# Block-Abstraction Memoization with CEGAR (In-Place vs. Copy-On-Write Refinement)

SoSy-Lab Software Systems

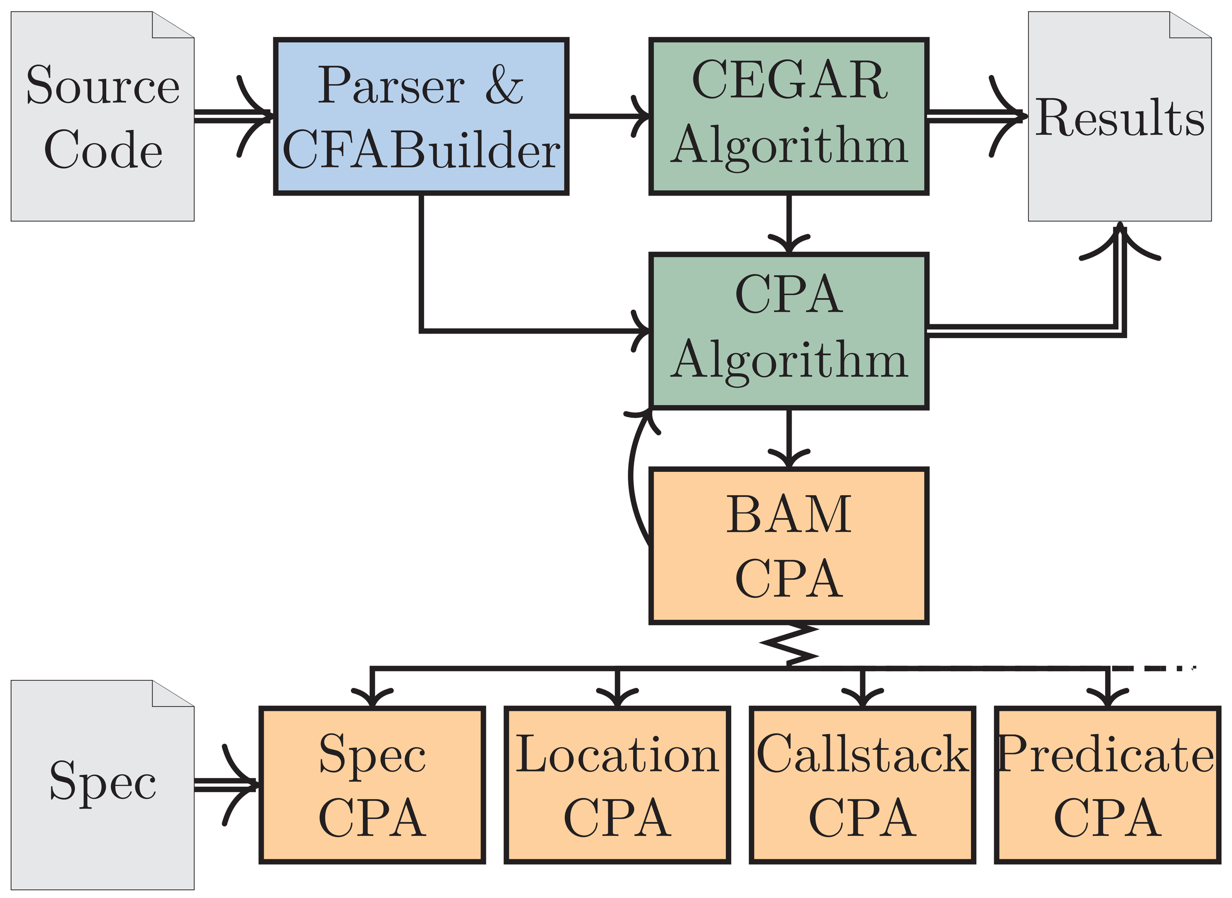
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Karlheinz Friedberger

