BAM: Efficient Model Checking for Barriers

Michalis Kokologiannakis and Viktor Vafeiadis

MPI-SWS, Germany
{michalis,viktor}@mpi-sws.org

Abstract. Stateless Model Checking (SMC) and Dynamic Partial Order Reduction (DPOR) are prominent techniques that are often used together to verify safety properties of concurrent programs under a variety of different memory models. Although existing SMC/DPOR implementations excel at verifying parallel algorithms, they scale extremely poorly once *barriers* are used to synchronize the participating threads.
In response, we develop BAM (Barrier-Aware Model-checker), a DPOR extension that explores exponentially fewer executions for programs that employ synchronization schemes involving barriers. We have implemented BAM in a verification tool for C programs, and show that it greatly outperforms the state-of-the-art for programs with barriers.

1 Introduction

Barriers (as in e.g., `pthread_barrier` [24]) are synchronization primitives used to ensure that the execution of a program will continue only after all threads have reached a certain point ("a barrier"). Their usage is best understood with an example:

$$\begin{array}{l}
\qquad\qquad\quad \texttt{barrier\_init}(b, N); \\
\left.\begin{array}{l} m[1] := \ldots; \\ \texttt{barrier\_wait}(b); \\ n[1] := \ldots; \end{array}\right\| \ \ldots \ \left\|\begin{array}{l} m[N] := \ldots; \\ \texttt{barrier\_wait}(b); \\ n[N] := \ldots; \end{array}\right.
\end{array} \qquad (\textsc{Barrier-}N\textsc{-Sync})$$

In this program, the main thread first initializes a barrier object to N, indicating that N threads will meet together ("rendezvous") at the barrier. Each thread calculates a part of the array m, and waits for all the other threads using a `barrier_wait` call: no thread gets past `barrier_wait` until all threads have executed their respective `barrier_wait` call. After all threads have met at the barrier, each thread continues and calculates a part of the array n, which (potentially) uses the array m that was calculated in the previous step. Such iterative parallel computations are common in scientific applications, e.g., simulations.

More generally, barriers are useful when we want to wait for the threads to perform some calculations before continuing. Upon continuation, all calculations performed by one thread will be visible to all other threads. In contrast to joining the threads, using barriers does not cause the threads to be terminated, but rather blocked; this can be crucial for performance reasons.

But while the usage of barriers is straightforward, verifying programs with barriers is not always so. Suppose that we want to verify the BARRIER-N-SYNC program from above automatically, and that we want to use *Stateless Model Checking* (SMC) [12, 21] coupled with *Dynamic Partial Order Reduction* (DPOR) [11, 1] to do so. This combination has been proven to scale very well for parallel programs [17, 13, 22], and also takes into account the effects of the underlying weak memory model [2, 3, 4, 14, 16].

Alas, all existing SMC/DPOR techniques explore an exponential number of executions for this program, as they examine all possible orderings in which different threads arrive at the barrier (see §2). Even worse, they do so even though the order in which the threads rendezvous is irrelevant. In fact, the order in which threads reach the barrier is not even observable by the user program; the only thing that *is* observable according to the `pthread_barrier` documentation [24], is whether a thread was the last one to reach the barrier. However, for the programs we are aware of, even that condition is never used.

Leveraging this insight, we develop BAM (Barrier-Aware Model-checker), a memory-model-agnostic DPOR extension that reconciles SMC/DPOR with barriers. By avoiding the exploration of executions that only differ in the order in which threads execute `barrier_wait`, BAM explores exponentially fewer executions than state-of-the-art SMC/DPOR tools. Concretely, we make the following contributions:

§3 We introduce BAM, an SMC/DPOR extension that does not order calls to `barrier_wait`, and yet models barrier semantics correctly: all instructions executed after a rendezvous at a barrier will see the effects of all instructions executed before the rendezvous.

§4 We implement BAM as an extension of the state-of-the-art GENMC model checker [16], and show that BAM is exponentially faster than vanilla GENMC in programs with barriers.

We start with an overview of how barriers are handled by the state-of-the-art stateless model checkers. To simplify the presentation, we assume a model of sequential consistency (SC) [20]. Our results carry over to all other axiomatic memory models.

2 State-of-the-Art

Why is it that SMC/DPOR experiences an exponential slowdown in programs with barriers? To answer this question, we first have to review the fundamentals of SMC/DPOR.

2.1 SMC and DPOR

SMC verifies a program by checking all of its thread interleavings. For example, for the W+R+W program below, an SMC algorithm would enumerate all 6 interleavings of the program, and validate that all of them satisfy the desired properties.

$$x := 1 \parallel r := x \parallel y := 1 \qquad (\text{W+R+W})$$

Of course, enumerating interleavings does not scale as programs become larger. Hence, SMC is usually coupled with *Dynamic Partial Order Reduction* (DPOR) [11, 1, 16], which avoids exploring an interleaving if an equivalent one has already been explored. DPOR considers two interleavings equivalent if one can be obtained from the other by swapping adjacent, non-conflicting instructions. While many notions of conflict have been proposed in the literature [1, 8, 9, 16, 10], the simplest one considers two instructions as conflicting if they access the same memory location, and at least one of them is a write. For W+R+W, the only conflicting instructions are $x := 1$ and $r := x$. Thus, a DPOR algorithm would verify the program by exploring only 2 interleavings: one where $x := 1$ is executed before $r := x$, and one where the order is reversed.

