Stateless Model Checking Concurrent Programs with Maximal Causality Reduction

Jeff Huang
Texas A&M University, USA
jeff@cse.tamu.edu

Abstract

We present maximal causality reduction (MCR), a new technique for stateless model checking. MCR systematically explores the state-space of concurrent programs with a *provably minimal* number of executions. Each execution corresponds to a distinct *maximal causal model* extracted from a given execution trace, which captures the largest possible set of causally equivalent executions. Moreover, MCR is embarrassingly parallel by shifting the runtime exploration cost to offline analysis. We have designed and implemented MCR using a constraint-based approach and compared with iterative context bounding (ICB) and dynamic partial order reduction (DPOR) on both benchmarks and real-world programs. MCR reduces the number of executions explored by ICB and ICB+DPOR by orders of magnitude, and significantly improves the scalability, efficiency, and effectiveness of the state-of-the-art for both state-space exploration and bug finding. In our experiments, MCR also revealed several new data races and null pointer dereference errors in frequently studied real-world programs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Software/Program Verification—Model checking

General Terms Algorithms, Design, Theory

Keywords Maximal Causality Reduction, Model Checking

1. Introduction

A fundamental challenge in concurrent program verification is thread interleaving explosion: the number of possible interleavings grows exponentially with the number of threads and the length of program execution. A general idea to alleviate this problem is to ignore redundant interleavings that produce equivalent program states. In stateless model checking, techniques systematically explore the state-space of a given program by driving its executions via a special scheduler, which makes scheduling decisions to explore only non-redundant interleavings. Several stateless model checkers have been developed, such as VeriSoft [12] and CHESS [25], and proven valuable in practice for finding errors.

A crucial task in stateless model checking is how to identify redundant interleavings. The classical approach is *partial order reduction* (POR) [5, 10, 11], based on the observation that an interleaving is redundant if it can be obtained from another by swapping adjacent non-conflicting events from different threads. All interleavings can be categorized into a number of distinct *equivalent classes* called Mazurkiewicz traces [1]. POR explores one interleaving in each Mazurkiewicz trace and is proven sufficient for checking most interesting safety properties, such as absence of assertion violations and race conditions.

A major limiting factor of POR is that the Mazurkiewicz trace is characterized based on *strict* event dependencies with respect to the *happens-before* relation [20]. For shared memory systems with locks and shared variables, the happens-before relation enforces dependence on all the lock acquire/release and conflicting read/write events. This misses many opportunities for identifying redundant interleavings. Consider three threads, p, q and r, performing read and write accesses to a shared variable x:

p: *write* x; q: *write* x; r: *read* x;

According to happens-before, all the three accesses are dependent. Hence the six interleavings $\{p.q.r,\ p.r.q,\ q.p.r,\ q.r.p,\ r.p.q,\ r.q.p\}$ belong to six different Mazurkiewicz traces and POR must explore all of them. From the surface, this is needed as none of them are redundant. However, if we take a closer look, only half of them are necessary to explore. In fact, $p.q.r$ is equivalent to $q.r.p$, $p.r.q$ to $q.p.r$, and $r.q.p$ to $r.q.p$. The reason is that r is the only thread which reads x, of which the returned value may affect the to-be-verified property. If the two writes by p and q do not result in a *different value* read by r, then their order is redundant. Even better, if both of the two writes produce the same value, then $p.q.r$ is also equivalent to $p.r.q$, resulting in only two interleavings ($p.q.r$ and $r.p.q$) to explore.

Our first contribution in this work is to characterize the Mazurkiewicz trace based on a new criterion: *maximal causality*. Conceptually, maximal causality characterizes the largest possible set of equivalent interleavings in each Mazurkiewicz trace by taking the value of reads and writes into consideration. When applied in stateless model checking, it minimizes the number of interleavings that must be explored. In other words, it enables exploring the entire state-space of a concurrent program with respect to a given input with a minimal number of executions.