## Overview

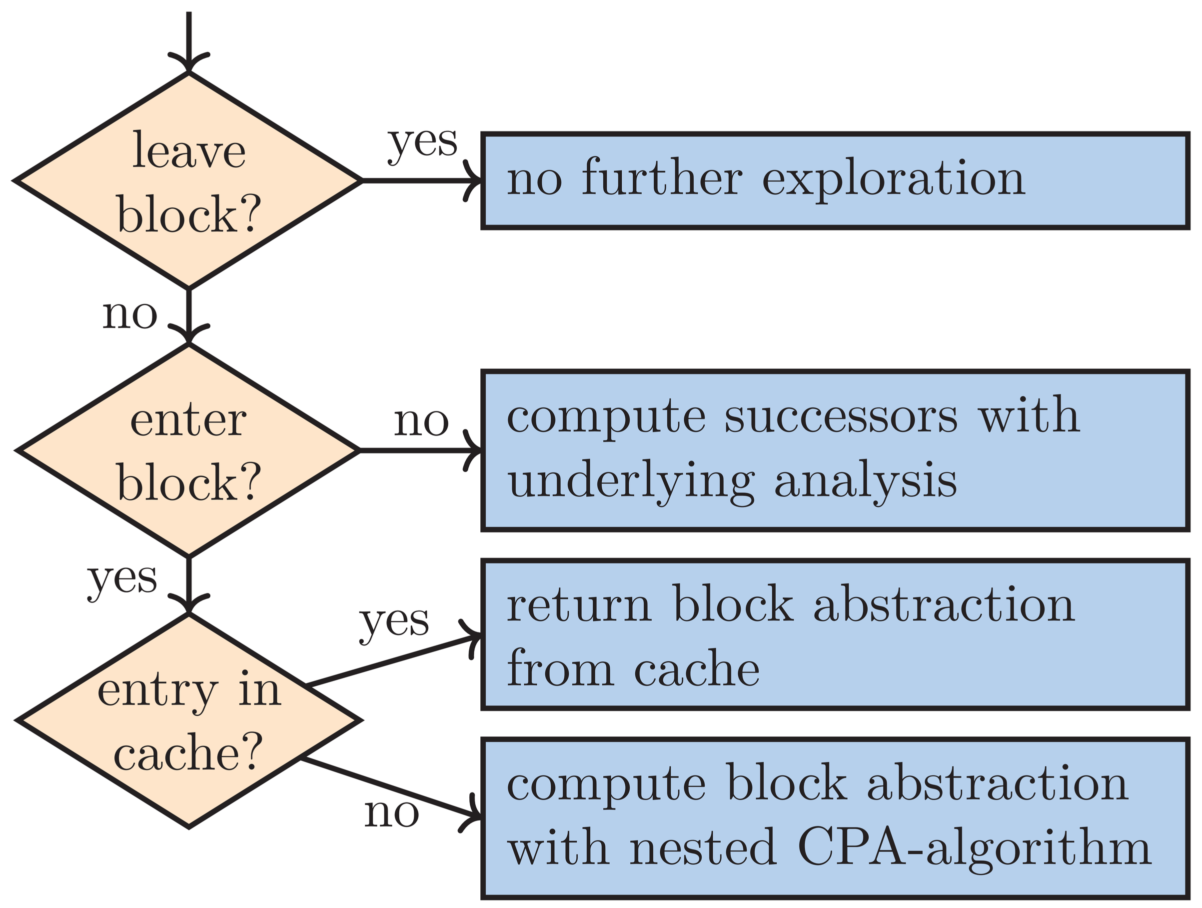
Block-abstraction memoization (BAM) [5] is a technique for software verification that aims towards a modular scalable analysis for large programs.
It is based on common concepts like
- configurable program analysis (CPA) [1] and
- caching and information reuse.

BAM is independent of the underlying analysis and can be used in combination with
- predicate abstraction [2]
- explicit-state model checking [3]
- BDD-based software verification [4]




## Control Flow of BAM

BAM computes new states for the state space based on blocks, the cache, and the underlying analysis.




