_t id1, id2;
int i=1, j=1;

void main() {
  pthread_create(&id1, 0, t1, 0);
  pthread_create(&id2, 0, t2, 0);

  pthread_join(id1, 0);
  pthread_join(id2, 0);

  assert(j <= 8);
}

void t1() {
  i+=j;
  i+=j;
}

void t2() {
  j+=i;
  j+=i;
}
```

Example

CFA and ARG





We have to handle several **call stacks**, one per thread
→ integrate CallstackCPA into ThreadingCPA

We have to handle several **call stacks**, one per thread
→ integrate CallstackCPA into ThreadingCPA

ValueCPA, BDDCPA, IntervalCPA:
→ track assignments, identify variables as $f::x$
→ problem: same function called in several threads?
→ solution: avoid colliding function names by *cloning* each function before the analysis

We have to handle several **call stacks**, one per thread
$\rightarrow$ integrate CallstackCPA into ThreadingCPA

ValueCPA, BDDCPA, IntervalCPA:
$\rightarrow$ track assignments, identify variables as $f::x$
$\rightarrow$ problem: same function called in several threads?
$\rightarrow$ solution: avoid colliding function names by *cloning* each function before the analysis

Other CPAs and algorithms: TODO
$\rightarrow$ some small changes required (several locations per state)
$\rightarrow$ PredicateCPA: block operator matches thread interleavings?
$\rightarrow$ more advanced thread management

Optimization for the ThreadingCPA

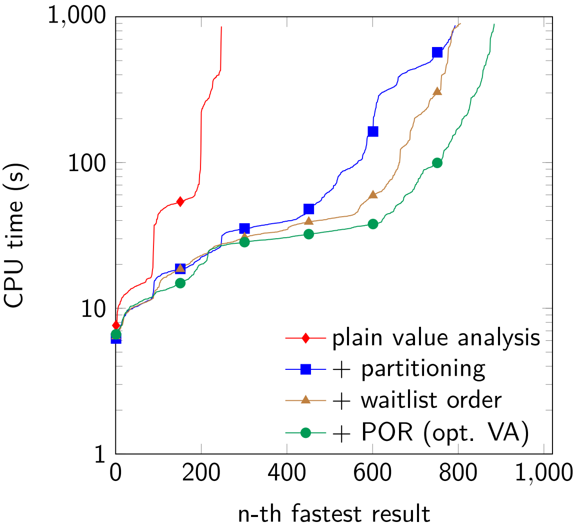
Is this simple approach efficient? Not yet!

We need optimization!

- partial order reduction
  → implemented in ThreadingCPA
- bound number of threads
  → implemented in ThreadingCPA
- iteration order
  → implemented as *waitlist order*, like BFS and DFS
- partitioning abstract states based on program location
  → inherit from *Partitionable* and use *PartitionedReachedSet*
- equality for call stack states with different object identities
  ! CPAchecker does not use equality for call stacks by default !

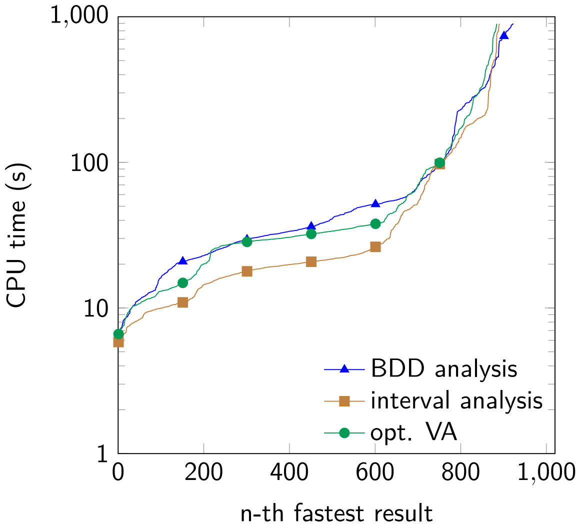
Evaluation on the Category "Concurrency", SV-Comp'16

Value Analysis with Optimization Steps



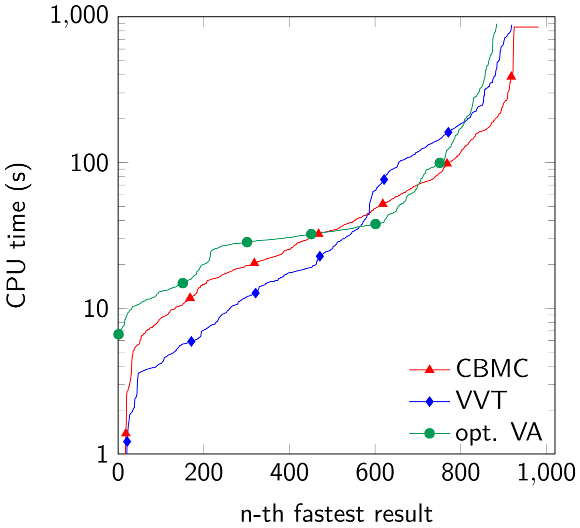
Evaluation on the Category "Concurrency", SV-Comp'16

Different analyses in CPAchecker



Evaluation on the Category "Concurrency", SV-Comp'16

Comparison of CPAchecker with other tools



Further Possibilites

CPAchecker is very flexible

Validation Witnesses:

- export counterexamples in Graphml
- extension of the format: include identifiers for threads

Further Possibilites

CPAchecker is very flexible

Validation Witnesses:

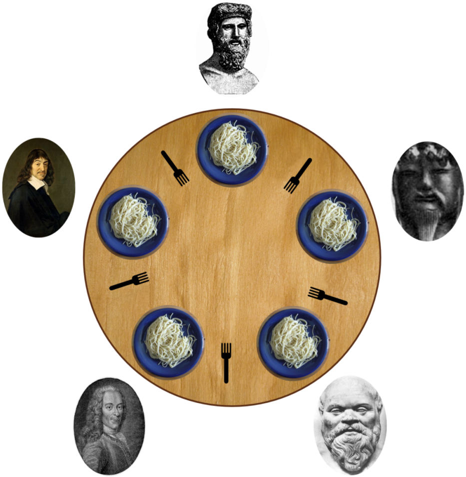
- export counterexamples in Graphml
- extension of the format: include identifiers for threads

Deadlock detection:

- for the user: just change the specification
- detail: the *strengthening* operator allows to inform the AutomatonCPA about deadlock found by the ThreadingCPA

Dining Philosophers Problem

Questions before Dinner?



⟳: Plato, Konfuzius, Socrates, Voltaire and Descartes