Dynamic Reductions for Model Checking Concurrent Software

Henning Günther[1], Alfons Laarman[1], Ana Sokolova[2], and Georg Weissenbacher[1]

[1] TU Wien*
[2] University of Salzburg

Abstract. Symbolic model checking of parallel programs stands and falls with effective methods of dealing with the explosion of interleavings. We propose a dynamic reduction technique to avoid unnecessary interleavings. By extending Lipton's original work with a notion of bisimilarity, we accommodate dynamic transactions, and thereby reduce dependence on the accuracy of static analysis, which is a severe bottleneck in other reduction techniques.
The combination of symbolic model checking and dynamic reduction techniques has proven to be challenging in the past. Our generic reduction theorem nonetheless enables us to derive an efficient symbolic encoding, which we implemented for IC3 and BMC. The experiments demonstrate the power of dynamic reduction on several case studies and a large set of SVCOMP benchmarks.

1 Introduction

The rise of multi-threaded software—a consequence of a necessary technological shift from ever higher frequencies to multi-core architectures—exacerbates the challenge of verifying programs automatically. While automated software verification has made impressive advances recently thanks to novel symbolic model checking techniques, such as lazy abstraction [27,6], interpolation [34], and IC3 [9] for software [7,10], multi-threaded programs still pose a formidable challenge.

The effectiveness of model checking in the presence of concurrency is severely limited by the state explosion caused through thread interleavings. Consequently, techniques that avoid thread interleavings, such as partial order reduction (POR) [39,42,20] or Lipton's reduction [33], are crucial to the scalability of model checking, while also benefitting other verification approaches [18,12,15].

These reduction techniques, however, rely heavily on the identification of statements that are either independent or commute with the statements of all other threads, i.e. those that are *globally independent*. For instance, the single-action rule [32]—a primitive precursor of Lipton reduction—states that a sequential block of statements can be considered an atomic transaction if all but one of

*This work is supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

the statements are globally independent. Inside an atomic block, all interleavings of other threads can be discarded, thus yielding the reduction.

Identifying these globally independent statements requires non-local *static analyses*. In the presence of pointers, arrays, and complicated branching structures, however, the results of an up-front static analysis are typically extremely conservative, thus a severe bottleneck for good reduction.

Fig. 1 shows an example with two threads (T1 and T2). Let's assume static analysis can establish that pointers `p` and `q` never point to the same memory throughout the program's (parallel) execution. This means that statements involving the pointers are globally independent, hence they globally commute, e.g. an interleaving `*p++; *q = 1` always yields the same result as `*q = 1; *p++;`. Assuming that `*p++;` is also independent of the other statements from T2 (`b = 2` and `c = 3`), we can reorder any trace of the parallel program to a trace where `*p++` and `*q = 2` occur subsequently without affecting the resulting state. The figure shows one example. Therefore, a syntactic transformation from `*p++; *q = 2` to `atomic{*p++; *q = 2}` is a valid static reduction.

Still, it is often hard to prove that pointers do not overlap throughout a program's execution. Moreover, in many cases, pointers might temporarily overlap at some point in the execution. For instance, assume that initially `p` points to the variable `b`. This means that statements `b = 2` and `*p++` no longer commute, because `b = 2; b++` yields a different result than `b++; b = 2`. Nevertheless, if `b = 2` already happened, then we can still swap instructions and achieve the reduction as shown in Fig. 1. Traditional, static reduction methods cannot distinguish whether `b = 2` already happened and yield no reduction. Sec. 2 provides various other real-world examples of hard cases for static analysis.

In Sec. 4.2, we propose a dynamic reduction method that is still based on a similar syntactic transformation. Instead of merely making sequences of statements atomic, it introduces branches as shown in Fig. 1 (T1'). A dynamic commutativity condition determines whether the branch with or without reduction is taken. In our example, the condition checks whether the program counter of T2 (`pc_T2`) still points to the statement `b = 2` (`pc_T2` == 1). In that case, no reduction is performed, otherwise the branch with reduction is taken. In addition to conditions on the program counters, we provide other heuristics comparing pointer and array values dynamically.

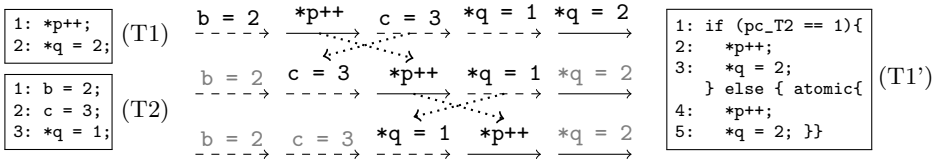


Fig. 1: (Left) C code for threads T1 and T2. (Middle) Reordering (dotted lines) a multi-threaded execution trace (T1's actions are represented with straight arrows and T2's with 'dashed' arrows). (Right) The instrumented code for T1.

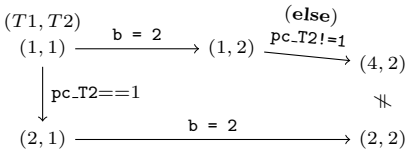


Fig. 2: Loss of commutativity

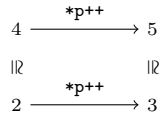


Fig. 3: Bisimulation (T1)

The instrumented code (T1') however poses one problem: the branching condition no longer commutes with the statement that enables it. In this case, the execution of `b = 2` disables the condition, thus before executing `b = 2`, T1' ends up at Line 2, whereas after `b = 2` it ends up at Line 4 (see Fig. 2). To remedy this, we require in Sec. 4.3 that the instrumentation guarantees bisimilarity of target states. Fig. 3 shows that locations 2 and 4 of T1' are bimilar, written $2 \cong 4$, which implies that any statement executable from the one is also executable from the other, ending again in a bisimilar location, e.g. $3 \cong 5$. As bisimularity is preserved under parallel composition, e.g. $(4, 2) \cong (2, 2)$, we can prove the correctness of our dynamic reduction method (see our technical report [19]).

The benefit of our syntactic approach is that the technique can be combined with symbolic model checking checking techniques (Sec. 5 provides an encoding for our lean instrumentation). Thus far, symbolic model checkers only supported more limited and static versions of reduction techniques as discussed in Sec. 7.

We implemented the dynamic reduction and encoding for LLVM bitcode, mainly to enable support for C/C++ programs without dealing with their intricate semantics (the increased instruction count of LLVM bitcode is mitigated by the reduction). The encoded transition relation is then passed to the Vienna Verification Tool (VVT) [25], which implements both BMC and IC3 algorithms extended with abstractions [7]. Experimental evaluation shows that (Sec. 6) dynamic reduction can yield several orders of magnitude gains in verification times.

2 Motivating Examples

Lazy initialization. We illustrate our method with the code in Fig. 4. The main function starts two threads executing the `worker_thread` function, which processes the contents of `data` in the for loop at the end of the function. Using a common pattern, a worker thread lazily delays the initialization of the global `data` pointer until it is needed. It does this by reading some content from disc and setting the pointer atomically via a compare-and-swap operation (`CAS`) at label `W` (whose semantics here is an atomic C-statement: `if (data==NULL) { data =`

```
   int *data = NULL;
   void worker_thread(int tid) {
c:   if (data == NULL) {
d:     int *tmp = read_from_disk(1024);
W:     if (!CAS(&data, NULL, tmp)) free(tmp);
     }
     for (int i = 0; i < 512; i++)
R:     process(data[i + tid * 512]);
   }
   int main () {
a:   pthread_create(worker_thread, 0); // T1
b:   pthread_create(worker_thread, 1); // T2
   }
```

Fig. 4: Lazy initialization

`tmp; return 1; }` `else return 0;`). If it fails (returns 0), the locally allocated data is freed as the other thread has won the race.

