Domain-Independent Multi-threaded Software Model Checking

Dirk Beyer
LMU Munich
Germany

Karlheinz Friedberger
LMU Munich
Germany

ABSTRACT

Recent development of software aims at massively parallel execution, because of the trend to increase the number of processing units per CPU socket. But many approaches for program analysis are not designed to benefit from a multi-threaded execution and lack support to utilize multi-core computers. Rewriting existing algorithms is difficult and error-prone, and the design of new parallel algorithms also has limitations. An orthogonal problem is the granularity: computing each successor state in parallel seems too fine-grained, so the open question is to find the right structural level for parallel execution. We propose an elegant solution to these problems: Block summaries should be computed in parallel. Many successful approaches to software verification are based on summaries of control-flow blocks, large blocks, or function bodies. Block-abstraction memoization is a successful domain-independent approach for summary-based program analysis. We redesigned the verification approach of block-abstraction memoization starting from its original recursive definition, such that it can run in a parallel manner for utilizing the available computation resources without losing its advantages of being independent from a certain abstract domain. We present an implementation of our new approach for multi-core shared-memory machines. The experimental evaluation shows that our summary-based approach has no significant overhead compared to the existing sequential approach and that it has a significant speedup when using multi-threading.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Parallel algorithms*;

KEYWORDS

Program Analysis, Software Verification, Parallel Algorithm, Multi-threading, Block-Abstraction Memoization

ACM Reference Format:
Dirk Beyer and Karlheinz Friedberger. 2018. Domain-Independent Multi-threaded Software Model Checking. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238195

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238195

1 INTRODUCTION

Program verification has been applied successfully to find errors in applications or to prove their correctness. Recent hardware development aims towards parallel execution of programs either on multi-core machines or shared across several machines in a computing cluster. For large-scale program verification, we do not only need efficient algorithms, but also make use of available hardware resources up to their limits. There are some approaches to leverage such systems, but most recent algorithms for program verification and model checking are not designed to work in parallel manner and utilize only a small part of available resources. There are several reasons for this: Either the verification algorithms have dependencies between intermediate results, such that only a sequential execution is useful, or the amount of parallelism is bound by a small number, e.g., only two analyses are executed in parallel and communicate information, effectively using only a small number of CPU cores. The main question is whether and how we can (re-)design existing verification techniques such that they can be executed on parallel computer architectures.

We contribute the idea to use summaries as the objects to compute in parallel, instead of inventing new parallel state-space iteration algorithms. Block-abstraction memoization (BAM) [31] is a particularly nice method to summarize blocks of program statements, because it is independent from a particular analysis — it wraps an existing analysis without much interference and stores block summaries in a cache. We use this concept to develop a domain-independent analysis that distributes a verification problem across multiple processing units without changes to the analysis technique. Our analysis is based on a standard state-space exploration using a control-flow automaton that represents the program. The approach is orthogonal to other data-flow-based analyses, and thus, it can be combined with analyses based on different abstract domains like BDDs, explicit values, intervals, or predicates.

The value of our approach is its level of *separation of concerns*: it separates the concern of making an analysis multi-threaded from the concern of designing and implementing an abstract domain and its operators. We base our approach on BAM and use most of its data structures, such that most parts of the (wrapped) analysis (and its implementation) remain unchanged. We redesigned the algorithm such that we can efficiently execute it across several processing units. The parallelism of the analysis is only bound by the structure of the program to be analyzed and the amount of work found during the analysis. Our work includes a transformation of the existing algorithm of BAM from a sequential, recursively defined algorithm into a parallel approach. Additionally, we benefit from the existing infrastructure of BAM, i.e., we also use a cache for block abstractions and apply the operators reduce and expand to increase the cache hit rate. The analysis is sound, implemented

in the open-source verification framework CPAchecker, and can be combined with existing components of the framework, including CEGAR [20] or witness export [5, 6].

Contributions. We make the following contributions:

- We introduce a new technique for parallelization of verification algorithms that is independent from particular abstract domains because it is based on a flexible and configurable block summarization.
- We implemented the technique in the open-source verification framework CPAchecker. Our implementation and all experimental data are available to other researchers and practitioners for replication via our artifact [9] and supplementary website.[1]
- We evaluated our new technique on a large set of benchmarks and show (1) that the parallel version of BAM (if using only one CPU core) behaves similar to the sequential version (i.e., there is no significant overhead for parallelization) and (2) that the parallel version of BAM significantly improves the response time of the verification process for programs that are large enough to benefit from multi-threading.

Related Work. The idea to use parallel algorithms in software verification is not new. There exist several approaches reaching from plain parallel execution of different algorithms (until the first analysis succeeds) via one-way communication between (some) analyses (one analysis provides additional information for another one) to fully parallel analyses (dividing the state space into partitions that are explored separately).

Portfolio Approaches. A simple, but effective approach is to run a portfolio analysis [24], i.e., a fixed number of predefined analyses in parallel to leverage the available CPU cores on a single machine, such that the verifier terminates with the first succeeding analysis (e.g. [22, 26]). This strategy is applied either to separately explore the state space, e.g., with different domains, or in a way that one analysis provides information for another one, for example to enrich it with additional invariants [7]. Such approaches for parallel software verification are not scalable due to its fixed number of different analyses, and they suffer from the problem that each single analysis only uses a small fraction of the available resources. If all but one analysis fail to determine a verification result (because of unsupported features in the task, or imprecision of the analysis), the remaining work is sometimes limited to a single analysis and thus a single core.

Multi-Threading Approaches. SPIN [23] and Divine [3, 29] are based on pure explicit model checking and use a central hash table to check for existing (already analyzed) states. LTSmin [18] either performs explicit state-space search in a parallel manner or uses a BDD-based approach using the BDD-library Sylvan [30] that internally parallelizes its operations. Other approaches divide a given problem into smaller components that are verified separately, before joining the results to get a proof for a whole program [21, 25]. An example implementation for such a technique is the tool SoftVer that uses BDDs and predicates.

Structurally-defined conditional analysis [28] is an approach that splits a program according to conditions as in conditional model checking [11], that is, given a program P and a condition ψ, two analysis instances can be created, one conditional analysis of P and ψ and one conditional analysis of P and $\neg\psi$. The two analysis instances are completely independent and can be executed in parallel. The approach can scale up to an arbitrary number of splits. The elegance of this approach is that it does not depend on a specific implementation but can be built on top of existing, off-the-shelf tool components.

Multi-Machine Approaches. State-space exploration can be distributed across several machines by partitioning the possible state-space. Tools like SPIN [23], CSeq-Swarm [27], or the SPARK Analysis Tools [19] divide the verification problem after a short pre-analysis of the program, and split the potential state space and the verification condition according to given time and memory limitations, available processing units, or other criteria. This approach is potentially problematic due to the unknown nature of the program to be analyzed, e.g., it might not match the pre-defined scheduling. For degenerated state spaces, the parallel analysis might be imbalanced between different threads/processes. Other tools like LTSmin [18] or Divine [3, 29] circumvent such imbalances by a dynamic scheduling approach. The approach of structurally-defined conditional analysis [28] can also be extended to benefit from multi-machine environments.