SMC/DPOR provides an excellent solution for verifying concurrent programs as it does not explicitly store the states of the program that have already been visited, and its notion of conflict has been extended to weak memory models [2, 25, 3, 4, 14, 16]. In particular, SMC/DPOR scales very well for programs with few conflicts, such as parallel algorithms. We will not go into details of how SMC/DPOR works, as SMC/DPOR has been thoroughly studied in the literature, and the exact details are not important for this paper. Instead, we only provide a high-level overview of DPOR later on (see § 3.4), and refer interested readers to Kokologiannakis et al. [16].

2.2 Barriers in SMC/DPOR

The reason why barriers and SMC/DPOR do not work well together is that barriers *inhibit* DPOR. Existing DPOR algorithms consider `barrier_wait` calls conflicting, and thus explore an exponential number of interleavings, even for a barrier program doing the bare minimum:

$$\begin{array}{c} \texttt{barrier\_init}(b, N); \\ \texttt{barrier\_wait}(b); \parallel \ldots \parallel \texttt{barrier\_wait}(b); \end{array} \qquad (\text{BARRIER-}N)$$

For BARRIER-N, an SMC/DPOR algorithm would explore $N!$ executions, effectively rendering DPOR a useless addition to SMC.

To understand why barriers are considered conflicting operations by DPOR, however, we have to examine how barriers are implemented. Typically, barriers are implemented using condition variables or futexes: a thread executing `barrier_wait` acquires a lock, manipulates a variable indicating the number of threads that have reached the barrier, and then waits on a futex/condition variable. Such implementations, however, while standard for barrier libraries, are suboptimal for model checking: each `barrier_wait` call would boil down to many different instructions, thus unnecessarily increasing the number of different events a model checker would have to generate.

```
barrier_init($b, N$) :
  $b := N$;

barrier_wait($b$) :
  atomic { if ($b = 1$)  $b := N$ else  $b := b-1$;  }; assume($b = N$);
```

Fig. 1. Implementation of `barrier_init` and `barrier_wait`.

Since we are only interested in verifying programs that *use* barriers, we can get away with a much more abstract barrier implementation, such as the one in Fig. 1. We model each `barrier_init`(b, N) as a plain write that initializes a shared variable b to N, and each `barrier_wait`(b) as an atomic read-modify-write (RMW) instruction followed by an `assume` instruction. For the `barrier_wait` call, the RMW instruction decrements b each time it is called, apart from when the value read is 1, at which point it resets is back to N (so that the barrier can be subsequently reused). For the same call, the `assume` reads b and blocks the calling thread if the value read was different than N.

Given this implementation, it becomes clear that programs like BARRIER-N lead to an exponential blowup in the state space. Since the RMW instructions all write to the same location (b), they are considered conflicting, and so the model checker will examine all their $N!$ possible orderings. In addition to these $N!$ executions, some state-of-the-art DPOR implementations, such as GENMC [16], may also consider an exponential number of blocked executions (see §4).

3 BAM: Barrier Model Checking

We now present BAM and explain how it improves over baseline DPOR for programs with barriers. After presenting the key idea behind BAM (§ 3.1), we provide a formal framework in which the executions of a program can be modeled, and show how BAM's modeling of barriers leads to exponential savings when verifying programs with barriers, while at the same time maintaining the guarantees that barriers provide (§ 3.3).

3.1 Key Idea

We note that, although the barrier implementation effectively records the order in which different thread call `barrier_wait` by counting the number of threads that have joined the barrier, programs that use barriers do not care about this order. In fact, even though barrier implementations typically provide a distinct value returned by the `barrier_wait` call that resets the barrier to its initial value, the user programs we are aware of do not make use of that.

We further observe that programs using barriers typically initialize the barrier to the number of threads in the system, and so there is never a case with more parallel calls to `barrier_wait` than the barrier's initial value. Intuitively, this is because the standard scenario for barrier synchronization is to arrange a rendezvous between all threads participating in a parallel computation. With that in mind, it does not really make sense to initialize a barrier with a value smaller than the number of threads calling `barrier_wait`, as that would imply that only some threads will be unblocked after reaching the barrier, while the others will remain blocked.

The key insight behind BAM is that, for programs satisfying the two conditions described above, tracking the order between `barrier_wait` calls is unnecessary. BAM models `barrier_wait` calls as *dummy events* that are not considered conflicting, thus enabling the underlying DPOR algorithm to consider fewer executions. More specifically, when a thread executes `barrier_wait` it simply checks how many threads have reached the barrier: if not all threads have arrived, the thread blocks; otherwise all program threads unblock and continue their execution. Notice that, when all threads unblock, all the instructions before the respective `barrier_wait` statements will have been executed, thereby satisfying the fundamental guarantee provided by barriers i.e., instructions executed after the threads have rendezvoused will see the effects of the instructions executed before the rendezvous.

Let us now make the above idea formal in the framework of axiomatic memory models.

3.2 Execution Graphs

Although the executions of a concurrent program under SC are usually thought of as interleavings, we model them using *execution graphs* [7]. Execution graphs allow for a flexible formalization that can easily be extended to weak memory models, but also, as we will shortly see, abstract away the notion of a "conflict" used by the DPOR algorithm.