The subsequent read access at label `R` is only reachable once `data` has been initialized. Consequently, the write access at `W` cannot possibly interfere with the read accesses at `R`, and the many interleavings caused by both threads executing the for loop can safely be ignored by the model checker. This typical pattern is however too complex for static analysis to efficiently identify, causing the model checker to conservatively assume conflicting accesses, preventing any reduction.

Hash table. The code in Fig. 5 implements a lockless hash table (from [31]) inserting a value `v` by scanning the bucket array `T` starting from `hash`, the hash value calculated from `v`. If an empty bucket is found (`T[index]==E`), then `v` is atomically inserted using the `CAS` operation. If the bucket is not empty, the operation checks whether the value was already inserted in the bucket (`T[index] == v`). If that is not the case, then it probes the next bucket of `T` until either `v` is found to be already in the table, or it is inserted in an empty slot, or the table is full. This basic bucket search order is called a linear *probe sequence*.

A thread performing `find-or-put(25)`, for instance, merely reads buckets `T[2]` to `T[5]`. However, other threads might write an empty bucket, thus causing interference. To show that these reads are independent, the static analysis would have to demonstrate that the writes happen to different buckets. Normally this is done via alias analysis that tries to identify the buckets that are written to (by the CAS operation). However, because of the hashing and the probe sequence, such an analysis can only conclude that all buckets may be written. So all operations involving `T`, including the reads, will be classified as non-commuting. However if we look at the state of individual buckets, it turns out that a common pattern is followed using the CAS operation: A bucket is only written when it is empty, thereafter it doesn't change. In other words, when a bucket `T[i]` does not contain `E`, then any operation on it is a read and consequently is independent.

```
int T[10] = {E,E,22,35,46,25,E,E,91,E};

int find-or-put(int v) {
   int hash = v / 10;
   for (int i = 0; i < 10; i++) {
      int index = (i + hash) % 10;
      if (CAS(&T[index], E, v)) {
         return INSERTED;
      } else if (T[index] == v)
         return FOUND;
   }
   return TABLE_FULL;
}
int main() {
   pthread_create(find-or-put, 25);
   pthread_create(find-or-put, 42);
   pthread_create(find-or-put, 78);
}
```

Fig. 5: Lockless hash table.

```
  int x = 0, y = 0;
  int *p1, *p2;

  void worker(int *p) {
     while (*p < 1024)
        *p++;
  }
  int main(){
a:   if (*)
b:      { p1 = &x; p2 = &y; }
     else
c:      { p1 = &y; p2 = &x; }
     pthread_create(worker, p1); // T1
     pthread_create(worker, p2); // T2
     pthread_join(t1);
     pthread_join(t2);
     return x+y;
  }
```

Fig. 6: Load balancing.

Load balancing. Fig. 6 shows a simplified example of a common pattern in multi-threaded software; load balancing. The work to be done (counting to 2048) is split up between two threads (each of which counts to 1024). The work assignment is represented by pointers `p1` and `p2`, and a dynamic hand-off to one of the two threads is simulated using non-determinism (the first if branch). Static analysis cannot establish the fact that the partitions are independent, because they are assigned dynamically. But because the pointer is unmodified after assignment, its dereference commutes with that in other worker threads.

Sec. 4 shows how our examples can be reduced with dynamic commutativity.

3 Preliminaries

A concurrent program consists of a finite number of sequential procedures, one for each thread i. We model the syntax of each thread i by a control flow graph (CFG) $G_i = (V_i, \delta_i)$ with $\delta_i \subseteq V_i \times A \times V_i$ and A being the set of actions, i.e., statements. V_i is a finite set of locations, and $(l, \alpha, l') \in \delta_i$ are (CFG) edges. We abbreviate the actions for a thread i with $\Delta_i = \{\alpha \mid \exists l, l' \colon (l, \alpha, l') \in \delta_i)\}$.

A state of the concurrent system is composed of (1) a location for each thread, i.e., a a tuple of thread locations (the set Locs contains all such tuples), and (2) a data valuation, i.e., a mapping from variables (Vars) to data values (Vals). We take Data to be the set of all data valuations. Hence, a state is a pair, $\sigma = (\mathsf{pc}, d)$ where $\mathsf{pc} \in \prod_i V_i$ and $d \in \mathsf{Data}$. The locations in each CFG actually correspond to the values of the thread-local program counters for each thread. In particular, the global locations correspond to the global program counter pc being a tuple with $\mathsf{pc}_i \in V_i$ the thread-local program counter for thread i. We use $\mathsf{pc}[i := l]$ to denote $\mathsf{pc}[i := l]_i = l$ and $\mathsf{pc}[i := l]_j = \mathsf{pc}_j$ for all $j \neq i$.

Domains

i, j, k: Threads
a, b, x, y, p, p': Vars
$c, c', \ldots$: Vals
$l, l', l_1, \ldots$: V_i
d, d': Data
$\mathsf{pc}, \mathsf{pc}', \ldots$: Locs
$\sigma, \sigma', \ldots$: S
α_i: $\mathcal{P}(\mathsf{Data}^2)$

Each possible action α semantically corresponds to a binary relation $\alpha \subseteq \mathsf{Data} \times \mathsf{Data}$ representing the evolution of the data part of a state under the transition labelled by α. We call α the transition relation of the statement α, referring to both simply as α. We also use several simple statements from programming languages, such as `C`, as actions.

The semantics of a concurrent program consisting of a finite number of threads, each with CFG $G_i = (V_i, \delta_i)$, is a transition system with data (TS) $C = (S, \to)$ with $S = \mathsf{Locs} \times \mathsf{Data}$, $\mathsf{Locs} = \prod_i V_i$ and $\to = \bigcup_i \to_i$ where $\to_i$ is given by $(\mathsf{pc}, d) \to_i (\mathsf{pc}', d')$ for $\exists \alpha \colon \mathsf{pc}_i = l \wedge (l, \alpha, l') \in \delta_i \wedge (d, d') \in \alpha \wedge \mathsf{pc}' = \mathsf{pc}[i := l']$. We also write $(\mathsf{pc}, d) \xrightarrow{\alpha}_i (\mathsf{pc}', d')$ for $\mathsf{pc}_i = l \wedge (l, \alpha, l') \in \delta_i \wedge (d, d') \in \alpha \wedge \mathsf{pc}' = \mathsf{pc}[i := l']$. Hence, the concurrent program is an asynchronous execution of the parallel composition of all its threads. Each step (transition) is a local step of one of the threads. Each thread i has a unique initial location $\mathsf{pc}_{0,i}$, and hence the TS has one initial location pc_0. Moreover, there is an initial data valuation d_0 as well. Hence, the initial state of a TS is $\sigma_0 \triangleq (\mathsf{pc}_0, d_0)$.

Since we focus on preserving simple safety properties (e.g. assertions) in our reduction, w.l.o.g., we require one sink location per thread l_{sink} to represent errors (it has no outgoing edges, no selfloop). Correspondingly, error states of a TS are those in which at least one thread is in the error location.

In the following, we introduce additional notation for states and relations. Let $R \subseteq S \times S$ and $X \subseteq S$. Then left restriction of R to X is $X /\!\!/ R \triangleq R \cap (X \times S)$ and right restriction is $R \setminus\!\!\setminus X \triangleq R \cap (S \times X)$. The complement of X is denoted $\overline{X} \triangleq S \backslash X$ (the universe of all states remains implicit in this notation). Finally, R does not enable X if $\overline{X} /\!\!/ R \setminus\!\!\setminus X = \emptyset$, and R does not disable X if $X /\!\!/ R \setminus\!\!\setminus \overline{X} = \emptyset$.