## State-Space Exploration

Basic steps of BAM:
- program is divided into blocks (functions or loops)
- nested CPA algorithm explores and analyzes the state space of each block
- block abstractions are cached for reuse

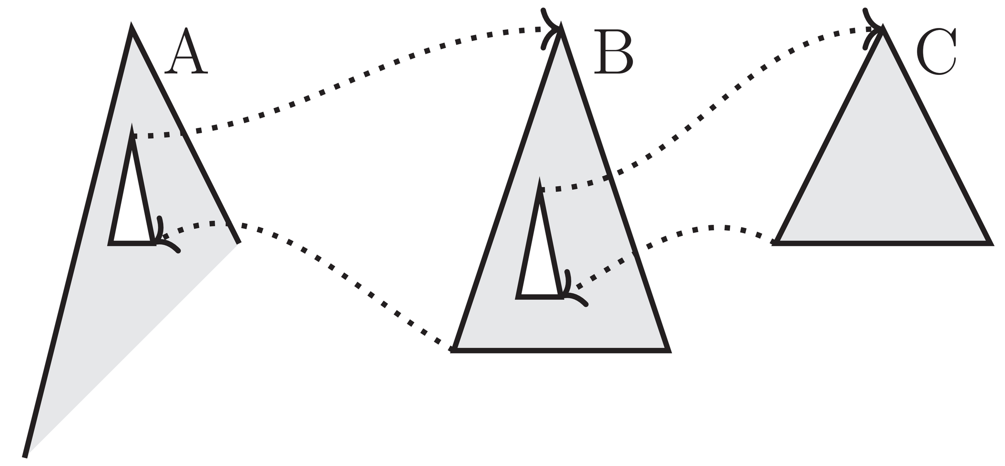



**Figure 1:** Block A is analyzed, nested blocks B and C already finished

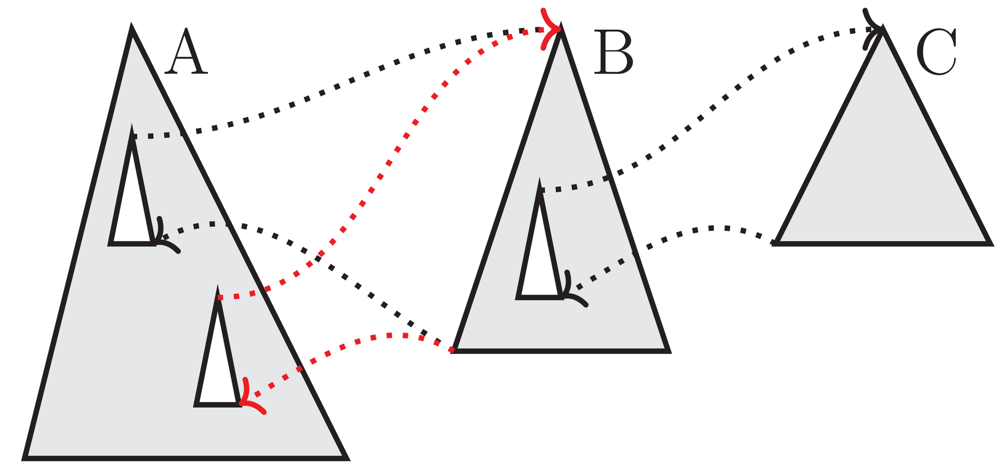



**Figure 2:** Block abstraction for state space B is reused from cache

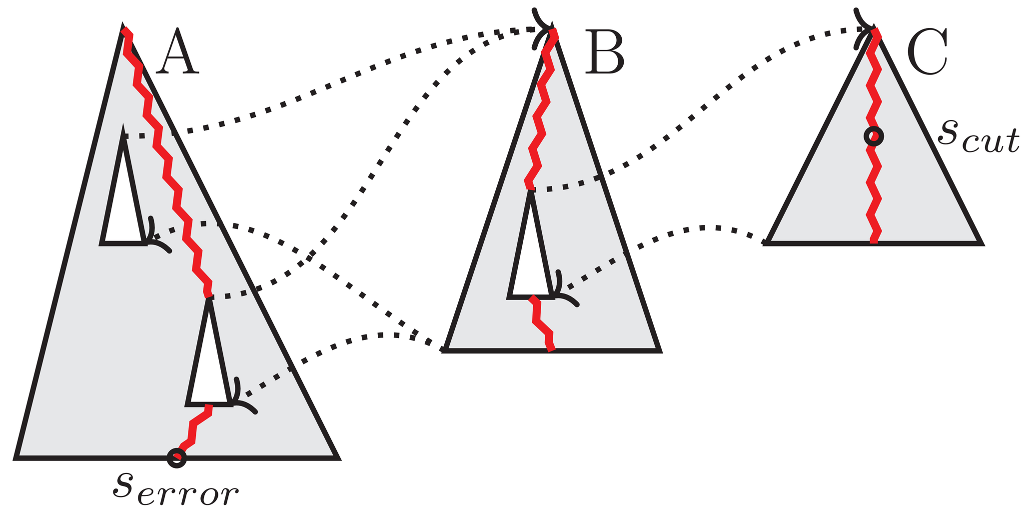