Execution graphs have two basic components:

(i) a set of events (nodes), modeling the memory accesses performed by the program, and
(ii) some relations on these events (edges).

Standard relations included in all memory models are the *program order* (`po`) and *reads-from* (`rf`) relations: `po` relates events in the same thread according to their serial execution order, while `rf` relates reads to writes they are reading from. In this paper, we also assume the existence of a *happens-before* (`hb`) relation, a strict partial order that includes `po`, and which models ordering due to synchronization between events.

Let us now formally describe events and execution graphs.

Definition 1. *An* event, $e \in \mathsf{Event}$, *is either an initialization event* $\langle \mathtt{init}\ l \rangle \in \mathsf{Event}_0 \subseteq \mathsf{Event}$ *for a location* $l \in \mathsf{Loc}$ *or a thread event* $\langle t, i, lab \rangle$ *where* $t \in \mathsf{Tid}$ *is a thread identifier,* $i \in \mathsf{Idx} \triangleq \mathbb{N}$ *is a serial number inside each thread, and* $lab \in \mathsf{Lab}$ *is a label that takes one of the following forms:*

– *Read label:* $\mathtt{R}(l, v)$ *where* $l \in \mathsf{Loc}$ *is the location accessed, and* $v \in \mathsf{Val} \triangleq \mathbb{Z}$ *is the value read.*

– *Write label:* $\mathtt{W}(l, v)$ *where* $l \in \mathsf{Loc}$ *is the location accessed, and* $v \in \mathsf{Val}$ *is the value written.*
– *Read-modify-write label:* $\mathtt{RMW}(l, v_1, v_2)$ *where* $l \in \mathsf{Loc}$ *is the location accessed,* v_1 *is the value read, and* $v_2 \in \mathsf{Val}$ *is the value written. This label models a single atomic RMW operation.*
– *Error label:* `error`*, denoting a safety violation.*

The functions `tid`*,* `idx`*,* `loc`*,* `valr`*,* `valw` *return (when applicable) the thread identifier, serial number, location, read-value and written-value of an event, respectively.*

Given the above representation of events, we induce the *program order*, which is a strict partial order on events given by:

$$\mathsf{po} \triangleq \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \;\cup \left\{ \langle\langle t_1, i_1\rangle, \langle t_2, i_2\rangle\rangle \middle| t_1 = t_2 \wedge i_1 < i_2 \right\}$$

Intuitively, initialization events precede all non-initialization events, while events in the same thread are ordered according to their serial numbers.

Definition 2. *An* execution graph G *consists of:*

1. *a set* $G.\mathtt{E}$ *of events that includes initialization events for all locations accessed by the program, and*
2. *a relation* $G.\mathtt{rf} \subseteq G.\mathtt{E} \times G.\mathtt{E}$*, called the* reads-from *relation, that relates each write event to the same-location reads that read from it.*

We write $G.\mathtt{R}, G.\mathtt{W}$ *to denote the set of events of the respective type (RMW events belong both to* $G.\mathtt{R}$ *and* $G.\mathtt{W}$*), and use subscripts to further restrict these sets (e.g.,* $G.\mathtt{W}_x = \{w \in G.\mathtt{W} \mid \mathtt{loc}(w) = x\}$*).*

Definition 3 (Well-formedness). *An execution graph* G *is* well-formed *if the following hold for* $G.\mathtt{rf}$*:*

1. $\mathtt{rf}$ *only relates writes and reads with matching locations and values, i.e., for every* $\langle w, r\rangle \in G.\mathtt{rf}$ *it is* $w \in G.\mathtt{W}$*,* $r \in G.\mathtt{R}$*,* $\mathtt{loc}(w) = \mathtt{loc}(r)$ *and* $\mathtt{valw}(w) = \mathtt{valr}(r)$*,*
2. $\mathtt{rf}$ *is functional on its range, i.e., if* $\langle w_1, r\rangle, \langle w_2, r\rangle \in G.\mathtt{rf}$ *it is* $w_1 = w_2$*, and*
3. *each read reads a value, i.e.,* $\forall r \in G.\mathtt{R}.\, \exists w.\, \langle w, r\rangle \in G.\mathtt{rf}$*.*

The semantics of a program P is simply given by the set of well-formed execution graphs that satisfy a consistency predicate dictated by the memory-model. For instance, *sequential consistency* (SC) [20] can be defined using a coherence order as follows.

Definition 4 (Coherence order). *A relation* $\mathtt{co}$ *is a* coherence order *for an execution* G *iff* $\mathtt{co}$ *is a strict partial order,* $\mathtt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} G.\mathtt{W}_l \times G.\mathtt{W}_l$*, and for every location* $l \in \mathsf{Loc}$*,* $\mathtt{co}$ *is total on* $G.\mathtt{W}_l$*.*

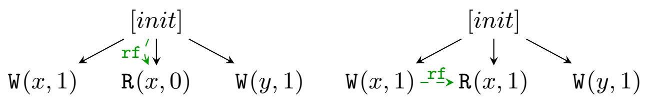


Fig. 2. Execution graphs of W+R+W under SC.