Commutativity. We let $R \circ Q$ denote the *sequential composition* of two binary relations R and Q, defined as: $\{(x, z) \mid \exists y \colon (x, y) \in R \wedge (y, z) \in Q\}$. Moreover, let:

$$
\begin{aligned}
R \bowtie Q &\triangleq R \circ Q = Q \circ R && \text{(both-commute)} \\
R \overset{\rightarrow}{\bowtie} Q &\triangleq R \circ Q \subseteq Q \circ R && (R \text{ right commutes with } Q) \\
R \overset{\leftarrow}{\bowtie} Q &\triangleq R \circ Q \supseteq Q \circ R && (R \text{ left commutes with } Q)
\end{aligned}
$$

Illustrated graphically for transition relations, $\rightarrow_i$ right commutes with $\rightarrow_j$ iff

$$
\forall \sigma, \sigma', \sigma'' : \quad \begin{array}{c} \sigma \\ \downarrow \\ \sigma' \end{array} \quad \begin{array}{c} \\ \\ \rightarrow_j \end{array} \quad \begin{array}{c} \\ \\ \sigma'' \end{array} \qquad \Rightarrow \exists \sigma''' : \quad \begin{array}{c} \sigma \\ \downarrow \\ \sigma' \end{array} \quad \begin{array}{c} \rightarrow_j \\ \\ \rightarrow_j \end{array} \quad \begin{array}{c} \sigma''' \\ \downarrow \\ \sigma'' \end{array} \tag{1}
$$

Conversely, $\rightarrow_j$ *left commutes* with $\rightarrow_i$. The typical example of (both) commuting operations $\xrightarrow{\alpha}_i$ and $\xrightarrow{\beta}_i$ is when α and β access a disjoint set of variables. Two operations may commute even if both access the same variables, e.g., if both only read or both (atomically) increment/decrement the same variable.

Lipton Reduction. Lipton [33] devised a method that merges multiple sequential statements into one atomic operation, thereby radically reducing the number of states reachable from the initial state as Fig. 7 shows for a transition system composed of two (independent, thus commuting) threads.

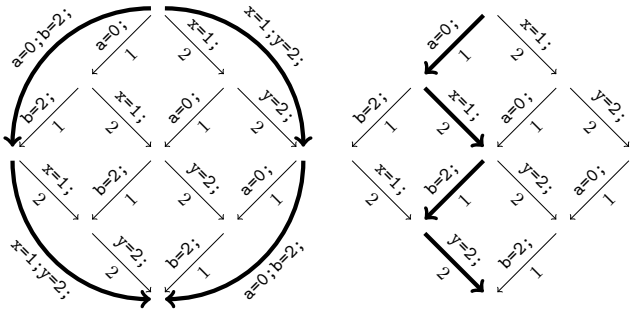


Fig. 7: Example transition system composed of two independent threads (twice). Thick lines show a Lipton reduced system (left) and a partial-order reduction (right).

Lipton called a transition $\xrightarrow{\alpha}_i$ a right/left mover if and only if it satisfies:

$$\xrightarrow{\alpha}_i \overset{\rightarrow}{\bowtie} \bigcup_{j\neq i} \rightarrow_j \text{ (right mover)} \quad \xrightarrow{\alpha}_i \overset{\leftarrow}{\bowtie} \bigcup_{j\neq i} \rightarrow_j \text{ (left mover)}$$

Both-movers are transitions that are both left and right movers, whereas *non-movers* are neither. The sequential composition of two movers is also a corresponding mover, and vice versa. Moreover, one may always safely classify an action as a non-mover, although having more movers yields better reductions.

Lipton reduction only preserves halting. We present Lamport's [32] version, which preserves safety properties such as $\Box\varphi$: Any sequence $\xrightarrow{\alpha_1}_i \circ \xrightarrow{\alpha_2}_i \circ \cdots \circ \xrightarrow{\alpha_{n-1}}_i \circ \xrightarrow{\alpha_n}_i$ can be *reduced* to a single *transaction* $\xrightarrow{\alpha}_i$ where $\alpha = \alpha_1; \ldots; \alpha_n$ (i.e. a compound statement with the same local behavior), if for some $1 \leq k < n$:

L1. statements before α_k are right movers, i.e.: $\xrightarrow{\alpha_1}_i \circ \cdots \circ \xrightarrow{\alpha_{k-1}}_i \overset{\rightarrow}{\bowtie} \bigcup_{j\neq i} \rightarrow_j$,
L2. statements after α_k are left movers, i.e.: $\xrightarrow{\alpha_{k+1}}_i \circ \cdots \circ \xrightarrow{\alpha_n}_i \overset{\leftarrow}{\bowtie} \bigcup_{j\neq i} \rightarrow_j$,
L3. statements after α_1 do not block, i.e.: $\forall\sigma\, \exists\sigma' \colon \sigma \xrightarrow{\alpha_1}_i \circ \cdots \circ \xrightarrow{\alpha_n}_i \sigma'$, and
L4. φ is not disabled by $\xrightarrow{\alpha_1}_i \circ \cdots \circ \xrightarrow{\alpha_{k-1}}_i$, nor enabled by $\xrightarrow{\alpha_{k+1}}_i \circ \cdots \circ \xrightarrow{\alpha_n}_i$.

The action α_k might interact with other threads and therefore is called the *commit* in the database terminology [37]. Actions preceding it are called *pre-commit* actions and gather resources, such as locks. The remaining actions are *post-commit* actions that (should) release these resources. We refer to pre(/post)-commit transitions including source and target states as the *pre(/post) phase*.

4 Dynamic Reduction

The reduction outlined above depends on the identification of movers. And to determine whether a statement is a mover, the analysis has to consider *all other statements in all other threads*. Why is the definition of movers so strong? The answer is that 'movability' has to be preserved in all future computations for the reduction not to miss any relevant behavior.

For instance, consider the system composed of `x=0; y=2` and `y=1; x=y` with initial state $\sigma_0 = (\mathsf{pc}_0, d_0)$, $d_0 = (x = 0, y = 0)$ and $\mathsf{pc}_0 = (1, 1)$ using line numbers as program counters. Fig. 8 shows the TS of this system, from which we can derive that `x:=0` and `y:=1` do not commute except in the initial state (see the diamond structure of the top 3 and the middle state). Now assume, we have a dynamic version of Lipton reduction that allows us to apply the reduction `atomic{x=0; y=2;}` and `atomic{y=1; x=y;}`, but only in the initial state where both `x=0` and `y=1` commute. The resulting reduced system, as shown with bold arrows, now discards various states. Clearly, a safety property

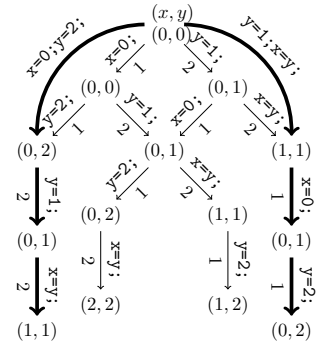


Fig. 8: Transition system of $\xrightarrow{x:=0}_1 \xrightarrow{y:=2}_1 \parallel \xrightarrow{y:=1}_2 \xrightarrow{x:=y}_2$. *Thick lines show an incorrect reduction, missing* $(2, 2)$ *and* $(1, 2)$.

such as $\Box\neg(x = 1 \wedge y = 2)$ is not preserved anymore by such a reduction, even though `x=0` and `y=1` never disable the property (L4 in Sec. 3 holds).