Our contribution is a more general parallel technique for program analysis and can be applied to an arbitrary domain and even combinations of several domains. Thus, explicit-value analysis, BDD-based analysis, as well as predicate analysis can benefit from our approach. The parallelism of the approach presented in this paper is based on the internal structure of the program, i.e., an automatic partitioning of the control flow, and tries to use all available processing units, only depending on the dynamic behavior of the program analysis, i.e., the unfolding of the abstract state space.

2 BACKGROUND

The following section provides an overview of basic concepts and definitions that our approach is based on. We describe the program representation, configurable program analysis, the details of block-abstraction memoization, and how we advanced it towards an efficient parallel algorithm for program analysis (for more detail see the original articles [12, 31]).

2.1 Program Representation

We restrict the presentation to a simple imperative programming language, where all operations are either assignment or assume operations. A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which is a directed graph consisting of a set L of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ of control-flow edges (modeling the computation steps from one location to the next: assignment or assume operations), and an initial program location l_0 (entry point of the program).

2.2 CPA and CPA Algorithm

A *configurable program analysis* (CPA) [12] is specified by an abstract domain for a program analysis and operators to model the behavior of the program analysis: A CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ consists of

[1] https://www.sosy-lab.org/research/bam-parallel/

(1) an abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ that consists of a set C of concrete states, a lattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract-domain elements (i.e., abstract states) and a partial order $\sqsubseteq$, and a concretization function $[\![\cdot]\!]$ that maps each abstract-domain element to the represented set of concrete states.
(2) a transfer relation $\rightsquigarrow \subseteq E \times E$ that yields successors of an abstract state.
(3) a merge operator merge $\subseteq E \times E \to E$ that determines how to merge two abstract states when control flow meets).
(4) a termination check stop $\subseteq E \times 2^E \to \mathbb{B}$ that specifies whether an abstract state is covered by a set of abstract states.

Algorithm 1 CPAalg performs a state-space exploration. It computes an overapproximation of the reachable states by constructing abstract states for the program based on a given CPA and an initial abstract state. The algorithm is a fixed-point iteration and maintains a set waitlist of abstract states that still have to be explored, and a set reached of already explored abstract states. In each iteration, the algorithm takes an abstract state from waitlist (line 2) and computes its successors (line 3). The algorithm checks whether a new state can be merged with an existing state, and updates the work sets accordingly (lines 5–8). The operator stop ensures that the new abstract state is only added to the work sets if the abstract state is not already covered by any of the existing states in reached (lines 9–11). The algorithm terminates if either the set waitlist is empty or there is another reason to abort early, e. g., a property violation.

We use a simplified version of algorithm CPAalg [8] in order to shorten the presentation. The precision and precision adjustment, which determine the granularity of the analysis within a CEGAR loop, are neglected in this description, but fully available and supported in our implementation.

Different aspects of a program are analyzed by different CPAs, and compositions of CPAs allow more advanced analyses. CPAs have been defined for many abstract domains, such as BDD-based analysis [17], (explicit or symbolic) value analysis [14, 15], predicate analysis [8, 10, 13], or combination thereof [2]. Also the tracking of the program counter and of the call stack for procedures are defined as CPAs. We will not go into detail for all their definitions and descriptions here, because our approach works on an abstract level and is independent from a specific domain. For our evaluation later, we use a value analysis that tracks variables and their values explicitly, e.g., an abstract state is a (partial) function that maps program variables to values.

Algorithm 1 CPAalg($\mathbb{D}$, reached, waitlist), taken from [8]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
where E denotes the set of elements of the lattice of D,
a set reached $\subseteq E$ of abstract states,
a set waitlist $\subseteq$ reached of frontier abstract states,
a function abort : $E \to \mathbb{B}$ that defines whether the algorithm should abort early
Output: the updated sets reached and waitlist

```
1:  while waitlist ≠ ∅ do
2:    pop(e) from waitlist
3:    for each e′ with e ⇝ e′ do
4:      for all e″ ∈ reached do
5:        e_new := merge(e′, e″)
6:        if e_new ≠ e″ then
7:          reached := reached ∪ {e_new} \ {e″}
8:          waitlist := waitlist ∪ {e_new} \ {e″}
9:      if ¬stop(e′, reached) then
10:       reached := reached ∪ {e′}
11:       waitlist := waitlist ∪ {e′}
12:       if abort(e′) then
13:         return (reached, waitlist)
14: return (reached, waitlist)
```

2.3 BAM

Block-abstraction memoization (BAM) [31] is a modular approach for reachability analysis of abstract state graphs (such as abstract models of programs). Therefore, it treats a large program as a set of *blocks*, and analyzes the blocks separately. The result of a block analysis (the *block abstraction*) of a nested block is embedded in the surrounding block's analysis. Block abstractions are also stored in a cache for later reuse in order to avoid repeated computation of the same block abstraction, to speed up the analysis. BAM defines the two operators reduce and expand that aim at a higher cache hit rate. For simplicity we will neglect both operators in the further description. They are orthogonal to the approach of parallel analysis that we present here.

The components of BAM are defined in detail in the following:

2.3.1 Blocks. The basic components of BAM are *blocks*, which are formally defined as parts of a program: A block $B = (L', G')$ of a CFA $A = (L, l_0, G)$ consists of a set $L' \subseteq L$ of connected program locations and a set $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ of control-flow edges. Two different blocks $B_1 = (L'_1, G'_1)$ and $B_2 = (L'_2, G'_2)$ are either disjoint ($L'_1 \cap L'_2 = \emptyset$) or one block is completely nested in the other block ($L'_1 \subset L'_2$). Each block $B = (L', G')$ has *entry* and *exit locations*, which are defined as $In(B) = \{l \in L' \mid (\exists (l', op, l) \in G \wedge l' \notin L') \vee \nexists (l', op, l) \in G\}$ and $Out(B) = \{l \in L' \mid (\exists (l, op, l') \in G \wedge l' \notin L') \vee \nexists (l, op, l') \in G\}$, respectively. In general, the block size can be freely chosen in BAM. In most cases, functions and loops are used as block size, because they represent the logical structure of a program and lead to natural block abstractions.

Figure 1 shows a schematic example of a CFA and how it could be divided into blocks. It does not show any operations; we omit details for ease of presentation. The largest block (denoted as B_A) consists of all locations and represents the whole CFA of the program. The other blocks (denoted as B_B to B_F) are smaller and consists of fewer locations. Block B_F is nested in block B_E, which in turn is nested in block B_A. Location 3 is the entry location of block B_B, i.e., $In(B_B) = \{3\}$, and location 4 is its exit location, i.e., $Out(B_B) = \{4\}$.