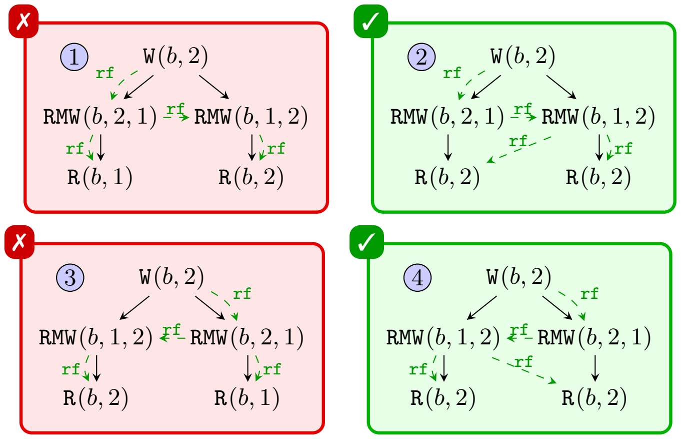


Fig. 3. Execution graphs of BARRIER-N for $N = 2$.

Definition 5 (SC). *G is sequentially consistent, written* $\mathsf{cons}_{\mathsf{SC}}(G)$, *iff there is a coherence order* $\mathtt{co}$ *for G such that* $\mathtt{hb} \triangleq \mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr}$ *is acyclic, where* $\mathtt{fr} \triangleq \{(a, b) \mid a \neq b \wedge \exists c.\, (c, a) \in \mathtt{rf} \wedge (c, b) \in \mathtt{co}\}$ *is the* from-reads *relation.*

As an example, Fig. 2 shows the two sequentially consistent execution graphs of W+R+W. Notice that each of these graphs corresponds to multiple interleavings. In effect, the graphs subsume the notion of a conflict used by DPOR algorithms; each linearization of $\mathtt{hb}$ in these graphs yields a possible interleaving. Thus, an SMC/DPOR algorithm can alternatively be seen as a procedure that verifies a program by enumerating its execution graphs.

As a further example, Fig. 3 shows the sequentially consistent executions of BARRIER-N for $N = 2$ with the conventional modeling of barriers shown in Fig. 1. The two execution graphs on the left are blocked because one `assume` condition is violated. By contrast, the two graphs on the right satisfy the `assume` conditions and are thus non-blocked. SMC/DPOR algorithms will thus have to generate at least the two non-blocked executions, though actual implementations typically generate all four (blocked and non-blocked) executions.

3.3 BAM: Keeping Barriers Unordered

To model barriers, we extend the definition of events (Def. 1) to allow for a new kind of label modeling calls to the `barrier_wait` operation:

– Barrier-wait label: $\mathtt{B}(l)$ where $l \in \mathsf{Loc}$ is the barrier location accessed.

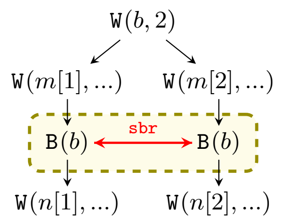


Fig. 4. BAM: Execution graphs of BARRIER-N-SYNC for $N = 2$.

We write $G.\mathtt{B}$ for all the barrier events of an execution graph G. Barrier events do not participate in the `rf` relation of execution graphs.

Keeping barriers unordered by `rf` achieves an exponential reduction in the number of execution graphs of programs like BARRIER-N, as all four graphs of Fig. 3 would correspond to the following single execution graph.

$$\begin{tikzcd}
 & \mathtt{W}(b,2) \arrow[dl] \arrow[dr] & \\
\mathtt{B}(b) & & \mathtt{B}(b)
\end{tikzcd}$$

Treating barrier events as dummy events is inadequate because `barrier_wait` calls also provide some synchronization guarantees. Specifically, every event `po`-before a barrier call is guaranteed to happen before every event `po`-after a barrier call in the same rendezvous. Recall the BARRIER-N-SYNC program from §1:

$$\begin{array}{l||c||l}
m[1] := \ldots; & & m[N] := \ldots; \\
\mathtt{barrier\_wait}(b); & \ldots & \mathtt{barrier\_wait}(b); \\
n[1] := \ldots; & & n[N] := \ldots;
\end{array} \qquad (\text{BARRIER-}N\text{-SYNC})$$

Here, merely treating `B` events as dummy events is unsound. As `B` events do not contribute to `hb` between different threads, each thread will only see its own calculation of a single part of m. By contrast, had we used the conventional barrier representation, the `rf` edges across threads would ensure that the calculation of m is visible when n is calculated.

To solve this problem, we extend the definition of execution graphs (Def. 2) with a new component:

– a partial equivalence relation $G.\mathtt{sbr}$, called *same-barrier-round*, that relates barrier events that synchronize with each other in a rendezvous. Events related by $G.\mathtt{sbr}$ act on the same (barrier) location.

We will use the `sbr` relation to enforce synchronization between the events executed before the threads meet at the barrier, and the events executed after the rendezvous at the barrier. But before presenting how barrier synchronization works, we assume two basic conditions about the `sbr` relation.