The mover definition comparing all behaviors of all other threads is thus merely a way to (over)estimate the computational future. But we can do better, without precisely calculating the future computations (which would indulge in a task that the reduction is trying to avoid in the first place). For example, unreachable code should not negatively impact movability of statements in the entire program. By the same line of reasoning, we can conclude that lazy initialization procedures (e.g. Fig. 4) should not eliminate movers in the remainder of the program. Intuitively, one can run the program until after initialization, then remove the initialization procedure and restart the verification using that state as the new initial state. Similarly, reading unchanging buckets in the hash table of Fig. 5 should not cause interference. And the dynamically assigned, yet disjoint, pointers of Fig. 6 never overlap, so their dereferences can also become movers *after* their assignment. The current section provides dynamic notion of movability and a generalized reduction theorem that can use this new notion. Proofs of all lemmas and theorems can be found in our technical report [19].

4.1 Dynamic Movers

Recall from the example of Fig. 1 that we introduce branches in order to guide the dynamic reductions. This section formalizes the concept of a dynamic both-moving condition, guarding these branches. We only consider both movers for ease of explanation. Nonetheless, our report [19] considers left and right movers.

Definition 1 (Dynamic both-moving conditions).
A state predicate (a subset of states) c_α is a dynamic both-moving condition for a CFG edge $(l, \alpha, l') \in \delta_i$, if for all $j \neq i$ and $\beta \in \Delta_j$: $(c_\alpha /\!\!/ \xrightarrow{\alpha}_i) \bowtie (c_\alpha /\!\!/ \xrightarrow{\beta}_j)$ and both $\xrightarrow{\alpha}_i$, $\xrightarrow{\beta}_j$ do not disable c_α, i.e. $c_\alpha /\!\!/ \xrightarrow{\beta}_j \setminus\!\!\setminus \overline{c_\alpha} = c_\alpha /\!\!/ \xrightarrow{\alpha}_i \setminus\!\!\setminus \overline{c_\alpha} = \emptyset$.

One key property of a dynamic both-moving condition for $\alpha \in \Delta_i$ is its monotonicity: In the transition system, the condition c_α can be enabled by remote threads ($j \neq i$), but never disabled. While the definition allows us to define many practical heuristics, we have identified the following both-moving conditions as useful. Although our heuristics still rely on static analysis, the required information is easier to establish (e.g. with basic control-flow analysis and the identification of CAS instructions) than for the global mover condition. When static analysis still fails to derive enough information for establishing one of these heuristics, $c_\alpha :=$ false can safely be taken, destining α as a non-mover statically.

Reachability As in Fig. 4, interfering actions, such as the write at label `W`, may become unreachable once a certain program location has been reached. The dynamic condition for the read $\alpha \triangleq \texttt{process(data[i + tid * 512])}_i$ therefore becomes: $c_\alpha := \bigwedge_{j \neq i} \bigwedge_{l \in L(j)} \texttt{pc}_j \neq l$, where $L(j)$ is the set of all locations in V_j that can reach the location with label `W` in V_j. For example, for thread T1 we obtain $c_\alpha :=$ `pc_T2 != a,b,c,d,W` (abbreviated).
Deriving this condition merely requires a simple reachability check on the CFG.

Static pointer dereference If pointers are not modified in the future, then their dereferences commute if they point to different memory locations.
For thread T1 in the pointer example in Fig. 6, we obtain $c_\alpha :=$ `p1 != p2 && pc_T2 != a,b,c` (here `*p++` is the pointer dereference with `p = p1`).

Monotonic atomic A CAS instruction `CAS(p, a, b)` is monotonic, if its expected value `a` is never equal to the value `b` that it tries to write. Assuming that no other instructions write to the location where `p` refers to, this means that once it is set to `b`, it never changes again.
In the hash table example in Fig. 5, there is only a CAS instruction writing to the array `T`. The dynamic moving condition is: $c_\alpha :=$ `T[index] != E`.

Lemma 1. *The above conditions are dynamic both-moving conditions.*

4.2 Instrumentation

Fig. 1 demonstrated how our instrumentation adds branches to dynamically implement the basic single-action rule. Lipton reduction is more complicated. Here, we provide an instrumentation that satisfies the constraints on these phases (see L1–L4 in Sec. 3). Roughly, we transform each CFG $G_i = (V_i, \delta_i)$ into an instrumented $G'_i \triangleq (V'_i, \delta'_i)$ as follows:

1. Replicate all $l_a \in V_i$ to new locations in $V'_i = \{l_a^N, l_a^R, l_a^L, l_a^{R'}, l_a^{L'} \mid l_a \in V_i\}$: Respectively, there are external, pre-, and post- locations, plus two auxiliary pre- and post- locations for along branches.
2. Add edges/branches with dynamic moving conditions according to Table 1.

The rules in Table 1 precisely describe the instrumented edges in G'_i: for each graph part in the original G_i (middle column), the resulting parts of G'_i are shown (right column). As no non-movers are allowed in the post phase, R4 only checks the dynamic moving condition for all outgoing transitions of a post-location l_a^L. If it fails, the branch goes to an external location l_a^N from where the

Table 1: The CFG instrumentation

	$G_i \triangleq (V_i, \delta_i)$	V'_i, δ' in G'_i (pictured)
R1	$\forall (l_a, \alpha, l_b) \in \delta_i$:	$l_a^N \xrightarrow{c_\alpha \,/\!/\, \alpha} l_b^R$, $l_a^N \xrightarrow{\neg c_\alpha \,/\!/\, \alpha} l_b^L$
R2	$\forall (l_a, \alpha, l_b) \in \delta_i$:	$l_a^{R'} \xrightarrow{c_\alpha \,/\!/\, \alpha} l_b^R$, $l_a^{R'} \xrightarrow{\neg c_\alpha \,/\!/\, \alpha} l_b^L$
R3	$\forall l_a \in V_i$:	$l_a^R \xrightarrow{\text{true}} l_a^{R'}$
R4	$\forall l_a \in V_i \setminus LFS_i$:	$l_a^L \xrightarrow{c(l_a)} l_a^{L'}$, $l_a^L \xrightarrow{\neg c(l_a)} l_a^N$ with $c(l_a) \triangleq \bigwedge_{(l_a,\alpha,l_b)\in\delta_i} c_\alpha$
R5	$\forall (l_a, \alpha, l_b) \in \delta_i, l_a \in V_i \setminus LFS_i$:	$l_a^{L'} \xrightarrow{\alpha} l_b^L$
R6	$\forall l_a \in LFS_i$:	$l_a^L \xrightarrow{\text{true}} l_a^N$

actual action can be executed (R1). If it succeeds, then the action commutes and can safely be executed while remaining in the post phase (R5). We do this from an intermediary post location $l_a^{L'}$. Since transitions α thus need to be split up into two steps in the post phase, dummy steps need to be introduced in the pre phase (R1 and R2) to match this (R3), otherwise we lose bisimilarity (see subsequent subsection). As an intermediary pre location, we use $l_a^{R'}$.

All new paths in the instrumented G_i' adhere to the pattern: $l_1^N \xrightarrow{\alpha_1} l_2^R \dots l_k^R \xrightarrow{\alpha_k} l_{k+1}^L \dots l_n^L \xrightarrow{\alpha_n} l_{n+1}^N$. Moreover, using the notion of *location feedback sets* (LFS) defined in Def. 2, R4 and R6 ensure that all cycles in the post phase contain an external state. This is because our reduction theorem (introduced later) allows non-terminating transactions as long as they remain in the pre-commit phase (it thus generalizes L3). Fig. 9 shows a simple example CFG with its instrumentation. The subsequent reduction will completely hide the internal states, avoiding exponential blowup in the TS (see Sec. 4.3).