2.3.2 BAM-CPA. The basis of CPAchecker is the idea of configurable program analysis. Thus, BAM is formalized as a CPA $\mathbb{BAM} = (D_{\mathbb{BAM}}, \rightsquigarrow_{\mathbb{BAM}}, \text{merge}_{\mathbb{BAM}}, \text{stop}_{\mathbb{BAM}})$. BAM works on an abstract, domain-independent level and uses an abstract-domain-dependent wrapped analysis (like the BDD-based, explicit value, interval, or predicate analysis) to track variables and values. This wrapped analysis is also given as CPA $\mathbb{W} = (D_{\mathbb{W}}, \rightsquigarrow_{\mathbb{W}}, \text{merge}_{\mathbb{W}}, \text{stop}_{\mathbb{W}})$, based on which we now formalize BAM:

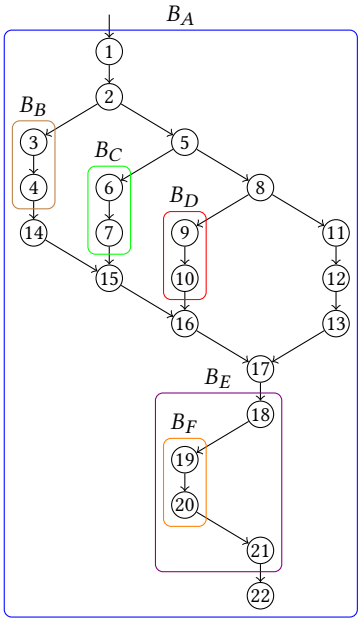


Figure 1: Schematic control-flow automaton with blocks

(1) The domain $D_{\mathbb{BAM}}$ wraps the domain $D_{\mathbb{W}}$.
(2) The transfer relation $\rightsquigarrow_{\mathbb{BAM}}$ for a block B has a transfer $s \rightsquigarrow_{\mathbb{BAM}} s'$ for two abstract states s and s' if

$$s' \in \begin{cases} \{s'' \mid s \overset{B_{sub}}{\rightsquigarrow}_{\mathbb{BAM}} s''\} & \text{if } l \in In(B_{sub}) \text{ // apply BAM to } B_{sub} \\ \{s'' \mid s \rightsquigarrow_{\mathbb{W}} s''\} & \text{if } l \notin Out(B) \text{ // delegate to } \mathbb{W} \end{cases}$$

where l is the program location of s.
Depending on the currently analyzed program location l, the transfer relation chooses between two possible steps: For an entry location of a block B_{sub}, the operation $\overset{B_{sub}}{\rightsquigarrow}_{\mathbb{BAM}}$ represents the block abstraction for the block B_{sub} and the block-entry abstract state s. The block abstraction is computed by a call CPAalg($D_{\mathbb{BAM}}$, $\{s\}$, $\{s\}$). For exit locations of blocks, there is no succeeding abstract state (in the analysis of the current block B). For other program locations, the wrapped transfer relation $\rightsquigarrow_{\mathbb{W}}$ is applied.
(3) The merge operator $\text{merge}_{\mathbb{BAM}} = \text{merge}_{\mathbb{W}}$ and the termination check $\text{stop}_{\mathbb{BAM}} = \text{stop}_{\mathbb{W}}$ correspond to the wrapped analysis.

The performance of BAM can easily be increased by a cache cache $\subseteq$ $(Blocks \times E) \rightarrow (2^E \times 2^E)$, which maps a block and an entry abstract state of the block to the set of reached abstract states and the set of frontier states. The cache is optional for the application of BAM, but the memoization of block abstractions improves the performance. Additionally, the operators reduce and expand can be applied for a higher cache hit rate. We ignore them for simplicity.

2.4 Towards Parallel BAM

A simple state-space exploration that enumerates all reachable abstract states and only checks whether they were already part of the set reached can be done with the operators merge_{sep} and stop_{sep} (defined as $\text{merge}_{sep}(e, e') := e$ and $\text{stop}_{sep}(e, R) := \exists e' \in R : e \sqsubseteq e'$, or even with a simpler form $\text{stop}_{sep}(e, R) := \exists e' \in R : e = e'$). Well-known techniques for explicit-state model checking [3, 23] use such an approach to analyze the state space. This approach can be parallelized easily by synchronizing the access to the existing abstract states in the sets reached and waitlist and applying the operators $\rightsquigarrow$, merge_{sep}, and stop_{sep} concurrently. With lock-free implementations of the set data structures for reached and waitlist there is only minimal synchronization necessary for an efficient analysis. However, when using more general (and possibly more expensive) operator instances, the complete sets reached and waitlist (and also larger parts of the CPA algorithm) would need to be locked to ensure single-thread access, which prevents an efficient parallel application of the algorithm.

To circumvent this problem, our new approach does not introduce parallelism within the CPAalg algorithm, but applies several independent CPAalg instances in parallel. Each CPAalg invocation is executed in a separate thread on its own part of the state space, i.e., with its own sets reached and waitlist of abstract states, such that there is only minimal communication between the algorithm instances. The necessary infrastructure for such an approach is based on BAM. The previously given basic definition of BAM leaves room for several implementation details, such that both (the sequential and the parallel) implementation match the given specification. The computation and application of block abstractions can be done in sequential or parallel manner.

3 PARALLEL BAM

Our contribution is a scalable parallelization of the sequential algorithm of BAM. Block abstractions are independent from each other and also from the surrounding context. Thus, they can be computed in parallel, as soon as the initial abstract state of a block abstraction is known. The sequential version of BAM, which was defined by Wonisch and Wehrheim [31], recursively calls another CPA algorithm for each newly entered block, waits for its termination and directly uses the result as a block abstraction of the entered block. In contrast to that, our parallel version schedules the computation of a nested block abstraction in another thread and continues with the analysis of further abstract states from the set waitlist.

Each block abstraction is computed by a separate instance of the CPAalg algorithm (in own thread), with own instances of the sets reached and waitlist, and a thread-safe instance of the transfer relation $\rightsquigarrow$ and the operators merge and stop. The operators are stateless, and thus can be used in parallel from several threads. There is no need to lock the data structures of a CPAalg instance. In parallel algorithms, a critical point is the number of synchronizations. Block abstractions are *large enough* to avoid expensive synchronization for single steps during the computation. Synchronization is only needed when entering or leaving a block, i.e., when starting or terminating a block's analysis instance. Additionally, the communication only happens between dependent block abstractions, such that no global locking is required in the algorithm.

3.1 Jobs as Components with Dependencies

Our technique is based on the parallel execution of components named *jobs*. A job $job = (\mathbb{D}, \text{reached}, \text{waitlist}, B)$ consists of

- a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ that determines the analysis (in our case we always set $\mathbb{D} = \mathbb{BAM}$),
- a set reached and a set waitlist of abstract states to be analyzed, and
- a block $B = (L', G')$ representing the partition of the program's CFA to be analyzed.

A job is executed by applying Alg. 1 CPAalg with the given CPA $\mathbb{D}$ on the sets reached and waitlist. Note that there can be several jobs for the same block B, but each set reached and each set waitlist are assigned to exactly one job. There are no shared data based on abstract states for different jobs. This allows us to execute jobs in parallel, because the job executions are independent from each other. If a block has nested blocks, the corresponding block abstraction depends on the block abstractions of those nested blocks. In the sequential implementation of BAM, the dependencies of block abstractions on nested-block abstractions are implicitly solved by calling algorithm CPAalg recursively, i.e., the analysis of an outer block waits until a nested block abstraction is computed completely, and then continues. In the parallel approach we explicitly maintain such dependencies between (analyses of) block abstractions. A relation deps $\in jobs \times E \times jobs$ tracks at which abstract state a block abstraction needs to be computed and applied. This relation needs to be globally visible, shared across all threads, and modifications are applied atomically. As dependencies are only modified when a job is started or terminated, the overhead for synchronization is negligible. Our implementation does currently not support recursive tasks and thus there are no cyclic dependencies between block abstractions.