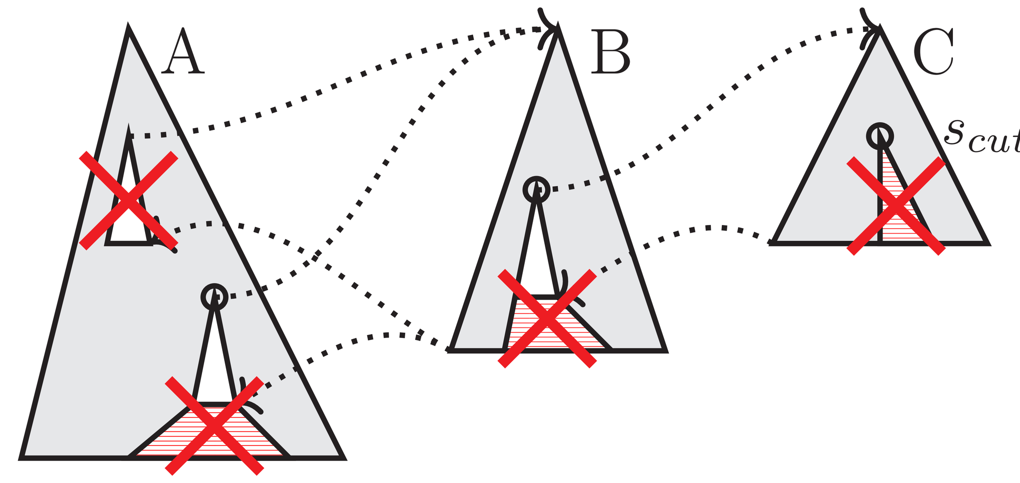

**Figure 3:** Finding a (spurious) counterexample to an error state $s_{error}$ and determining a cutpoint $s_{cut}$ for the refinement

## Optimization and Heuristics

- *Reducer*: hide unnecessary information in states to increase cache hit rate
- *Aggressive caching*: over-approximate entries when accessing the cache
- *Refinement strategies*: refine *one*, *some*, or *all* states along a counterexample trace