Definition 2 (LFS). *A location feedback set (LFS) for thread i is a subset $LFS_i \subseteq V_i$ such that for each cycle $C = l_1, .., l_n, l_1$ in G_i it holds that $LFS_i \cap C \neq \emptyset$. The corresponding (state) feedback set (FS) is: $\mathcal{C}_i \triangleq \{(\mathsf{pc}, d) \mid \mathsf{pc}_i \in LFS_i)\}$.*

Corollary 1 ([30]). *$\bigcup_i \mathcal{C}_i$ is a feedback set in the TS.*

The instrumentation yields the following 3/4-partition of states for all threads i:

$$\mathcal{E}_i \triangleq \{(\mathsf{pc}, d) \mid \mathsf{pc}_i \in \{l_{sink}^N, l_{sink}^R, l_{sink}^L\}\} \quad \text{(Error)} \quad (2)$$

$$\mathcal{R}_i \triangleq \{(\mathsf{pc}, d) \mid \mathsf{pc}_i \in \{l^R, l^{R'}\}\} \setminus \mathcal{E}_i \quad \text{(Pre-commit)} \quad (3)$$

$$\mathcal{L}_i \triangleq \{(\mathsf{pc}, d) \mid \mathsf{pc}_i \in \{l^L, l^{L'}\}\} \setminus \mathcal{E}_i \quad \text{(Post-commit)} \quad (4)$$

$$\mathcal{F}_i \triangleq \{(\mathsf{pc}, d) \mid \mathsf{pc}_i \in \{l^N\}\} \setminus \mathcal{E}_i \quad \text{(Ext./non-error)} \quad (5)$$

$$\mathcal{N}_i \triangleq \mathcal{F}_i \uplus \mathcal{E}_i \quad \text{(External)} \quad (6)$$

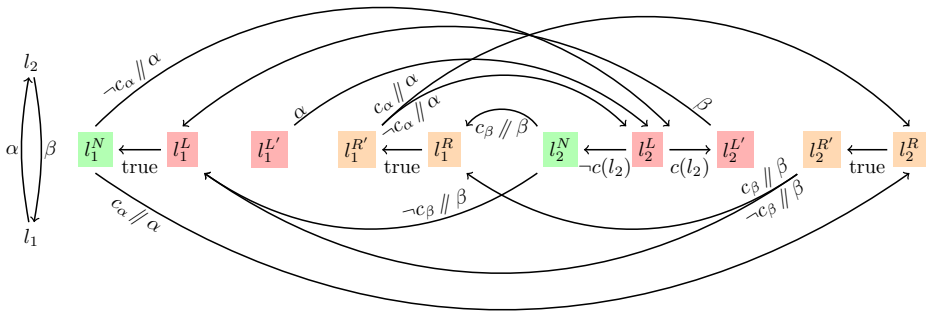
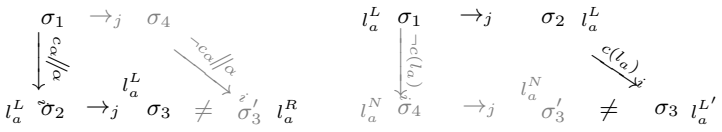


Fig. 9: Instrumentation (right) of a 2-location CFG (left) with LFS = $\{l_1\}$.

The new initial state is (pc'_0, d_0), with $\forall i\colon \mathsf{pc}'_{0,i} = l^N_{0,i}$. Let $\mathsf{Locs}' \triangleq \prod_i V'_i$ and $C' \triangleq (\mathsf{Locs}' \times \mathsf{Data}, \to')$ be the transition system of the instrumented CFG. The instrumentation preserves the behavior of the original system:

Lemma 2. *An error state is $\to$-reachable in the original system iff an error state is $\to'$-reachable in the instrumented system.*

Recall the situation illustrated in Fig. 3 within the example in Fig. 1. Rules R1, R2, and R4 of our instrumentation in Table 1 give rise to a similar problem as illustrated in the following.



Hence, our instrumentation introduces non-movers. Nevertheless, we can prove that the target states are bisimilar. This enables us to introduce a weaker notion of commutativity up to bisimilarity which effectively will enable a reduction along one branch (where reduction was not originally possible). The details of the reduction are presented in the following section. We emphasize that our implementation does not introduce any unnecessary non-movers.

4.3 Reduction

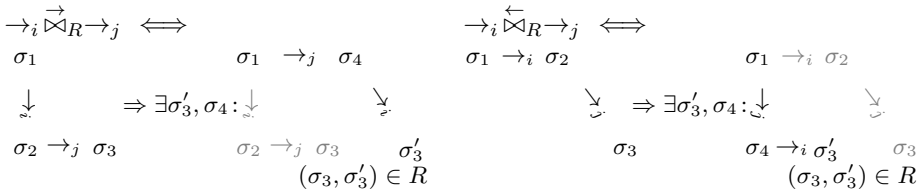
We now formally define the notion of thread bisimulation required for the reduction, as well as commutativity up to bisimilarity.

Definition 3 (thread bisimulation). *An equivalence relation R on the states of a TS $(S, \to)$ is a thread bisimulation iff*

$$\forall \sigma, \sigma', \sigma_1, i\colon \begin{array}{c} \sigma \;\; \to_i \;\; \sigma_1 \\ \big| R \\ \sigma' \end{array} \quad \Rightarrow \exists \sigma'_1\colon \quad \begin{array}{c} \sigma \;\; \to_i \;\; \sigma_1 \\ \big| R \qquad \big| R \\ \sigma' \;\; \to_i \;\; \sigma'_1 \end{array}$$

Standard bisimulation [35,38] is an equivalence relation R which satisfies the property from Def. 3 when the indexes i of the transitions are removed. Hence, in a thread bisimulation, in contrast to standard bisimulation, the transitions performed by thread i will be matched by transitions performed by the same thread i. As we only make use of thread bisimulations, we will often refer to them simply as bisimulations.

Definition 4 (commutativity up to bisimulation). *Let R be a thread bisimulation on a TS $(S, \to)$. The right and left commutativity up to R of the transition relation $\to_i$ with $\to_j$, notation $\to_i \overset{\to}{\bowtie}_R \to_j$ / $\to_i \overset{\leftarrow}{\bowtie}_R \to_j$ are defined as follows.*



Our reduction works on parallel transaction systems (PT), a specialized TS. While its definition (Def. 5) looks complicated, most rules are concerned with ensuring that all paths in the underlying TS form transactions, i.e. that they conform to the pattern $\sigma_1 \xrightarrow{\alpha_1} \sigma_2 \ldots \sigma_k \xrightarrow{\alpha_k} \sigma_{k+1} \ldots \sigma_n \xrightarrow{\alpha_n} \sigma_{n+1}$, where α_k is the non-mover, etc. We have from the perspective of thread i that: σ_1 and σ_{n+1} are *external*, $\forall 1 < x \leq k \colon \sigma_x$ *pre-commit*, and $\forall k < x \leq n \colon \sigma_x$ *post-commit* states. The rest of the conditions ensure bisimilarity and constrain error locations.

The reduction theorem, Th. 1, then tells us that reachability of error states is preserved (and reflected) if we consider only PT-paths between globally external states $\mathcal{N}$. The reduction thus removes all internal states $\mathcal{I}$ where $\mathcal{I} \triangleq \bigcup_i \mathcal{I}_i$ and $\mathcal{I}_i \triangleq \mathcal{L}_i \cup \mathcal{R}_i$ (at least one internal phase).