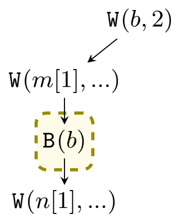
Given a graph G and a barrier location b initialized with value N (i.e., there is a unique write $w \in G.\mathtt{E}$ such that $\mathtt{lab}(w) = \mathtt{W}(b, N)$, and that $\langle w, n\rangle \in G.\mathtt{hb}$, for all $n \in G.\mathtt{B}_b$), we further require that $G.\mathtt{sbr}$ satisfy the following conditions:

$$|G.\mathtt{B}_b \setminus dom(G.\mathtt{sbr})| < N \qquad (\text{SBR-MUST-MEET})$$
$$\forall e \in G.\mathtt{B}_b.\ |\mathrm{succ}_{G.\mathtt{sbr}}(e)| = N \lor \mathrm{succ}_{G.\mathtt{sbr}}(e) = \mathrm{succ}_{G.\mathtt{po}}(e) = \emptyset \qquad (\text{SBR-BLOCK})$$

where $\mathrm{succ}_{\mathsf{r}}(e)$ denotes the set $\{e' \mid \langle e, e' \rangle \in \mathsf{r}\}$, i.e., set of successors of e in r.

The SBR-MUST-MEET condition captures the basic guarantee provided by the barrier implementation that once N `barrier_wait` calls are issued, then they will meet in a rendezvous round. A consistent graph can therefore contain at most $N-1$ barrier calls that do not belong to any barrier round.

The purpose of the SBR-BLOCK condition is twofold. First, it dictates that exactly N calls to `barrier_wait` participate in the same barrier round. That is, each event e either belongs in the same round with N events or does not have any events in the same round. Second, it dictates that no thread is allowed past a `barrier_wait` call before all threads rendezvous at the barrier. In other words, if an event does not participate in a (full) barrier round, it is blocked and has no `po`-successors in the graph. This condition renders graphs like the one below for BARRIER-N-SYNC and $N = 2$ invalid:



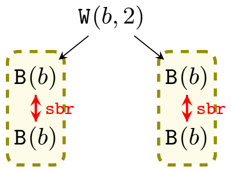
As soon as all threads reach the barrier, all corresponding barrier events become part of `sbr`, and events past the barrier may be added.

We next discuss how barrier synchronization contributes to the happens-before (`hb`) relation. We extend the (model-specific) definition of `hb` with `sbr`; `po` and `po`; `sbr`. That is, a barrier happens before the `po`-successors of any barriers it synchronizes with and after their `po`-predecessors. Since `hb` is transitive, this means that all events that are `po`-before a given barrier round happen before all events that are `po`-after the same barrier round. For example, for the BARRIER-N-SYNC program (cf. Fig. 4), all events `po`-after the highlighted barrier round will also be `hb`-after the events that are `po`-before the highlighted barrier round.

Synchronization ensures that the `barrier_wait` events related by `sbr` belong to the same barrier round. To see how this is achieved, consider the program below where two threads rendezvous at a barrier twice:

```
          barrier_init(b, N);
barrier_wait(b); || barrier_wait(b);        (BARRIER2-N)
barrier_wait(b); || barrier_wait(b);
```

For this example, graphs like the one below, where `sbr` includes `barrier_wait` events from different rounds of the same barrier acquisition, are invalid:



The reason why this graph is invalid, is that $G.\texttt{sbr}; G.\texttt{po}$ is included in $G.\texttt{hb}$. This condition implies that, e.g., the second barrier event of the first thread is hb-before itself (since we can take an sbr; po step), which contradicts the fact that hb is a strict partial order.

Finally, let us end this section by formalizing the conditions under which BAM can be used (see § 3.1). These are expressed by the notion of barrier well-formedness, as described below.

Definition 6 (Barrier Well-formedness). *An execution graph G is* barrier-well-formed *on a barrier location b if $G.\texttt{B}_b = \emptyset$ or if the following hold.*

1. *There is a unique write event $w_0 \in G.\texttt{E} \setminus \mathsf{Event}_0$ with $\texttt{loc}(w_0) = b$.*
2. *w_0 is a plain write event: $\texttt{lab}(w_0) = \texttt{W}(b, N)$ for some $N \in \mathbb{N}$.*
3. *w_0 is hb-before all $\texttt{B}_b$ events: $\langle w_0, e\rangle \in G.\texttt{hb}$ for all $e \in G.\texttt{B}_b$.*
4. *For all $S \subseteq G.\texttt{B}_b$ with $|S| > \texttt{valw}(w_0)$, there exist $e, e' \in S$ s.t. $\langle e, e'\rangle \in G.\texttt{hb}$.*

Barrier well-formedness ensures that there is a unique initializing write for each barrier location, and that no more threads than the barrier's initializing value call `barrier_wait` concurrently. Note that the latter precludes the usage of BAM in programs like the following:

$$\begin{array}{c} \texttt{barrier\_init}(b,2); \\ \texttt{barrier\_wait}(b); \parallel \texttt{barrier\_wait}(b); \parallel \texttt{barrier\_wait}(b); \end{array}$$

That said, as already mentioned, we do not expect such programs to show up often in practice, as they are built on the (not very useful) premise that some subset of the threads meeting at the barrier will continue past the barriers, while the rest will remain blocked.

3.4 BAM: Extending DPOR for Barriers

We now explain how DPOR can be extended to accommodate for BAM.