3.2 Scheduling and Job Execution

The parallel execution of analyses needs a scheduling algorithm that distributes the parallel running analyses onto the available processing units. In our case we chose a simple task queue from the Java Concurrency API, where we insert our jobs, and let the framework do the scheduling. We can set the number of running threads to the available hardware by using the default Java thread pool. For simplicity of Alg. 3, the actual scheduling is hidden in the call *schedule* that (asynchronously) executes the given job with the given data.[2] This solution has only small overhead (for run time and for developers) and is performant enough for the analysis, even when applied to a larger scale of computing resources. We have nearly linear speedup when using multiple cores (see Sect. 4), thus we assume that the build-in scheduling is efficient enough for our currently available hardware.

The basic idea of a parallel implementation is given in Algs. 2 and 3. The function abort of Alg. 1 CPAalg terminates the analysis as soon as a nested block abstraction needs to be computed. In this case, we determine the necessary data to compute the block abstraction in our scheduling algorithm and schedule a new analysis to compute the nested-block abstraction asynchronously. The abstract state before entering the block is removed from the current set waitlist and stored as a part of the dependency relation deps. After the computation of the nested-block abstraction is finished, the dependency is removed from deps and the state is re-added into the set waitlist. The function schedule executes the given job asynchronously with algorithm Alg. 3. The asynchronous execution of a job can be delayed due to limited resources or because the same job is scheduled twice, i.e., with different arguments. We use a thread pool for scheduled jobs based on a job queue with a FIFO ordering strategy. The function scheduleAndWait does the same, but awaits the termination of the job. The method registerBA is executed whenever a block analysis terminates. It extracts the block abstraction from the analyzed set reached and updates the cache of BAM.

[2]The pseudo code omits some scheduling-related code, as this would be too much detail for this description and can be looked up in our reference implementation.

Algorithm 2 ParallelBAM($\mathbb{D}$, reached, waitlist): Initial step for parallel BAM

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
where E denotes the set of elements of the lattice of D,
a set reached $\subseteq E$ of abstract states,
a set waitlist $\subseteq$ reached of frontier abstract states,
a global relation deps $\subseteq jobs \times E \times jobs$ to track computations of block abstractions
Output: a set of reachable abstract states,
a subset of frontier abstract states

```
1: deps := ∅
2: mainJob := (𝔻, reached, waitlist, mainBlock)
3: JobExecutor(mainJob, deps, ∅)
4: return (mainJob.reached, mainJob.waitlist)
```

Algorithm 3 JobExecutor(job, deps, statesToAdd): Job execution for parallel BAM

Input: a job $= (\mathbb{D}, \text{reached}, \text{waitlist}, B)$,
a global relation deps $\subseteq jobs \times E \times jobs$ to track computations of block abstractions,
a set statesToAdd $\subseteq E$ of abstract states to be added before starting the analysis

```
1: job.waitlist := job.waitlist ∪ statesToAdd
2: deps := deps \ {(job, e, ·) ∈ deps | e ∈ statesToAdd}
3: job.reached, job.waitlist :=
       CPAalg(𝔻, job.reached, job.waitlist)
4: missingBAs := {e ∈ reached | hasMissingBA(e)}
5: if missingBAs ≠ ∅ then    // nested BA needed
6:   for e ∈ missingBAs do
7:     job.waitlist := job.waitlist \ {e}
8:     childJob := (ℂ, {e}, {e}, getBlock(e))
9:     deps := deps ∪ {(job, e, childJob)}
10:    schedule(childJob, deps, ∅)
11:  schedule(job, deps, ∅)
12: else
13:  finished := job.waitlist = ∅ ∧ {(job, ·, ·) ∈ deps} = ∅
14:  shouldAbort := ∃e ∈ job.reached : abort(e)
15:  if finished ∨ shouldAbort then
16:    registerBA(job.reached, shouldAbort)
17:    parents := {(·, ·, job) ∈ deps}
18:    for (parentJob, updateState, ·) ∈ parents do
19:      schedule(parentJob, deps, {updateState})
20:    deps := deps \ parents
```

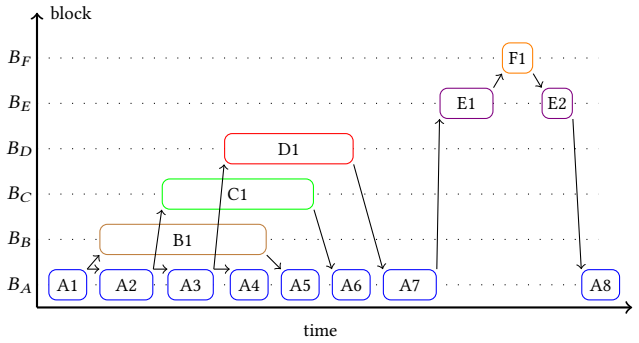


Figure 2: Schematic time line of a possible execution of jobs with parallel BAM for the example in Fig. 1

3.3 Example Application of Parallel BAM

The CFA in Fig. 1 consists of two characteristic parts: the upper part has heavy branching and several control-flow paths, the part below location 17 consists of a simple chain of locations. Parallel BAM implicitly recognizes this structure and the scheduling will apply a parallel analysis for the upper part. Figure 2 shows a possible time line for the execution of the new algorithm for the CFA given in Fig. 1. The heavy branching part of the program results in independent blocks B_B, B_C, and B_D, which can be analyzed in parallel. Each box in Fig. 2 represents a job, consisting of a CPA $\mathbb{W}$, a set reached, a set waitlist, and a block $B \in \{B_A, ..., B_F\}$. For each block (more concretely: for each set reached), there can be several jobs that are applied in sequential order.

For the example, let us assume a depth-first search as iteration order and an expensive computation in the blocks B_B, B_C, and B_D. In general, the iteration order for the program analysis can be configured by the user, and the effort to analyze blocks depends of course on the given task.

Initially, Alg 2 creates job $A1$ (Alg. 2, line 2) for the analysis of the block B_A. Figure 2 shows the execution of job $A1$ with Alg. 3 as a box along the time axis. Internally, Alg. 1 CPAalg analyzes the first abstract states of the given task (Alg. 3, line 3), until the entry location of block B_B is reached. Algorithm CPAalg terminates for the job $A1$ and two further (independent) jobs $A2$ and $B1$ are scheduled (Alg. 2, line 10 and 11) and executed in parallel. The job $B1$ analyses the block B_B and is not interrupted by another block-entry location. The job $A2$ is scheduled because there is a branching at location 2 in the CFA, such that the set waitlist of the terminated CPAalg in job $A1$ was not empty.

For the example, we assume that the job $A7$ analyzes the program location with CFA location 17. For the part below location 17 however, inter-block dependencies prevent a parallel execution of jobs and we need to explicitly wait for nested-block abstractions to be computed. In Fig. 2 this is visible for jobs $E1$, $F1$, $E2$, and $A8$, which do not have any parallel execution. Overall, our parallel version of BAM uses a dynamic scheduling, such that such imbalances are prevented in most cases.