Definition 5 (transaction system). *A parallel transaction system PT is a transition system $TS = (S, \to)$ whose states are partitioned in three sets of phases and error states in one of the phases, for each thread i. For each thread i, there exists a thread bisimulation relation $\cong_i$. Additionally, the following properties hold (for all i, all $j \neq i$):*

1. $S = \mathcal{R}_i \uplus \mathcal{L}_i \uplus \mathcal{N}_i$ *and* $\mathcal{N}_i = \mathcal{E}_i \uplus \mathcal{F}_i$ *(the 3/4-partition)*
2. $\forall \sigma \in \mathcal{L}_i \colon \exists \sigma' \in \mathcal{N}_i \colon \sigma \to_i^+ \sigma'$ *(post phases terminate)*
3. $\to_i \subseteq \mathcal{L}_j^2 \cup \mathcal{R}_j^2 \cup \mathcal{E}_j^2 \cup \mathcal{F}_j^2$ *(i preserves j's phase)*
4. $\mathcal{E}_i \mathbin{/\!\!/} \to_i \mathbin{\backslash\!\!\backslash} \overline{\mathcal{E}_i} = \emptyset$ *(local transitions preserve errors)*
5. $\mathcal{L}_i \mathbin{/\!\!/} \to_i \mathbin{\backslash\!\!\backslash} \mathcal{R}_i = \emptyset$ *((locally) post does not reach pre)*
6. $\cong_i \subseteq \mathcal{E}_i^2 \cup \overline{\mathcal{E}_i}^2$ *(bisimulation preserves (non)errors)*
7. $\cong_i \subseteq \mathcal{L}_j^2 \cup \mathcal{R}_j^2 \cup \mathcal{E}_j^2 \cup \mathcal{F}_j^2$ *($\cong_i$ entails j-phase-equality)*
8. $(\to_i \mathbin{\backslash\!\!\backslash} \mathcal{R}_i) \overrightarrow{\bowtie}_{\{j\}} \to_j$ *(i to pre right commutes up to $\cong_j$ with j)*
9. $(\mathcal{L}_i \mathbin{/\!\!/} \to_i) \overleftarrow{\bowtie}_{\{i,j\}} \to_j$ *(i from post left commutes up to $\cong_{\{i,j\}}$ with j)*

In item 8 and item 9, $\overrightarrow{\bowtie}_Z$ and $\overleftarrow{\bowtie}_Z$ (for a set of threads Z) are short notations for $\overrightarrow{\bowtie}_{\cong_Z}$ and $\overleftarrow{\bowtie}_{\cong_Z}$, respectively, with $\cong_Z$ being the transitive closure of the union of all $\cong_i$ for $i \in Z$.

Theorem 1. *The block-reduced transition relation $\rightsquigarrow$ of a parallel transaction system $PT = (S, \to)$ is defined in two steps:*

$$\hookrightarrow_i \triangleq \mathcal{N}_{j \neq i} \mathbin{/\!\!/} \to_i \qquad \text{(}i\text{ only transits when all }j\text{ are in external)}$$

$$\rightsquigarrow_i \triangleq \mathcal{N}_i \mathbin{/\!\!/} (\hookrightarrow_i \mathbin{\backslash\!\!\backslash} \overline{\mathcal{N}_i})^* \hookrightarrow_i \mathbin{\backslash\!\!\backslash} \mathcal{N}_i \qquad \text{(block steps skip internal states } \overline{\mathcal{N}_i}\text{)}$$

Let $\rightsquigarrow \triangleq \bigcup_i \rightsquigarrow_i$, $\mathcal{N} \triangleq \bigcap_i \mathcal{N}_i$ and $\mathcal{E} \triangleq \bigcup_i \mathcal{E}_i$. We have $p \to^* q$ for $p \in \mathcal{N}$ and $q \in \mathcal{E}$ if and only if $p \rightsquigarrow^* q'$ for $q' \in \mathcal{E}$.

Our instrumentation from Table 1 in Sec. 4.2 indeed gives rise to a *PT* (Lemma 3) with the state partitioning from (Eq. 2–6). The following equivalence relation $\sim_i$ over locations becomes the needed bisimulation $\cong_i$ when lifted to states. (The locations in the rightmost column of Table 1 are intentionally positioned such that vertically aligned locations are bisimilar.)

$$\sim_i \triangleq \left\{(l^X, l^Y) \mid l \in V_i \wedge X, Y \in \{L, R\}\right\} \cup \left\{(l^X, l^Y) \mid l \in V_i \wedge X, Y \in \{N, R', L'\}\right\}$$
$$\cong_i \triangleq \left\{((\mathsf{pc}, d), (\mathsf{pc}', d')) \mid d = d' \wedge \mathsf{pc}_i \sim_i \mathsf{pc}'_i \wedge \forall j \neq i : \mathsf{pc}_j = \mathsf{pc}'_j\right\}$$

The dynamic both-moving condition in Def. 1 is sufficient to prove (item 8–9). The LFS notion in Def. 2 is sufficient to prove post-phase termination (item 2).

Lemma 3. *The instrumented TS* $C' = (\mathsf{Locs}' \times \mathsf{Data}, \to')$ *is a PT.*

All of the apparent exponential blowup of the added phases ($5^{|\mathsf{Threads}|}$) is hidden by the reduction as $\rightsquigarrow$ only reveals external states $\mathcal{N} \triangleq \bigcap_i \mathcal{N}_i$ (note that $S = \mathcal{I} \uplus \mathcal{N}$) and there is only one instrumented external location (replicated sinks can be eliminated easily with a more verbose instrumentation).

5 Block Encoding of Transition Relations

We implement the reduction by encoding a transition relation for symbolic model checking. Transitions encoded as SMT formulas may not contain cycles. Although our instrumentation conservatively eliminates cycles in the post-commit phase of transactions with external states, cycles (without external locations) can still occur in the pre-phase. To break these remaining cycles, we use a refined location feedback set LFS'_i of the instrumented CFG without external locations $G'_i \setminus \left\{l^N \in V'_i\right\}$ (this also removes edges incident to external locations).

Now, we can construct a new block-reduced relation $\twoheadrightarrow$. It resembles the definition of $\rightsquigarrow$ in Th. 1, except for the fact that the execution of thread i can be interrupted in an internal state $\mathcal{C}'_i$ (LFS'_i lifted to states) in order to break the remaining cycles.

$$\twoheadrightarrow \triangleq \bigcup_i \twoheadrightarrow_i \text{ , where } \twoheadrightarrow_i \triangleq \mathcal{X}_i \mathbin{/\!\!/} (\hookrightarrow_i \mathbin{\backslash\!\!\backslash} \overline{\mathcal{X}_i})^* \hookrightarrow_i \mathbin{\backslash\!\!\backslash} \mathcal{X}_i \quad \text{with} \quad \mathcal{X}_i \triangleq \mathcal{N}_i \cup \mathcal{C}'_i$$

Here, the use of $\hookrightarrow_i$ (from Th. 1) warrants that only thread i can transit from the newly exposed internal states $\mathcal{C}'_i \subseteq \mathcal{N}_{j \neq i}$. Therefore, by carefully selecting the exposed locations of $\mathcal{C}'_i$, e.g. only l^R_a, the overhead is limited to a factor two.

To encode $\twoheadrightarrow$, we identify blocks of paths that start and end in external or LFS locations, but do not traverse external or LFS locations and encode them using large blocks [5]. This automatically takes care of disallowing intermediate states, except for the states $\mathcal{C}'_i$ exposed by the breaking of cycles. At the corresponding locations, we thus add constraints to the scheduler encoding to only allow the current thread to execute. To support `pthreads` constructs, such as locks and thread management, we use similar scheduling mechanisms.

6 Experiments

We implemented the encoding with dynamic reduction in the Vienna Verification Tool (VVT) [25,24]. VVT implements CTIGAR [7], an IC3 [9] algorithm with predicate-abstraction, and bounded model checking (BMC) [26]. VVT came fourth in the concurrency category of SVComp 2016 [3] the first year it participated, only surpassed by tools based on BMC or symbolic simulation.