Algorithm 1 shows the general structure of a DPOR algorithm with BAM's extensions highlighted. VERIFY verifies a program P by enumerating its execution graphs, and ensuring that none of them contains an `error` label. VERIFY achieves this by repeatedly calling VISITONE (Line 4): the latter will explore one full execution of P, and at the same time populate an *environment* Γ (initially empty; cf. Line 2) with alternative exploration options. These exploration options will be subsequently explored by VERIFY (Line 5).

VISITONE is the workhorse of the DPOR algorithm. At each step, as long as G remains consistent according to the memory model (Line 8), VISITONE uses $\mathsf{next}_P(G)$ to extend the current graph G by an event a from a non-blocked

Algorithm 1 Dynamic Partial Order Reduction

```
 1: procedure VERIFY($P$)
 2:     $\langle G, \Gamma \rangle \leftarrow \langle G_0, \emptyset, \Gamma_0 \rangle$
 3:     do
 4:         VISITONE($P, G, \Gamma$)
 5:     while $\langle G, \Gamma \rangle \leftarrow \mathsf{pop}(\Gamma)$
 6: end procedure

 7: procedure VISITONE($P, G, \Gamma$)
 8:     while $\mathsf{cons_m}(G) \land a \leftarrow \mathsf{next}_P(G)$ do
 9:         $G.\mathtt{E} \leftarrow G.\mathtt{E} \cup \{a\}$
10:         if $a \in \mathtt{error}$ then exit("error")
11:         if $a \in G.\mathtt{R}$ then CALCRFS($G, \Gamma, a$)
12:         if $a \in G.\mathtt{W}$ then CALCREVISITS($G, \Gamma, a$)
13:         if $a \in G.\mathtt{B}$ then
14:             $N \leftarrow \mathtt{valw}(w)$ where $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$
15:             $S \leftarrow G.\mathtt{B}_{\mathtt{loc}(a)} \setminus \mathit{dom}(G.\mathtt{sbr})$
16:             if $|S| = N$ then $G.\mathtt{sbr} \leftarrow G.\mathtt{sbr} \cup \{\langle e, e' \rangle \mid e \in S, e' \in S\}$
17:         end if
18:     end while
19: end procedure
```

thread. A thread is considered blocked if it contains a barrier event that is not in the domain of $G.\mathtt{sbr}$. (By construction, such events are $\mathtt{po}$-maximal.) When there are no more events to add, then G is complete, and VISITONE returns.

Depending on the type of an added event a, VISITONE takes appropriate action. Specifically, if a denotes an error (e.g., an assertion violation), it is reported to the user and the verification terminates (Line 10). If a is a read, then we need to find an appropriate $\mathtt{rf}$ edge for it from G. To that end, VISITONE calls CALCRFS (Line 11), which will calculate possible $\mathtt{rf}$ options for a, set one, and push the rest to Γ. If a is a write, it needs to revisit existing reads of the same location in G, because a was not present in the graph when VISITONE was considering possible reads-from options for these reads. To that end, VISITONE calls CALCREVISITS (Line 12), which extends Γ with such alternative explorations.

If a is a barrier-wait event, BAM-specific code takes over. First, BAM finds this barrier's initializing value N (Line 14). Well-formed programs contain a unique initialization of barrier, and so their execution graphs have a unique write event w to each barrier location. Then, BAM collects in the set S all barrier events to the same location as a that are not related by $G.\mathtt{sbr}$ (Line 15. This set contains a as well as all blocked events to the same location. If the number of such events is N, then they form a rendezvous and are thus added to $G.\mathtt{sbr}$, which has the effect of unblocking the waiting threads (Line 16).

As can be seen, BAM can be seamlessly integrated into existing DPOR algorithms. The additional work performed—a linear scan over the graph—does not incur any overhead as it is dominated by the DPOR's consistency checks.

4 Evaluation

4.1 Implementation

We have implemented BAM as an extension of the state-of-the-art stateless model checker GenMC [16]. GenMC operates at the level of LLVM-IR, and can verify C/C++ programs under different (weak) memory models such as RC11 [19] and IMM [23]. We have made our implementation publicly available at https://github.com/MPI-SWS/genmc.

4.2 Experiments

In what follows we compare BAM against the baseline GenMC implementation. We do not directly compare BAM against other tools as 1) most other tools do not offer built-in support for barriers and would thus yield similar results to the baseline GenMC encoding, and 2) GenMC has been extensively compared with other model checking tools in the past (e.g., [16, 18]).

Instead, we set out to show that BAM yields exponential benefits compared to the baseline GenMC implementation for programs with barriers, while at the same time imposes zero overhead.

Experimental Setup We conducted all experiments on a Dell PowerEdge M620 blade system, with two Intel Xeon E5-2667 v2 CPUs (8 cores @ 3.3 GHz) and 256GB of RAM, running a custom Debian-based distribution. We used LLVM 7 for GenMC (v0.5.3). All reported times are in seconds, unless explicitly noted otherwise. We set the timeout limit to 30 minutes.

Benchmarks We evaluate the effectiveness of BAM using a variety of synthetic benchmarks, ranging from simple benchmarks containing a single rendezvous round with no additional computation to benchmarks that involve multiple rendezvous rounds. The results are reported in Tables 1 and 2. As expected, BAM achieves exponential gains over GenMC for all these benchmarks, and scales very well to larger programs. By contrast, the baseline GenMC implementation frequently times out, especially on benchmarks with multiple rendezvous rounds.