3.4 Soundness of the Parallel Approach

We take a short look at the soundness of the parallel algorithm based on its sequential instance. The main difference between the sequential and the parallel version of BAM is the computation order of block abstractions. Instead of computing one block abstraction after another, they are computed in parallel whenever possible. As the computations of block abstractions themselves are independent and do not share any relevant data, the theoretical basis for soundness does not change. Thus, the parallel approach is as sound as the sequential algorithm that was proven to be sound in [31], i.e., only the iteration strategy for the state space differs and the soundness relies on the underlying analysis $\mathbb{W}$ of BAM. In other words: If there exists an abstract path in the analyzed source file that reaches a property violation, then the same path is also explored by the parallel algorithm, consisting of the same block abstractions and abstract states as computed by a sequential analysis.

3.5 Requirements for Parallel Execution

Our parallel approach has some additional requirements on the used components: Each used CPA has to allow multi-threaded access to its main components, the operators must be thread-safe and usable in parallel. This can either be implemented (a) by stateless operators (which is the intended behavior of operators anyway) or (b) by separate instances of the operators for each accessing thread (including independent data structures). **(a)** An ideal framework would only have stateless operators (just as their theoretically defined mathematical pendant) and thus, they would easily be usable in multi-threaded context without locking or synchronization. **(b)** While the operators are stateless in theory, a large software system (such as the framework CPAchecker), where the developers integrate several different theoretical approaches, requires an implementation that partially deviates from the concept of stateless operators. We noticed that the transfer relation $\rightsquigarrow$ and also the operators merge and stop for several CPAs were already designed and implemented in a stateless manner, such that they can easily be used for our parallel BAM implementation. Depending on the CPA, most of the code (and also most of the theoretical background) is placed in the transfer relation, and thus the conceptual difficulty was to rewrite those parts that are critical and might need to be synchronized. To avoid heavy synchronization, we have converted some (non-critical) parts like statistics and time measurement into a thread-safe implementation or provide independent instances of operators for special cases.

We have not only added the new algorithms (Alg. 2 and 3) for parallel BAM into the framework, but also modified some other components such that they can be combined and used with the new algorithm. The following list contains a few corner cases of CPAs that were touched or are usable with our approach:

- LocationCPA: The program location for the current analysis is tracked with the LocationCPA. As the program location of each statement is constant after parsing the program and written into the CFA location, the 'state' of the operators is the (immutable) CFA itself. Thus no changes had to be made.
- CallstackCPA: The call stack for the current analysis is determined by the CallstackCPA. As the corresponding operators are stateless (i.e., only depending on the abstract call-stack state given as parameter), no changes were required.
- ValueCPA: The ValueCPA performs an explicit-value analysis and tracks numerical values for variables. The analysis itself does not need to be changed for synchronization.

4 EVALUATION

This section compares our new parallel approach with the existing sequential implementation and shows that the new approach can reduce the response time considerably when executed on several cores. First, we compare the old sequential implementation with the new implementation (executed with only one thread), in order to show that no regression appears and that both analyses behave as similar as possible. Second, we explore the speedup of the analysis depending on the number of threads (as far as our hardware allows).

4.1 Evaluation Goals

It is clear from theory that not all verification tasks will benefit from our parallelized verification approach: (a) There are many programs where most paths have sequential dependencies between blocks and therefore, there is not much room for performance improvements from parallelization, and (b) there are many small programs, for which parallelization does not make a difference. We claim that our approach is effective in both regards: it parallelizes and speeds up verification process (response time) *if* the structure of the program contains sufficient branching and the size of the program is large enough *and* does not negatively influence the performance for those verification tasks that are small or have sequential dependencies.

Claim 1. The BAM-based approach to parallelization does not negatively impact the performance of verification tasks overall. *Evaluation Plan:* We take a large benchmark set of verification tasks and verify them with and without parallelization, restricted to one processing unit. If the run time is not worse for the parallel version, then the claim is valid.

Claim 2. The BAM-based approach to parallelization reduces the response time of verification tasks by leveraging several processing units. *Evaluation Plan:* We take a large set of verification tasks that can potentially benefit from parallelization and compare the response time of the verification with different numbers of processing units.

If this experiment is positive, the question raises where the benefit comes from: Is it the BAM-based approach to parallelization, or are there other technical components of the verifier that contribute to the speed up? What are the configurable parts of the verifier that can benefit from parallelization? Can they be controlled in an experiment (switched on and off separately)?

Claim 3. The parallelization of the program analysis using BAM contributes considerably to the speedup. *Evaluation Plan:* After identifying variables to control, we run experiments to investigate the influence of the identified components.

4.2 Benchmark Environment and Limitations

Benchmark Sets. For our evaluation we use a large subset of the SV-Benchmarks repository [4] containing over 5 400 verification tasks[3], sorted into different categories according their specification, internal structure, or behavior. For the comparison of the existing sequential implementation with the new parallel approach (limited to one CPU core), we use all verification tasks with a reachability property, in order to evaluate on a diverse set that the approach has no negative effect (Claim 1). To demonstrate the positive effect of parallelization of the new approach, we chose those verification tasks from the category *ReachSafety-ECA* that consists of rather large problems with a highly branching control flow (Claim 2).

Setup. We ran the experiments on a cluster of 168 identical machines with a hardware specification that roughly matches available resources on machines of software developers. This way, replication of our experiments does not require specific hardware. For each single verification run we limit the CPU time to 15 min and the memory to 15 GB, and we use an Intel Xeon E3-1230 v5 CPU with 3.40 GHz with 8 processing units (4 physical cores with hyper-threading). The limit of CPU time enables us to even compare the effectiveness of parallelization (response time vs. CPU time) for those verification tasks for which the verifier runs into a timeout. We evaluated our implementation in CPAchecker[4], revision `r28809`, from the official project repository[5].

Because we use Intel processors with hyper-threading, where two neighboring (virtual) processing units share some hardware components and influence each other, we pair the (virtual) processing units and use a step width of 2 for our experiments with varying number of processing units, i.e., we use 2, 4, 6, 8 processing units and omit the odd numbers of processing units. The benchmarking framework BenchExec [16] takes care of correctly assigning the two processing units of the same physical core together to the verification processes. We report all times in seconds and use the term *CPU time* for the accumulated usage of processing units of a CPU, and the terms *response time* or *wall time* for the time that elapses between the start and the termination of the verification run.

Analysis Configuration. We configure BAM to use function and loop bodies as blocks. BAM can be combined with several analyses; for our evaluation, we choose a combination where the performance influence from additional components is small: BAM with an explicit-value analysis (VA) without CEGAR. This way, we configure a simple state-space exploration based on an explicit tracking of variables and their values. Both the sequential and the parallel configurations apply a depth-first-search as exploration strategy, i.e., the set waitlist of the CPA algorithm is a FIFO queue for each configuration.

Unfortunately, we can not compare to other multi-threading verifiers for reachability properties of sequential C programs, because there exists no equivalent approach to the best of our knowledge (cf. related work in the introduction; there are portfolio verifiers).