The theoretical model of maximal causality was proposed in [16, 30], establishing the foundation that for any interleaving s, there exists a sound and maximal set of interleavings, MaxCausal(s), which comprises precisely the interleavings that can be generated by all programs that can generate s. The model is well-suited for stateless model checking as it is based on purely dynamic information (*a.k.a. trace*) emitted in the execution, such as what data and value an event reads from or writes to by which thread, with no assumption about the program code or the data/control flow between events.

PLDI'15 , June 13–17, 2015, Portland, OR, USA

As illustrated in Figure 1, our approach integrates stateless model checking with maximal causality in a closed loop. Whenever an interleaving is explored, the maximal causality engine takes the corresponding trace as input and generates a set of new interleavings (if there exists any), called *seed interleavings*, and feeds them back to the model checker to continue exploring. Each seed interleaving is a feasible interleaving in MaxCausal(s). However, executing the program following a seed interleaving will reach a new program state (because a read is forced to see a different value), and extending it with any new event will produce a new interleaving, which is *maximal-causally distinct* from any other explored interleavings. In other words, for each such new interleaving s', MaxCausal(s') accounts for a unique subspace of all interleavings. The whole process is systematic: it starts with an empty seed interleaving and stops when all seed interleavings are explored and no new can be generated, meaning that the entire state-space has been covered.

We call the above process *Maximal Causality Reduction* (MCR). There are two main technical challenges in achieving MCR for stateless model checking:

1. How to realize the maximal causal model (MCM)? Simply enumerating all possible interleavings [30] is intractable and has never been implemented in practice.
2. How to generate all the seed interleavings and ensure that none of them are redundant and none is missed?

We present an algorithm that encodes MCM as a series of quantifier-free first-order logic formulas, Φ, whose satisfiability can be decided by an off-the-shelf SMT solver. The solutions of Φ conjuncted with a property ϕ represent the interleavings in MaxCausal(s) that satisfy ϕ. To generate seed interleavings, we leverage Φ and conjunct it with additional *state-change* constraints (*i.e.*, enforce a read to return a different value), such that the solution to each new formula represents a seed interleaving. Although our algorithm requires solving constraints which may be expensive, this task can be done completely offline. As SMT solvers such as Z3 [7] and Yices [8] are becoming increasingly powerful, this constraint-based approach scales well in practice.

Moreover, separating the generation of seed interleaving (which is offline) from online exploration makes MCR easily parallelizable, with at least two levels of parallelism. First, the online exploration with different seed interleavings is embarrassingly parallel. Second, the tasks of generating different seed interleavings are independent of each other and thus can be performed in parallel. Comparatively, conventional model checking with POR is hard to parallelize, because the exploration of new interleavings is online and is dependent on previously explored interleavings.

Orthogonal to POR, another effective approach for stateless model checking is iterative context bounding (ICB) [23]. ICB bounds thread preemptions in the explored interleavings, which reduces the explored interleaving space to be polynomial in the execution length. ICB can be combined with dynamic partial order reduction (DPOR) [10] to further improve the effectiveness [6, 22, 31]. Differently, ICB does not reduce redundant interleavings and does not provide full state-space coverage. ICB is neither amenable to massive parallelization, because in ICB each iteration relies on the completion of previous iterations.

We have implemented MCR in a stateless model checker for Java and compared it with both ICB and ICB+DPOR on popular benchmarks and two real-world applications, `Jigsaw` and `Weblech`. Our evaluation results show that MCR reduces the number of explored interleavings by ICB and ICB+DPOR by orders of magnitude, and it improves the scalability, efficiency, and effectiveness over them significantly for both state-space exploration and bug finding in terms of data races and null pointer dereferences (NPE). We have also designed a parallel MCR algorithm and compared it with the other techniques on a 32-core machine. Our parallel algorithm achieves a much higher scalability and detects many more data races and NPEs. In our experiments with `Jigsaw` and `Weblech`, MCR also found three new data races and two new NPEs.