We evaluated our dynamic reductions on the running examples and compared the running time of the following configurations:

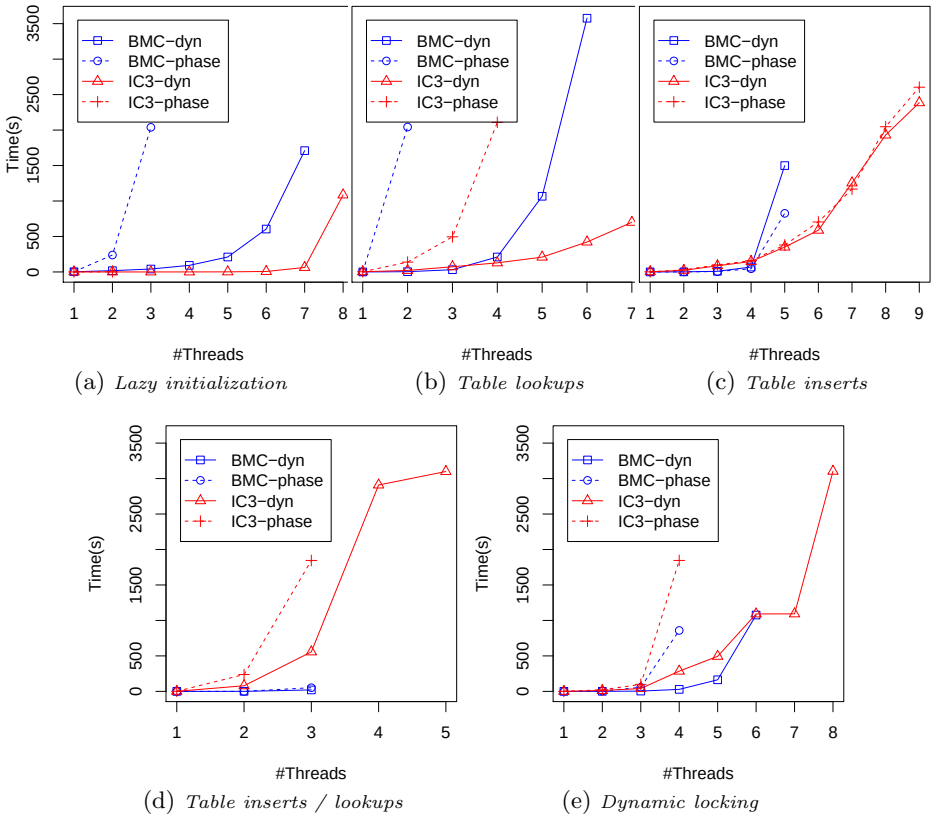
- BMC with all dynamic reductions (*BMC-dyn* in the graphs);
- BMC with only static reductions and phase variables from [17] (*BMC-phase*);
- IC3 with all dynamic reductions (*IC3-dyn*); and
- IC3 with only static reductions and phase variables from [17] (*IC3-phase*).

We used a one-hour time limit for each run and ran each instance four times. Variation over the four runs was insignificant, so we omit plotting it. Missing values in the graphs indicate a timeout. The whole process, including heuristic derivation, instrumentation and encoding, is automated.

Lazy initialization. We implemented a version of the program in Fig. 4 where the function `process` counts array elements. As verification condition, we used the correct total count. As no other heuristic applies, only the reachability heuristic can contribute. Fig. 10a shows that both BMC and IC3 benefit enormously from the obtained dynamic reductions: With static reductions, IC3 can only verify the program for one thread and BMC for three, while with dynamic reduction, both BMC and IC3 scale to seven threads.

Hashtable. The lockless hash table of Fig. 5 is used in the following three experiments. In each, we expected benefits from the monotonic atomic heuristic.

1. Every thread attempts to insert an already-present element into the table. The verification condition is that every *find-or-put* operation returns `FOUND`. Since a successful lookup operation doesn't change the hash table, the dynamic reduction takes full effect: While the static reduction can only verify two threads for BMC and four for IC3, the dynamic reduction can handle six threads for BMC and more than seven for IC3.
2. Each thread inserts one element into an empty hash table. The verification condition is that all inserted elements are present in the hash table after all threads have finished executing. We now see in Fig. 10c that the dynamic reduction benefits neither BMC nor IC3. This is because every thread changes the hash table thus forcing an exploration of all interleavings.
   The overhead of using dynamic reductions, while significant in the BMC case, seems to be non-existent in IC3.
3. Since both of the previous cases can be considered corner-cases (the hash table being either empty or full), this configuration has half of the threads inserting values already present while the other half insert new values.
   While the difference between static and dynamic reductions is not as extreme as before, we can still see that we profit from dynamic reductions, being able to verify two more threads in the IC3 case.



(a) *Lazy initialization* (b) *Table lookups* (c) *Table inserts*

(d) *Table inserts / lookups* (e) *Dynamic locking*

Fig. 10: Hash table and dynamic locking benchmark results

Load balancing. We used the load-balancing example (Fig. 6), expecting the static pointer heuristic to improve the dynamic reductions. We verified that the computed sum of the counters is indeed the expected result. Our experiment revealed that dynamic reductions reduce the runtime from 15 minutes to 97 seconds for two threads already.

Dynamic locking. In addition to the earlier examples of Sec. 2, we also study the effect of lock pointer analysis. To this end, we created a parallel program in which multiple threads use a common lock to access two global variables. To simulate locks in complex object structures (that are common, but impossible to track for static analysis), the single lock these threads use is randomly picked, similar to how the work load is assigned in Fig. 6. We extended our static pointer dereference heuristic to also determine whether other critical sections with the same conflicting operations are protected by the same lock, potentially allowing the critical section to become a single transaction. In the critical section we again count. The total is used as verification condition. Fig. 10e shows that the dynamic reduction indeed kicks in and benefits both IC3 and BMC.

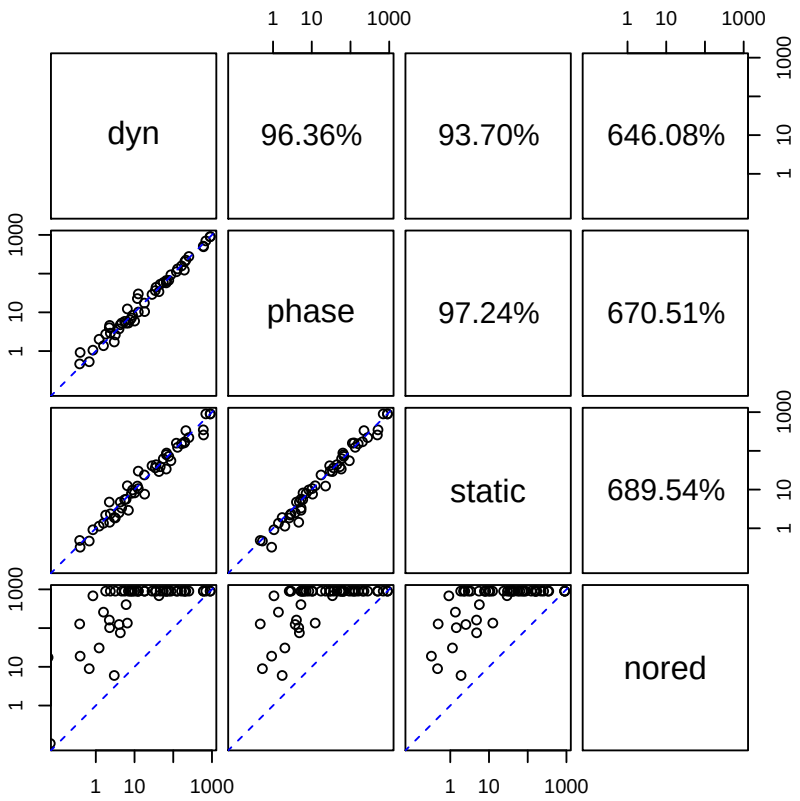


Fig. 11: Scatterplots comparing runtimes for all combinations of reduction variants on SVComp benchmarks. The upper half shows relative accumulated runtimes for these combinations.