4.3 Claim I: Sequential vs. Parallel Algorithm

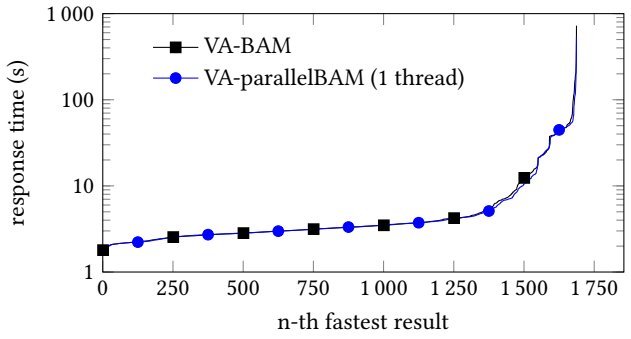
Configuration. In our first experiment we compare the existing sequential algorithm with the new parallel algorithm. Therefore, we run all experiments on only one processing unit. The FIFO ordering of the job queue (see Sect. 3.2) in the parallel algorithm guarantees that block abstractions are computed in the same order as their blocks are reached, i.e., it behaves as similar as possible to the sequential algorithm.

Results. Figures 3a and 3b show the response time of the configurations for the benchmark set containing all verification tasks with a reachability property. A quantile plot contains graphs that indicate the quantile of solved problem instances (x-axis) each

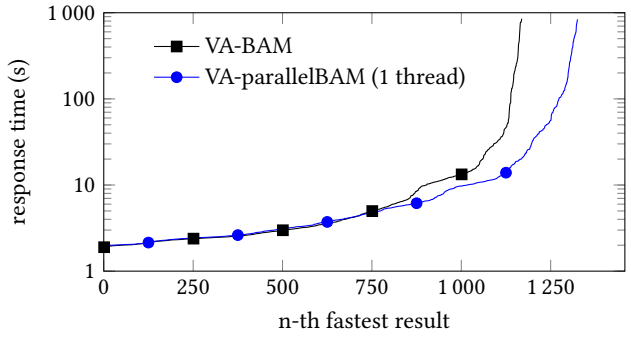
[3] https://github.com/sosy-lab/sv-benchmarks

[4] https://cpachecker.sosy-lab.org

[5] https://gitlab.com/sosy-lab/software/cpachecker



(a) Verification tasks with correctness proof



(b) Verification tasks with property violation

Figure 3: Quantile plots for results of BAM with value analysis, sequential compared to parallel version with one thread

within a certain response time (y-axis).[6] It does not show a direct comparison for individual verification tasks, but allows to compare the overall behavior of an analysis configuration. We divided the benchmarks into two groups: The plot in Fig. 3a contains results for all verification tasks for which a correctness proof was computed; the plot in Fig. 3b contains results for all verification tasks for which a property violation was found. The overall impression is that the (single-threaded) parallel technique does not have any noticeable overhead above the sequential approach, i.e., the scheduler and the job executor from Alg. 3 are efficient. The new approach behaves almost identical when computing proofs, and for finding property violations, it is even faster and can solve more problems, which we discuss in the following.

Discussion. The difference in Fig. 3b between the verification approaches results from the exploration order of the state space. After a nested-block abstraction has been computed, there is a small difference in the sorting of abstract states in the sets waitlist of both approaches. The existing sequential analysis has (and keeps) the abstract states in the set waitlist. The (single-threaded) parallel approach removes abstract states when finding a missing block abstraction (cf. Alg. 3, line 7) and re-adds those abstract states into each set waitlist (cf. Alg. 3, line 1) after computing the necessary block abstraction. There are small differences in the exploration order and depending on the task's structure, different paths might be analyzed first. In those cases, the parallel approach does not apply a pure depth-first exploration order, but partially prefers paths that do not traverse deeply nested blocks, which seems beneficial when it comes to finding property violations. For this reasoning, we conclude that for evaluating Claim II, it would not be valid to consider the verification tasks with property violations, because the variable "exploration order" is not controlled.

We conclude that Claim 1 holds, because we did not observe any negative impact of our new approach.

4.4 Claim II: Scalability of Parallel BAM

Configuration. We show the effectiveness of the parallelization of our new approach by increasing the number of threads (2, 4, 6, 8 threads) and observe the improvement of the response time. The upper limit of the number of threads is determined by the hardware that we use. We set the number of processing units assigned to the verification process to be equal to the number of threads. We chose a subset of 154 tasks from the category *ReachSafety-ECA*, such that they need a reasonable amount of time (at least 3 s with only one thread) and do not contain a property violation. With a too small analysis time, the default overhead of the CPAchecker framework itself (like JVM startup time or parsing time) hides the effect of the parallel approach and blurs the picture. Additionally, finding a path to a property violation with a parallel verification approach easily leads to non-deterministic results if there are several property violations in a verification task or a property can be reached via different program paths[7]. Thus, we select from the benchmark set only those verification tasks without property violation, in order to make sure to compare the response time that is necessary to analyze the whole state space. The used benchmark set consists of three groups: 47 simple tasks, 36 medium tasks, and 71 difficult tasks. The difficulty is roughly given by the size of the state space to be explored.

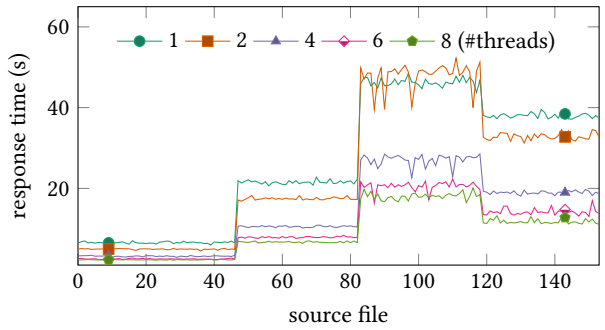
Results. Figure 4a shows the response time of the configurations for the benchmark set. Each function graph in the quantile plot refers to a different number of threads used in the analysis. A smaller response time of the analysis corresponds to a smaller state space and relates to a simpler task. The different groups of verification tasks (simple, medium, and difficult) are clearly recognizable by the level of response time, i.e., the plot contains larger steps. Overall, additional threads improve the performance of the analysis.

Figure 4b shows the speedup of our parallel approach over the single-threaded application in the evaluation using box plots. Each entry in the plot shows the median as the horizontal line within the box, together with its two surrounding quartiles between the upper and lower line of the box, as well as the minimum and maximum as whiskers. The speedup becomes larger the more threads we use. The evaluation with 2 threads outperforms the single-threaded execution by about 20% (median). The parallel approach with 8 threads is about three times as fast as with 2 threads.

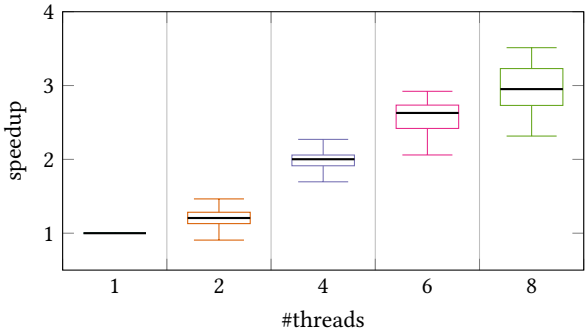
Discussion. The results look impressive: Only by parallelizing independent BAM explorations in a way that is not tailored in any specific way towards the framework or to a particular abstract domain, we observe a significant improvement of the response time. Obviously, some parts of the verification process cannot be executed in parallel. This denies a 'perfect' parallelization and is known as Amdahl's law [1]. The sequential parts include the startup process of

[6] A detailed description of quantile plots can be found in the literature [16].