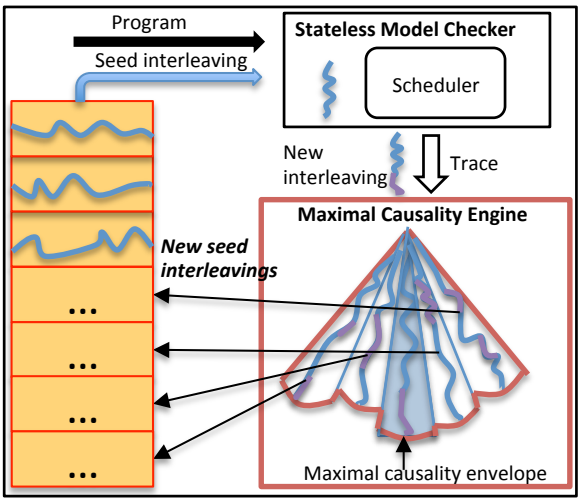


Figure 1: Technical overview of maximal causality reduction

We highlight our contributions as follows:

- We present maximal causality reduction which minimizes the number of explored executions for stateless model checking concurrent programs.
- We present a constraint-based approach to realize maximal causality reduction. This approach separates interleaving generation from exploration, shifting the runtime exploration to embarrassingly parallel offline analysis.
- We evaluate maximal causality reduction using both popular benchmarks and real programs and show that it significantly advances the state-of-the-art techniques.

2. Overview

We first provide an artificial example to illustrate the advantage of MCR, and then outline our approach.

2.1 Motivating Example

The example program in Figure 2 starts three concurrent threads `T1`, `T2` and `T3`, each loops twice and accesses two shared variables `x` and `y` and a lock `l`. There is an error at line 13 that will be triggered in `T3` if the two conditions ① `x > 1` and ② `y == 3` are both satisfied. The error is hard to find, however, because it is hidden deeply with complex thread interactions. For condition ① to be true, line 2 must be executed between line 7 and line 8; for ② to be true, line 9 must be executed after line 14. In addition, lines 11 and 12 must be executed after lines 10 and 9, respectively, and before the values of x and y being changed by any other writes. For the error to occur, each thread has to loop twice following the interleaving `T2-T2-T2-T1-...-T1-T3-T3` shown at the bottom of Figure 2. However, consider all the possible interleavings in this program, the total number is more than 10 million (by approximation [21])! This poses significant challenges for existing techniques to find the error.

We ran this program with a stateless model checker developed in this work with various algorithms implemented. The basic mode (depth-first-search until all interleavings are covered) ex-

initially **x**=**y**=0

Event at line *i* in loop *j*

i=1, 2, ... ,14 j=1, 2

L_i^j /U_i^j : *lock/unlock*

R_i^j/ W_i^j : *read/write*

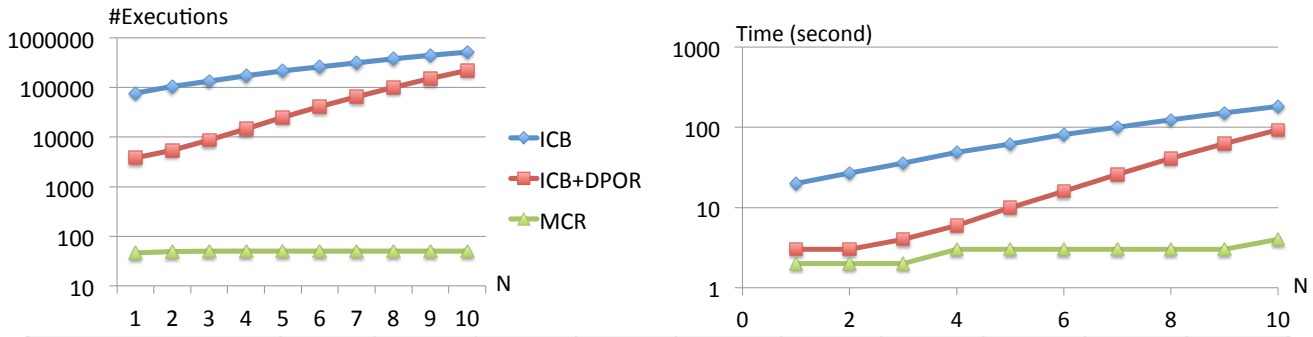
Error-triggering interleaving:

T1	**T2**	**T3**
loop twice:	*loop twice:*	*loop twice:*
1: lock(l)	5: lock(l)	11: if(**x**>1){
2: **x**=1	6: **x**=0	12: if(**y**==3){
3: **y**=1 ← *loop N times*	7: unlock(l);	13: ***Error***
4: unlock(l);	8: if(**x**>0){	else
	9: **y**++	14: **y**=2
	10: **x**=2	}
	}	}

$T2$ - $T2$ - $T2$ - $T1$ - $T1$ - $T1$ - $T1$ - $T2$ - $T2$ -$T2$ -$T2$ - $T3$ - $T3$ - $T3$ - $T2$ - $T2$ - $T2$ -$T2$ - $T1$ - $T1$ - $T2$ -$T2$ - $T2$ - $T2$ - $T3$ - $T3$

L_5^1 - W_6^1 - U_7^1 - L_1^1 - W_2^1 - W_3^1 - U_4^1 - R_8^1 - R_9^1 - W_9^1 - W_{10}^1 - R_{11}^1 - R_{12}^1 - W_{14}^1 - L_5^2 - W_6^2 - U_7^2 - L_1^2 - W_2^2 - R_8^2 - R_9^2 - W_9^2 - W_{10}^2 - R_{11}^2 - R_{12}^2

Figure 2: Example. The error-trigger interleaving and the corresponding trace are shown at the bottom. Each event is annotated with its line number and loop iteration number. For example, R_8^2 refers to the read event at line 8 in the second loop iteration.



Algorithms		N=1	2	3	4	5	6	7	8	9	10
Number of Executions	ICB	77322	104115	135721	172488	214764	262897	317235	378126	445918	520959
	ICB +DPOR	3782	5416	8702	14714	24898	41126	65750	101656	152318	221852
	MCR	46	49	50	50	50	50	50	50	50	50
Time (Second)	ICB	20	27	36	49	62	81	100	124	151	183
	ICB +DPOR	3	3	4	6	10	16	26	41	63	93
	MCR	2	2	2	3	3	3	3	3	3	4

Figure 3: Comparison between different algorithms for finding the error in Figure 2 with N from 1 to 10.

plored 3,293,931 interleavings before it timed out in one hour. The ICB mode (which implements the same ICB algorithm as that in CHESS [25], which is usually the most efficient method for finding concurrency bugs) took 77,322 executions until it hit the error after exploring for 20 seconds. ICB can be combined with DPOR [10] to further reduce the explored executions. We ran the ICB+DPOR mode and it still took 3782 executions before it found the error in 3 seconds. Comparatively, our approach with MCR took only 46 executions and found the error in 2 seconds, reducing the number of executions explored by ICB+DPOR by two orders of magnitude, and ICB by three orders of magnitude, because most of the executions are causally equivalent.

To make the example more interesting, we let line 3 (which sets y to 1 repeatedly) loop for N times and compare the performance of different algorithms as N increases. Since every execution of line 3 corresponds to a new write event on y, the interleaving space of the program increases exponentially with N. For each larger N, both ICB and DPOR have to explore significantly more interleavings before they can find the error. However, MCR does not, because the repeated write events at line 3 only create redundant interleavings which are captured by maximal causality. Figure 3 shows the comparison results with N from 1 to 10. Notice that the vertical axis is on logarithmic scale. For both ICB and ICB+DPOR, the number of executions and the total time for them to find the error both increase drastically as N increases. ICB+DPOR even increases faster than ICB alone, because DPOR is not able to reduce the redundant interleavings caused by the writes at line 3. However, MCR is almost insensitive to N. It becomes stable at 50 executions when N is larger than 3, and the total time only increases negligibly.

2.2 Maximal Causality Reduction

Conceptually, MCR differs from existing techniques by covering a *set* of interleavings for each *single* execution. Underpinned by the maximal causal model (MCM), MCR is able to analyze an

exponential and provably *maximal* number of interleavings derived from each single execution trace.