SVComp. We also ran our IC3 implementation on the `pthread-ext` and `pthread-atomic` categories of the *software verification competition* (SVComp) benchmarks [4,2]. In instances with an unbounded number of threads, we introduced a limit of three threads. To check the effect of different reduction-strategies on the verification time, we tested the following reductions:

***dyn*:** Dynamic with all heuristics from Sec. 4.1.
***phase*:** Dynamic phases only (equal to [17]).
***static*:** Static (as in Sec. 3).
***nored*:** No reduction, all interleavings considered.

Fig. 11 shows that static Lipton reduction yields an average six-fold decrease in runtime when compared to no reduction. Enabling the various dynamic improvements (*dyn*, *phase*) does not show improvement over the static case (*static*), since most of the benchmarks are either too small or do not have opportunities for reductions, but also not much overhead (up to 7%). Comparing the *nored* case with the other cases shows the benefit of removing intermediate states.

7 Related Work

Lipton's reduction was refined multiple times [32,21,13,11,41]. It has recently been applied in the context of compositional verification [40]. Qadeer and Flanagan [17] introduce dynamic phase variables to identify internal and external states. They also provided a dynamic solution for determining locked regions. Their approach, however, does not solve the examples featured in the current paper. Moreover, in [17], the phases of target locations of non-deterministic transitions are required to agree. This restriction is not present in our encoding.

Grumberg et al. [22] present underapproximation-widening, which iteratively refines an under-approximated encoding of the system. In their implementation, interleavings are constrained to achieve the under-approximation. Because refinement is done based on verification proofs, irrelevant interleavings will never be considered. The technique currently only supports BMC and the implementation is not available, so we did not compare against it.

Kahlon et al. [28] extend the dynamic solution of [17], by supporting a strictly more general set of lock patterns. They incorporate the transactions into the stubborn set POR method [43] and encode these in the transition relation in similar fashion as in Alur et al. [1]. Unlike ours, their technique does not support other constructs than locks.

While in fact it is sufficient for item 2 of Def. 5 to pinpoint a single state in each bottom SCC of the CFG, we use feedback sets because the encoding in Sec. 5 also requires them. Moreover, we take a syntactical definition for ease of explanation. Semantic heuristics for better feedback sets can be found in [30] and can easily be supported via state predicates. (Further optimizations are possible [30].) Obtaining the smallest (vertex) LFS is an NP-complete problem well known in graph theory [8]. As CFGs are simple graphs, estimations via basic DFS suffice. (In POR, similar approaches are used for *the ignoring problem* [44,36].)

Elmas et al. [15] propose dynamic reductions for type systems, where the invariant is used to weaken the mover definition. The over-approximations performed in IC3, however, decrease the effectiveness of such approaches.

In POR, similar techniques have been employed in [14] and the earlier-mentioned *necessary enabling sets* of [20,42]. Completely dynamic approaches exist [16], but symbolic versions remain highly static [1]. Notable exceptions are peephole and monotonic POR by Wang et al. [45,29]. Like sleep sets [20], however, these only reduce the number of transitions—not states—which is crucial in e.g. IC3 to cut counterexamples to induction [9]. Cartesian POR [23] is a dynamic form of Lipton reduction for explicit-state model checking.

8 Conclusions

Our work provides a novel dynamic reduction for symbolic software model checking. To accomplish this, we presented a reduction theorem generalized with bisimulation, facilitating various dynamic instrumentations as our heuristics show. We demonstrated its effectiveness with an encoding used by the BMC and IC3 algorithms in the model checker VVT.

References

1. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV*, volume 1254 of *LNCS*, pages 340–351. Springer, 1997.
2. Dirk Beyer. The Software Verification Competition website. http://sv-comp.sosy-lab.org/2016/.
3. Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In *TACAS*, volume 9636 of *LNCS*, pages 887–904. Springer, 2016.
4. Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses report on sv-comp 2016. In *TACAS*, volume 9636 of *LNCS*, pages 887–904. Springer, 2016.
5. Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FM-CAD*, pages 25–32. IEEE, 2009.
6. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification. In *CAV*, volume 4590 of *LNCS*, pages 504–518. Springer, 2007.
7. Johannes Birgmeier, Aaron Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *CAV*, volume 8559 of *LNCS*, pages 829–846. Springer, 2014.
8. John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.
9. Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
10. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *TACAS*, volume 8413 of *LNCS*, pages 46–61. Springer, 2014.
11. Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR*, volume 1466 of *LNCS*, pages 317–331. Springer, 1998.
12. Dimitar Dimitrov et al. Commutativity race detection. In *ACM SIGPLAN Notices*, volume 49 (6), pages 305–315. ACM, 2014.
13. Thomas W. Doeppner, Jr. Parallel program correctness through refinement. In *POPL*, pages 155–169. ACM, 1977.
14. Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *FMSD*, 25(2-3):199–240, 2004.
15. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15. ACM, 2009.
16. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, volume 40 (1), pages 110–121. ACM, 2005.
17. Cormac Flanagan and Shaz Qadeer. Transactions for software model checking. *ENTCS*, 89(3):518 – 539, 2003. Software Model Checking.
18. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.
19. Henning Gïther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic Reductions for Model Checking Concurrent Software. *CoRR*, abs/1611.09318, 2016. https://arxiv.org/abs/1611.09318.
20. Patrice Godefroid, editor. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *LNCS*. Springer, 1996.

21. Pascal Gribomont. Atomicity refinement and trace reduction theorems. In *CAV*, volume 1102 of *LNCS*, pages 311–322. Springer, 1996.
22. Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, pages 122–131. ACM, 2005.
23. Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *SPIN*, volume 4595 of *LNCS*, pages 95–112. Springer, 2007.
24. Henning Günther. The Vienna Verification Tool website. http://vvt.forsyte.at/. Last accessed: 2016-11-21.
25. Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna Verification Tool: IC3 for parallel software. In *TACAS*, volume 9636 of *LNCS*, pages 954–957. Springer, 2016.
26. Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In *SPIN*, pages 40–47. ACM, 2014.
27. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
28. Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *CAV*, volume 4144 of *LNCS*, pages 286–299. Springer, 2006.
29. Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.
30. Robert Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *TACAS*, volume 1384 of *LNCS*, pages 345–357. Springer, 1998.
31. A.W. Laarman, J.C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD*, pages 247–255. IEEE-CS, 2010.
32. Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical report, Cornell University, 1989.
33. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Comm. of the ACM*, 18(12):717–721, 1975.
34. Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
35. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
36. Ratan Nalumasu and Ganesh Gopalakrishnan. An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation. *FMSD*, 20(3):231–247, 2002.
37. Christos Papadimitriou. *The theory of database concurrency control*. Principles of computer science series. Computer Science Pr., 1986.
38. David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.
39. Doron Peled. All from One, One for All: on Model Checking Using Representatives. In *CAV*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
40. Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. Reduction for compositional verification of multi-threaded programs. In *FMCAD*, pages 187–194. IEEE, 2014.
41. Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. In *TACAS*, volume 2619 of *LNCS*, pages 489–504. Springer, 2003.
42. A. Valmari. Eliminating Redundant Interleavings During Concurrent Program Verification. In *PARLE*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.

43. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *ICATPN/APN'89*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.
44. Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer, 1990.
45. Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, volume 4963 of *LNCS*, pages 382–396. Springer, 2008.