[7] The supplementary artifact [9] and website include additional data about the evaluation of our approach on a benchmark set of tasks containing a property violation.



(a) Quantile plot for verification tasks without property violation



(b) Box plot comparing 1 thread to N threads; without property violation

Figure 4: Comparison of response time for different numbers of threads, based on restricted benchmark set

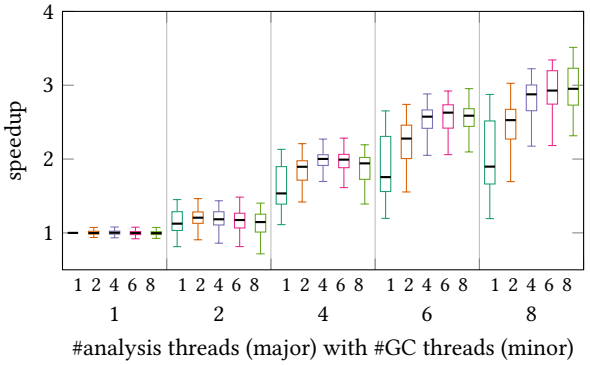
CPAchecker as well as the initial overhead of the analysis to compute blocks for BAM and analyze parts of the most outer block until a nested block is reached, which in turn can be analyzed in a parallel manner. Some parts of the implementation cause an additional synchronization overhead, like multi-threaded statistics for the concurrent access to shared resources like the cache of BAM. The rather modest improvement from 1 thread to 2 threads is most likely due to hyperthreading of the processor, where the two processing units of one physical core share important hardware resources.[8]

We conclude that Claim 2 holds, because the experiments show that for those programs that have potential for speedup by parallelization, we actually observe a significant speedup.

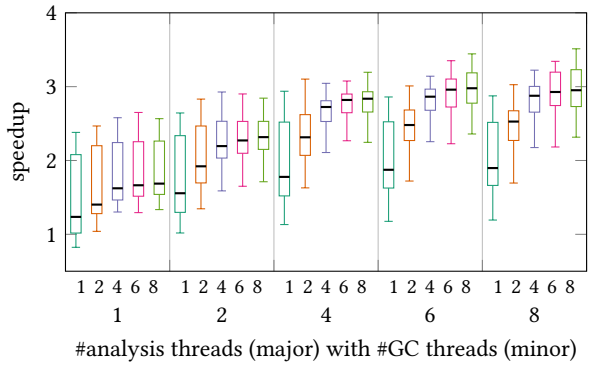
4.5 Claim III: Control Influencing Variables

The previous experiments show that several processing units are effectively used by the verification tool, but it is unclear where the benefit comes from. Therefore, we need to investigate which parts of the verifier are parallelized and make sure that our new approach contributed to the benefit. Since our implementation is based on Java, we have also enabled the JVM to use multi-threaded garbage collection (GC), because if we create abstract states in a parallel manner, we should also deallocate them in parallel. The default strategy for GC in OpenJDK 1.8.0 is a combination of *PS MarkSweep* and *PS Scavenge*. The mark-sweep collector applies a full mark-sweep garbage collection algorithm for old-generation objects. The parallel scavenge collector cleans up young-generation objects.

[8] In our experiments we assigned successive processing units to the verification runs; the experiment with 2 threads could be improved by using two processing units of different physical cores.



(a) Box plot comparing the response time of 1 thread to 8 threads, evaluated on as many processing units as #analysis threads



(b) Box plot comparing the response time of 1 thread to 8 threads, evaluated on 8 processing units

Figure 5: Comparison of different numbers of analysis threads and different numbers of GC threads

Configuration. We use the 154 tasks from the previous experiment and re-evaluate them. We divide our evaluation into two cases: First, the number of available processing units is equal to the number of analysis threads. Second, the number of available processing units is set to 8, which is the upper limit the available hardware. For both cases, we evaluated all combinations of analysis threads (using 1, 2, 4, 6, 8 threads; major, large numbers in figure) and GC threads (using 1, 2, 4, 6, 8 threads; minor, small numbers in figure).

Results. We present the speedup statistics for comparing the response time of a single-threaded analysis with a single-threaded GC on a single processing unit to an execution with a given number of analysis threads with a given number of threads for GC on a given number of processing units. In Fig. 5a the number of available processing units is equal to the number of analysis threads. In Fig. 5b all 8 processing units of the machine are available to the verifier. In both figures we configure the number of analysis threads and GC threads. In each plot, the horizontal axis contains 5 major groups (representing the number of analysis threads) of each 5 minor entries (number of threads for GC). For example, the five first (most left) entries in each figure show the speedup of the approach if using one thread for the analysis and a varying number of threads for GC. Unsurprisingly, the overall result is that using multiple threads for both the analysis and additionally the GC is beneficial. Nearly all tasks are solved faster if multiple processing units are assigned to the verification process.

Discussion. In Fig. 5a, the first entry of each group shows the speedup of the analysis when using only one thread for GC. This isolates the the benefit of multi-threading caused by our new analysis approach. Similarly, Fig. 5b shows (within each of the 5 groups) that keeping the number of analysis threads constant and incrementing the number of threads for GC also speeds up the verification process. Therefore both, analysis and GC, benefit from multi-threading. Figure 5b shows that if the analysis is bound to one thread, the benefit from multi-threading is rather limited, while the speedup is improved if we use more threads for the analysis. The most interesting indicators are the median value (middle line inside the box) and the minimal speedup values (lower whisker). The overall variance for the response time and speedup is quite large if there are several processing units available. This might indicate a non-deterministic scheduling of workload across free resources, in contrast to the narrow boxes in Fig. 5a in the left two groups (where the number of processing units is bound to one and two, respectively).

We conclude that Claim 3 holds, because we were able to isolate and control the only other cause for a significant speedup, and the experiments confirmed that our new approach is reponsible for the improved performance of the analysis, while the parallel GC algorithms of the JVM take care of parallelized deallocation.

4.6 Threats to Validity

External Validity: Our benchmark suite consists of a large set of C source files. We use the largest publicly available benchmark suite in order to optimize the diversity in size and type of programs. This is particularly important for evaluating Claim 1. For Claims 2 and 3, we restricted the benchmark set to verification tasks that have potential to benefit from parallelization. Our evaluation is restricted to the language C, and while it seems clear that the concepts and results can be transferred to other imperative languages, such a claim is not backed up by our experiments. The chosen time limit of 15 min and memory limit of 15 GB for verifying a given task is inspired by the research community on software verification (cf. one of the reports on the International Competition on Software Verification [4]). Of course, the evaluation of our approach depends on the tool in which it is implemented. There is currently no other tool implementing the same approach, and a comparison with a completely different approach for parallel analysis might be misleading. [9] With the assumption that the default configuration is optimized for most use cases, we did not change the configuration of the JVM except the increment of maximal heap memory and the adjustment of the garbage-collection strategy, such that the effect of the number of threads can be measured. The available hardware might also influence the results. For parallel execution, the internal structure of the CPU is a critical element, i.e., low-level caching and the hierarchy of processing units have a large effect on the run time of tasks. We used a modern Intel Xeon E3-1230 v5 that is available on the market for a reasonable price, in order to obtain results that have a higher externally validity than experiments on special high-performance clusters.