Let us first focus on Table 1. Starting with `barrier`, we see that GenMC explores exponentially more executions than BAM, most of which correspond to blocked executions. Indeed, as explained in § 2.2, since the `barrier_wait` operations are considered conflicting, GenMC explores an exponential number of executions for this benchmark. In fact, GenMC explores $(N!)^2$ executions for `barrier(N)`, of which $(N!)^2 - N!$ are blocked.

These numbers might come off as a surprise at first, since it would suffice for GenMC to explore precisely $(N!)$ executions, and no blocked executions. The discrepancy is due to the modeling of `barrier_wait` calls. As described in § 2.2 and Fig. 1, each `barrier_wait` comprises an RMW operation, but also an `assume`$(b == N)$ statement that re-reads the value of the barrier, and ensures that the value read is N. This second read, however, has another $N!$ consistent

Table 1. Synthetic benchmarks containing only barrier operations

	Executions		*Blocked*		*Time*	
	GenMC	BAM	GenMC	BAM	GenMC	BAM
barrier(4)	24	1	552	0	0.02	0.01
barrier(5)	120	1	14 280	0	0.21	0.01
barrier(6)	720	1	517 680	0	7.03	0.01
barrier2(4)	576	1	36 816	0	0.74	0.01
barrier2(5)	14 400	1	5 156 880	0	114.63	0.01
barrier2(6)	🕒	1	🕒	0	🕒	0.01
barrier3(4)	13 824	1	907 152	0	26.07	0.01
barrier3(5)	🕒	1	🕒	0	🕒	0.01
barrier3(6)	🕒	1	🕒	0	🕒	0.01

`barrier(N)`: N threads rendezvous at a barrier.
`barrier2(N)`: N threads rendezvous twice at a barrier.
`barrier3(N)`: N threads rendezvous thrice at a barrier.

`rf` options, which GenMC subsequently has to explore. And at this point, one may wonder: isn't it possible to pack the `assume` statement into the atomic block, and use the value already read for b for the `assume`? Unfortunately, the answer is no. Although we will not go into further details here, we mention in passing that the second read statement is necessary under weak memory models to ensure synchronization between the events before and after the barrier rendezvous.

The differences between GenMC and BAM are magnified once we consider benchmarks with multiple rendezvous rounds. Starting with 4 threads, GenMC explores 5 orders of magnitude more executions than BAM for `barrier2`, and 6 orders of magnitude more for `barrier3`. As the number of threads increases, the performance gap between GenMC and BAM increases even more, despite the fact that most of the executions that GenMC explores are blocked; as it turns out, the cost of enumerating blocked executions quickly becomes exorbitant.

We move on to Table 2, which contains some typical use cases of barriers. The observations here are similar to the ones made for Table 1. The simplest case is that of `barrier-det` that includes a single rendezvous round and only local computations. GenMC scales similarly to the `barrier` benchmark, but takes much more time because of the higher cost per execution. By contrast, the number of threads has a negligible effect to BAM's execution time.

The other three benchmarks use multiple rendezvous rounds to synchronize some computations, while still maintaining a high cost per execution. As expected, this makes GenMC quickly time out. In addition, observe that in the case of `barrier-lock` and `barrier-count` barriers are used to synchronize computations that have additional sources for an exponential number of executions. As the state space of these benchmarks is large to begin with (even disregarding barriers), GenMC quickly exceeds the time limit, while BAM is able to scale

Table 2. Benchmarks with realistic barrier use cases

	Executions		*Blocked*		*Time*	
	GenMC	BAM	GenMC	BAM	GenMC	BAM
barrier-det(3)	6	1	30	0	107.34	17.87
barrier-det(4)	24	1	552	0	424.04	17.87
barrier-det(5)	🕒	1	🕒	0	🕒	17.89
barrier-transc(3)	46 656	1	671 790	0	18 min	0.02
barrier-transc(4)	🕒	1	🕒	0	🕒	0.02
barrier-transc(5)	🕒	1	🕒	0	🕒	0.02
barrier-lock(3)	1296	36	7140	105	0.70	0.03
barrier-lock(4)	331 776	576	4 340 784	3100	417.58	0.42
barrier-lock(5)	🕒	14 400	🕒	143 385	🕒	18.99
barrier-count(3)	55 296	64	715 878	0	88.33	0.04
barrier-count(4)	🕒	4992	🕒	0	🕒	2.57
barrier-count(5)	🕒	2 276 352	🕒	0	🕒	28 min

`barrier-det(N)`: Given a matrix M, calculates the determinant of M^4. The calculation of M^4 is split among N threads, which rendezvous after calculating M^2.

`barrier-transc(N)`: N threads calculate the transitive closure of a matrix via a fixpoint. They rendezvous twice per fixpoint iteration.

`barrier-lock(N)`: N threads test a simple lock implementation: after they rendezvous at a barrier, the threads concurrently attempt to enter their critical section, and mutual exclusion is checked.

`barrier-count(N)`: Contains N threads, with each thread i waiting at barriers b_k, where $i \leq k \leq N$. Counts the number of threads getting through at each round.

to a larger number of threads. We note that the blocked executions that BAM explores in `barrier-lock` are not due to barriers, but rather due to spinloops that can block in the lock implementation under test.