Algorithm 1 **MaxCausalExplore**(S)

```
1: Input: S - a set of seed interleavings, initially {s0};
2:        s0 - an empty interleaving.
3: for s ∈ S do
4:    τ ← Execute(s);
5:    Φ ← ConstructMaxCausalModel(τ);
6:    PropertyCheck(Φ);
7:    S' ← GenerateSeedInterleavings(τ, Φ);
8:    MaxCausalExplore(S');
```

Algorithm 1 outlines our basic algorithm. It works in an iterative manner. Each iteration explores one interleaving online, and checks a maximally causal set of derived interleavings offline. In addition, each iteration may generate more *seed interleavings*, which will be used to drive other iterations. More specifically, each iteration has three steps:

Online exploring and tracing for one execution This step executes the program, first following a seed interleaving (initially empty) and continuing (with arbitrary interleaving) until the end of execution. In addition to executing the program, this step also collects a trace that includes the necessary information for constructing the MCM. Note that the trace collected here is not required to hit the error. We will present the details of MCM in the next section.

Offline checking for maximal causal interleavings This step constructs a MCM and checks properties offline. Each MCM contains a set of interleavings, MaxCausal(s), which is a unique and maximal set of feasible interleavings inferred from the trace corresponding to the interleaving s. To check properties, we encode MaxCausal(s) as a formula Φ over a set of order variables, such that any solution of the formula corresponds to a legal interleaving represented by a topological sort of the order variables. By encoding the properties as additional constraints and solving their conjunction with Φ, we can determine if a property holds or not for all the interleavings in MaxCausal(s).

Generating seed interleavings This step generates seed interleavings which will be used in the first step to produce new interleavings. For each new interleaving s, MaxCausal(s) covers a unique subspace of interleavings. This step is critical because to minimize the number of executions, we must ensure no two subspaces overlap and all subspaces together cover the entire interleaving space.

Our basic idea for generating the seed interleavings is to enforce reads in the trace to return *different values* allowed by the MCM. The rationale is that every seed interleaving must drive the program to reach a new state. For example, suppose in our example we get a trace in which R_9^2 reads value 1 on `x` and W_{14}^1 writes value 2. We will try to force R_9^2 to read a different value, 2 in this case, written by W_{14}^1. If there exists an interleaving in MaxCausal(s) that can satisfy this condition, then the interleaving will be generated as a seed interleaving. In our example, we will generate a seed interleaving in which W_{14}^1 happens before R_9^2 and R_9^2 reads 2 on y. Once this interleaving is enforced at runtime, the program will reach a new state and produce a new trace by continuing execution.

Our approach will terminate when all seed interleavings are explored and no new seed interleaving can be generated, indicating that the entire state-space has been covered.

Example Figure 4 illustrates our seed interleaving generation algorithm for our example in Figure 2. `s0` is the seed interleaving of the initial trace. In the first iteration, we generate four seed