[9] The supplementary artifact [9] and website include an additional comparison with some non-BAM analysis approaches, in order to show that using the BAM technology does not negatively effect an analysis' performance (known result [31]).

Internal Validity: Besides garbage collection of the JVM, there are other factors that influence the speedup of the parallel approach. Some components of CPAchecker, e.g., counters and measurements for statistics, are not yet fully optimized for parallel execution. Additionally, it depends on the task's structure how many blocks can be analyzed in parallel. Controlling this variable (number of parallelization blocks) is not possible or very difficult, thus, we prefer to increase the internal validity by the large number of experiments on different tasks. Another control variable is the block size. Larger blocks are beneficial for a concurrent analysis, due to the smaller synchronization footprint. For Claims 2 and 3, the benchmark set was already chosen such that it contains only programs where the block size is very large. Thus, we did not further analyze different block sizes. We also need to consider that the explicit-value analysis computes a large number of abstract states, while other abstract domains might lead to more compact representations of the state space, and the fewer abstract states are explored the less might be parallelized. Our time measurement includes the memory allocation for the JVM, parsing time, and internal statistics, which adds processing workload that cannot be parallelized currently. We mitigate this effect by using only those verification tasks that need more than 3 s when using one thread, i.e., we consider verification tasks for which the analysis itself consumes a portion of the run time that is not negligible.

5 CONCLUSION

We presented a new approach for multi-threaded software verification that is based on program-block summaries. Our emphasis is on providing a solution that follows the principle of separation of concerns: the problem of making the analysis benefit from multiple processing units is treated completely orthogonal from the problem of designing and implementing an abstract domain and the operators for a program analysis. We formally define the new algorithm in the framework, provide a working implementation, and demonstrate its applicability on a large set of benchmarks. The experiments show that our approach (a) does not add noticeable overhead for verification tasks that do not benefit from parallelization, (b) can considerably speed up the verification process in many cases (given the verification task has a certain minimal size and some independent branches to explore), and (c) contributes largely to the performance improvements, i.e., the speedup is not only due to multi-threading features that the JVM provides.

The presented algorithm is implemented as a shared-memory approach, which allows efficient interaction of all components. As the number of CPU cores per machine and also the amount of memory per host is limited, we plan to extend our algorithm to leverage several processes that might be distributed over several machines in a cluster. An additional benefit would be a simpler usage of abstract domains that rely on libraries that are not thread-safe, because there is no problem with interleaved usage of libraries in separate processes. Additionally we plan to offload the cache of BAM to a disk-based storage, in order to lower the memory usage for very resource intensive tasks. The combination of both, a distributed, multi-process verification algorithm and a disk-based cache, seems to be very promising for the verification of very large programs.

REFERENCES

[1] G. M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS*. ACM, 483–485. https://doi.org/10.1145/1465482.1465560

[2] P. Andrianov, K. Friedberger, M. U. Mandrykin, V. S. Mutilin, and A. Volkov. 2017. CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 355–359. https://doi.org/10.1007/978-3-662-54580-5_22

[3] J. Barnat, J. Havlícek, and P. Rockai. 2013. Distributed LTL Model Checking with Hash Compaction. *ENTCS* 296 (2013), 79–93. https://doi.org/10.1016/j.entcs.2013.07.006

[4] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[5] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[7] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[8] D. Beyer, M. Dangl, and P. Wendler. 2018. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[9] D. Beyer and K. Friedberger. 2018. Replication Package for Article "Domain-Independent Multi-threaded Software Model Checking" in Proc. ASE'18. https://doi.org/10.5281/zenodo.1322090

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[11] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *Proc. FSE*. ACM, Article 57, 11 pages. https://doi.org/10.1145/2393596.2393664

[12] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[13] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[14] D. Beyer and T. Lemberger. 2016. Symbolic Execution with CEGAR. In *Proc. ISoLA (LNCS 9952)*. Springer, 195–211. https://doi.org/10.1007/978-3-319-47166-2_14

[15] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://www.sosy-lab.org/research/pub/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf

[16] D. Beyer, S. Löwe, and P. Wendler. 2017. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer* (2017). https://doi.org/10.1007/s10009-017-0469-y

[17] D. Beyer and A. Stahlbauer. 2014. BDD-based software verification: Applications to event-condition-action systems. *STTT* 16, 5 (2014), 507–518. https://doi.org/10.1007/s10009-014-0334-1

[18] S. Blom, J. van de Pol, and M. Weber. 2010. LTSmin: Distributed and Symbolic Reachability. In *Proc. CAV (LNCS 6174)*. Springer, 354–359.

[19] M. Brain and F. Schanda. 2012. A Lightweight Technique for Distributed and Incremental Program Verification. In *Proc. VSTTE (LNCS 7152)*. Springer, 114–129. https://doi.org/10.1007/978-3-642-27705-4_10

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[21] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proc. SOSP*. ACM, 265–278. https://doi.org/10.1145/2043556.2043582

[22] A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. 2013. Ufo: Verification with Interpolants and Abstract Interpretation (Competition Contribution). In *Proc. TACAS (LNCS 7795)*. Springer, 637–640. https://doi.org/10.1007/978-3-642-36742-7_52

[23] G. J. Holzmann. 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

[24] B. A. Huberman, R. M. Lukose, and T. Hogg. 1997. An Economics Approach to Hard Computational Problems. *Science* 275, 7 (1997), 51–54. http://www.hpl.hp.com/research/idl/papers/EconomicsApproach/EconomicsApproach.pdf

[25] K. Laster and O. Grumberg. 1998. Modular Model Checking of Software. In *Proc. TACAS (LNCS 1384)*. Springer, 20–35. https://doi.org/10.1007/BFb0054162

[26] P. Müller, P. Peringer, and T. Vojnar. 2015. Predator Hunting Party (Competition Contribution). In *Proc. TACAS (LNCS 9035)*. Springer, 443–446.

[27] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato. 2017. Parallel bug-finding in concurrent programs via reduced interleaving instances. In *Proc. ASE*. IEEE Computer Society, 753–764. https://doi.org/10.1109/ASE.2017.8115686

[28] E. Sherman and M. B. Dwyer. 2018. Structurally Defined Conditional Data-Flow Static Analysis. In *Proc. TACAS, Part II (LNCS 10806)*. Springer, 249–265. https://doi.org/10.1007/978-3-319-89963-3_15

[29] V. Still, P. Rockai, and J. Barnat. 2016. DIVINE: Explicit-State LTL Model Checker (Competition Contribution). In *Proc. TACAS (LNCS 9636)*. Springer, 920–922.

[30] T. van Dijk. 2016. *Sylvan: multi-core decision diagrams*. Ph.D. Dissertation. University of Twente, Enschede, Netherlands. http://purl.utwente.nl/publications/100676

[31] D. Wonisch and H. Wehrheim. 2012. Predicate Analysis with Block-Abstraction Memoization. In *Proc. ICFEM (LNCS 7635)*. Springer, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24