We end this section with a remark on scalability. While it can be argued that scaling up to a large number of threads is unimportant (since e.g., these benchmarks are symmetric), this is not always the case. Often, concurrent implementations tune their behavior depending on the number of threads spawned, and concurrency bugs cannot be manifested with a few threads. Being able to verify programs that employ a large number of threads can therefore be crucial.

5 Summary and Related Work

We presented BAM, a DPOR extension that explores exponentially fewer executions than state-of-the-art stateless model checkers for programs that use synchronization barriers. BAM is based on the key insight that, for most programs, the order in which different threads rendezvous at the barrier is irrelevant, and thus `barrier_wait` statements can be seen as non-conflicting operations by the underlying DPOR algorithms.

After the inception of SMC with tools like VERISOFT [12] and CHESS [21], a growing number of different DPOR techniques has been proposed [11, 1, 8, 9, 10, 5, 6, 15, 2, 3, 14, 4, 16, 18]. Some of these extend DPOR to weak memory models (e.g., [2, 3]), others achieve a coarser equivalence partitioning (e.g., [6, 8, 15]), while others do both (e.g., [4, 16]).

While we are not aware of any other technique that extends DPOR for programs that use barriers, the two works that are closer to ours are CDPOR [6] and LAPOR [15], as they both extend DPOR to scale for particular classes of programs. CDPOR exploits conditional independence between atomic blocks: if the execution of two concurrent atomic blocks leads to the same state under some conditions $\bar{C}$, then the two blocks are deemed independent whenever $\bar{C}$ holds. Thus, if each `barrier_wait` is modeled as an atomic block, CDPOR would be able to explore only 1 execution in programs like BARRIER-N, assuming that the atomic blocks are proven (unconditionally) independent. Proving independence for CDPOR, however, is done using an SMT solver, which might not always be able to prove independence. Alternatively, such conditions would have to be provided manually by the user. LAPOR exploits a similar key idea to BAM and avoids exploring executions that only differ in the order that two critical sections were executed, assuming that these critical sections do not have any conflicting events.

Acknowledgements We thank the anonymous reviewers for their feedback. We are grateful to Xiaowei Ren for his initial implementation of BAM. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York, NY, USA (2014). DOI: 10.1145/2535838.2535845
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: TACAS 2015, LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). DOI: 10.1007/978-3-662-46681-0_28
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: CAV 2016, LNCS, vol. 9780, pp. 134–156. Springer, Heidelberg (2016). DOI: 10.1007/978-3-319-41540-6_8
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. 2(OOPSLA), 135:1–135:29 (2018) DOI: 10.1145/3276505
5. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, pp. 526–543. Springer International Publishing, Cham (2017). DOI: 10.1007/978-3-319-63387-9_26
6. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, pp. 392–410. Springer International Publishing, Cham (2018). DOI: 10.1007/978-3-319-96142-2_24
7. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014) DOI: 10.1145/2627752
8. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: TACAS 2018, LNCS, vol. 10806, pp. 229–248. Springer, Heidelberg (2018). DOI: 10.1007/978-3-319-89963-3_14
9. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. 2(POPL), 31:1–31:30 (2017) DOI: 10.1145/3158119
10. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-Centric Dynamic Partial Order Reduction. Proc. ACM Program. Lang. 3(OOPSLA) (2019) DOI: 10.1145/3360550
11. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York, NY, USA (2005). DOI: 10.1145/1040305.1040315
12. Godefroid, P.: Software Model Checking: The VeriSoft Approach. Form. Meth. Syst. Des. 26(2), 77–101 (2005) DOI: 10.1007/s10703-005-1489-x
13. Godefroid, P., Hanmer, R.S., Jagadeesan, L.J.: Model Checking Without a Model: An Analysis of the Heart-beat Monitor of a Telephone Switch Using VeriSoft. In: ISSTA 1998, pp. 124–133. ACM, Clearwater Beach, Florida, USA (1998). DOI: 10.1145/271771.271800
14. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2(POPL), 17:1–17:32 (2017) DOI: 10.1145/3158105
15. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. 3(OOPSLA) (2019) DOI: 10.1145/3360599

16. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019, ACM, New York, NY, USA (2019). DOI: 10.1145/3314221.3314609
17. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the Linux kernel's read–copy update (RCU). Int. J. Soft. Tool. Tech. Transf. (2019) DOI: 10.1007/s10009-019-00514-6
18. Kokologiannakis, M., Vafeiadis, V.: HMC: Model checking for hardware memory models. In: ASPLOS 2020, ASPLOS '20, pp. 1157–1171. ACM, Lausanne, Switzerland (2020). DOI: 10.1145/3373376.3378480
19. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017). DOI: 10.1145/3062341.3062352
20. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979) DOI: 10.1109/TC.1979.1675439
21. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280. USENIX Association (2008). URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf
22. Oberhauser, J., Chehab, R., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In: ASPLOS 2021, (2021)
. 23. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. 3(POPL), 69:1–69:31 (2019) DOI: 10.1145/3290382
24. *pthread.h man page*(2017). URL: https://man7.org/linux/man-pages/man0/pthread.h.0p.html. (Accessed: 2021–3–19)
. 25. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI 2015, pp. 250–259. ACM, New York, NY, USA (2015). DOI: 10.1145/2737924.2737956