## In-Place Refinement

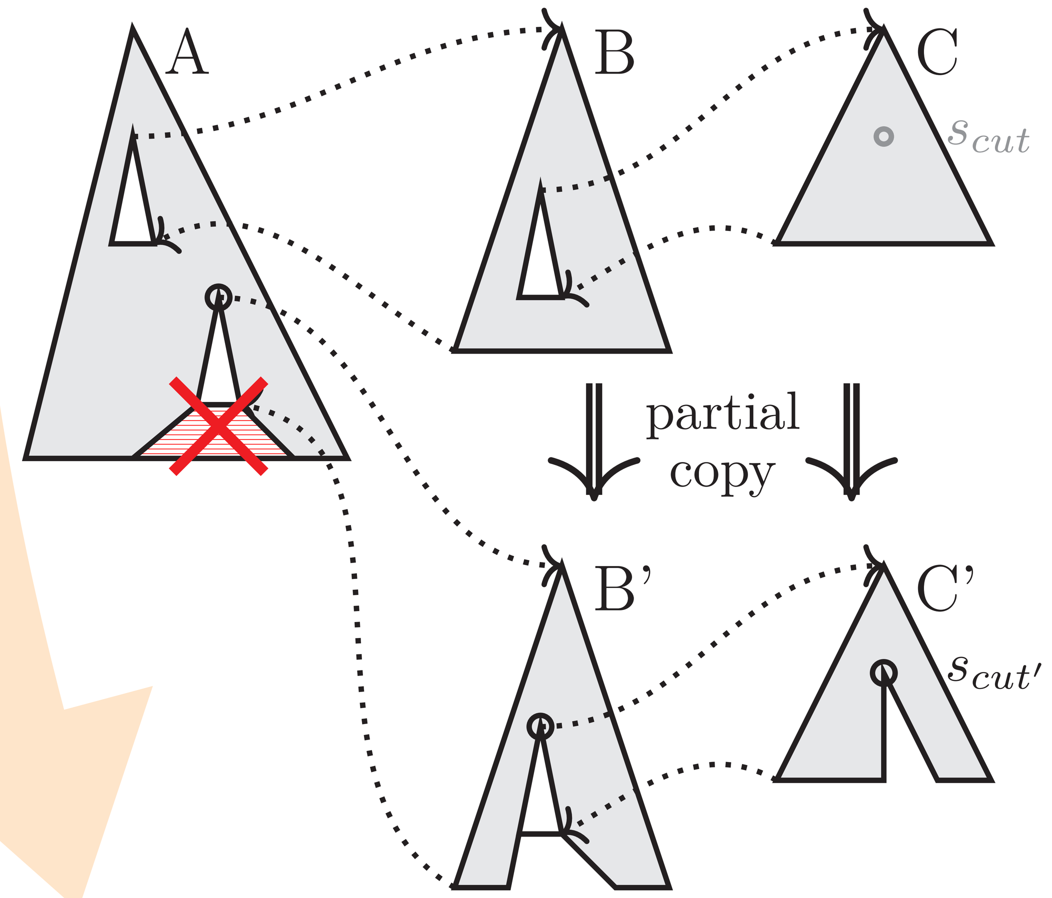
- change existing block abstractions
- remove several parts of the reached state space




Problems:
- contradicts the idea of *lazy refinement* by updating or deleting too many states
- overhead for recomputing previously deleted block abstractions
- recomputation can lead to repeated counterexamples due to information loss

## Copy-On-Write Refinement

- invalidate only some reached states
- use copies of existing block abstractions




Benefits:
- correct usage of *lazy refinement* by updating or deleting only necessary parts
- all important data remains in the cache
- better for programs with more refinements

## Evaluation

For simple tasks with only zero or one refinements both approaches behave identical. For difficult tasks that need more refinements (and thus more runtime) the *copy-on-write* approach shows its benefit over the *in-place* approach.

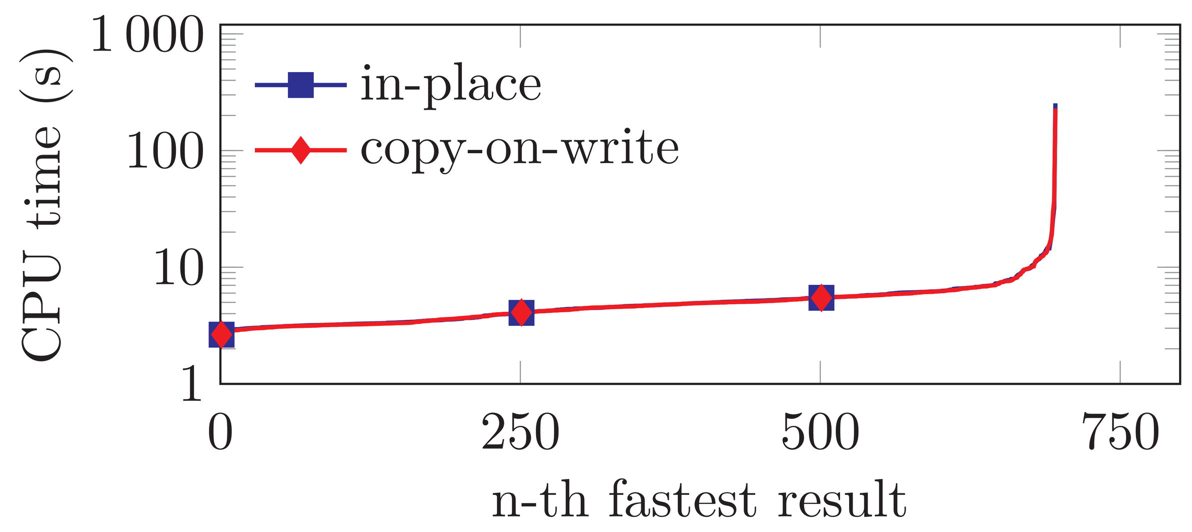



**Figure 4:** Quantile plot of BAM with predicate analysis, results with less than two refinements

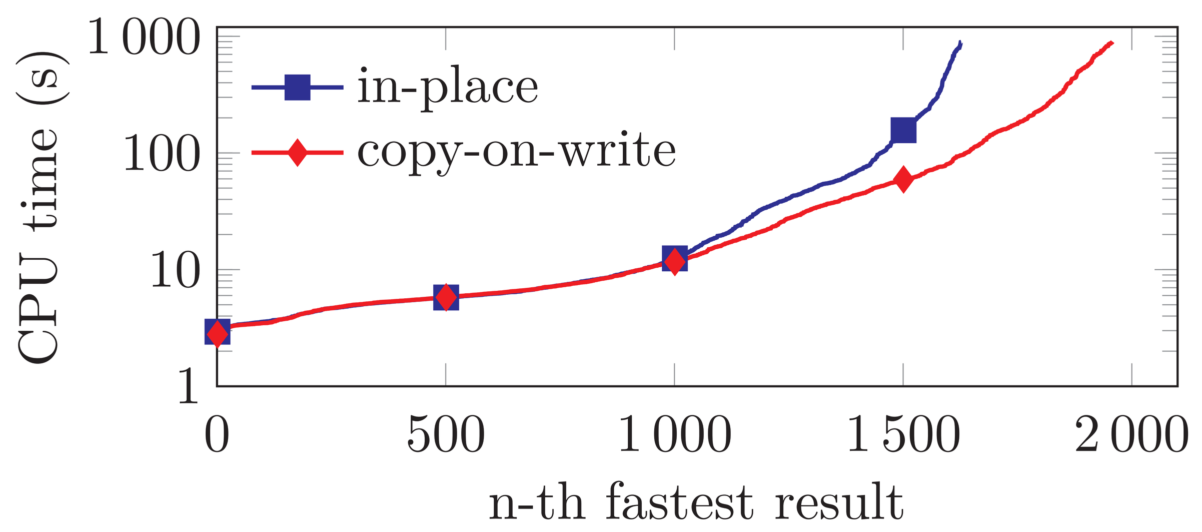



**Figure 5:** Quantile plot of BAM with predicate analysis, results with at least two refinements

## References


[1] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.

[2] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[3] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

[4] D. Beyer and A. Stahlbauer. BDD-based software model checking with CPAchecker. In *Proc. MEMICS*, LNCS 7721, pages 1–11. Springer, 2013.

[5] K. Friedberger. CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 912–915. Springer, 2016.