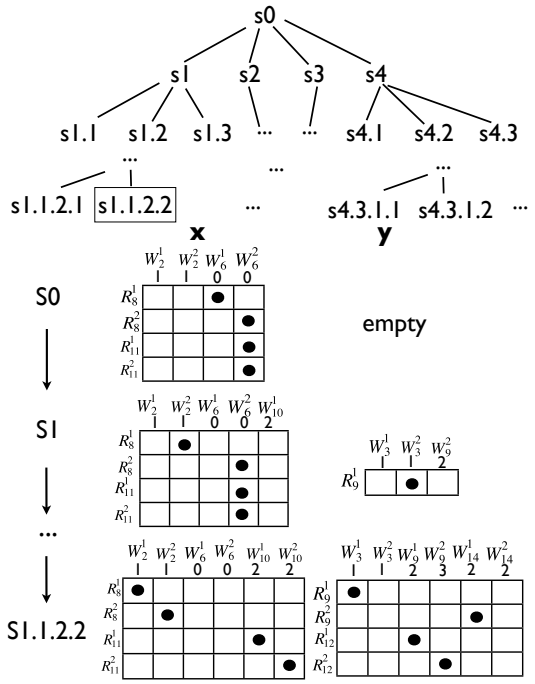


Figure 4: A hierarchical view of seed interleaving generation

interleavings {`s1`, `s2`, `s3`, `s4`}, which enforce the four reads R_8^1, R_8^2 R_{11}^1, R_{11}^2, respectively, to read value 1 (instead of 0 in the initial trace). In the second iteration, we continue to work on the traces corresponding to s_i (i=1,2,3,4), and generate `s1.1`, `s1.2`, ..., `s2.1`, `s2.2`, etc. All interleavings form a hierarchy, with each child interleaving enforcing a different read value. Again, each new interleaving may produce a trace containing new read events and/or write values, which can generate new children interleavings. For example, our approach will eventually generate the interleaving `s1.1.2.2`, which enforces R_{12}^2 to read 3 and triggers the error at line 13.

3. Approach

In this section, we first present MCM and how we encode it. We then present our seed interleaving generation algorithm in detail. Our approach shifts the reasoning of redundant interleavings to offline constraint solving. The resulting process can be parallelized easily, which in theory scales MCR to arbitrary large programs if there are sufficient cores. We present our parallel algorithm in Section 3.5.

3.1 Maximal Causal Model

Multithreaded programs $\mathscr{P}$ can be abstracted as the prefix-closed sets of finite traces that they can produce when completely or partially executed, called $\mathscr{P}$*-feasible* traces. A *trace* is abstracted as a sequence of events. *Events* are operations performed by threads on concurrent objects. In the original MCM [30], only reads/writes and mutex events are considered, as all the other events can be built as abstractions upon them. We consider the following common event types in this work:

- $begin(t)$/$end(t)$: the first/last event of thread t;
- $read(t,x,v)$/$write(t,x,v)$: read/write x with value v;
- $lock(t,l)$/$unlock(t,l)$: acquire/release a lock l;
- $fork(t,t')$: fork a new thread t';
- $join(t,t')$: block until thread t' terminates;

The sets of $\mathscr{P}$-feasible traces must obey two basic consistency axioms: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathscr{P}$-feasible trace are also $\mathscr{P}$-feasible. The latter says that each thread has a deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. These two axioms allow us to associate a causal model $\mathit{feasible}(\tau)$ to any consistent trace τ, which comprises precisely the traces that can be generated by any program that can generate τ.

It is shown in [16, 30] that $\mathit{feasible}(\tau)$ is both *sound* and *maximal*: any program which can generate τ can also generate all traces in $\mathit{feasible}(\tau)$, and for any trace τ' not in $\mathit{feasible}(\tau)$ there exists a program generating τ which cannot generate τ'. Comparatively, conventional happens-before models consisting of all the legal interleavings of τ and their prefixes are sound, *but not maximal*, indicating that *POR is not optimal* for reducing redundant interleavings. We refer the readers to [16, 30] for the proofs of soundness and maximality,

Due to the complexity of MCM, however, it is hard to implement in practice. In our prior work [16], we proposed to use constraints and realized a specialized MCM for race detection. Building upon [16], in this work we first realize the general MCM by encoding it as a formula Φ with first-order logical constraints. We next describe our constraint encoding in detail.

3.2 Constraint Encoding of Maximal Causal Model

From a high level view, Φ contains only variables of the form O_e corresponding to events e, which denote the order of the events in a trace in $\mathit{feasible}(\tau)$. Φ is constructed by a conjunction of two subformulas: $\Phi \equiv \Phi_{sync} \wedge \Phi_{rw}$, where Φ_{sync} denotes the inter-thread order constraints determined by synchronization events, and Φ_{rw} the data-validity constraints over read and write events. Φ_{sync} can be further decomposed as conjunction of the must-happen-before and the lock-mutual-exclusion constraints.

Must-happen-before constraints (Φ_{mhb}) The must-happen-before (MHB) constraints reflect a *subset* of the classical happens-before relation, ensuring a minimal set of ordering relations that events in any feasible interleaving must obey. Specifically, MHB requires that (1) the total orders of the events in each thread are always the same; (2) a *begin* event can happen only after the thread is forked by another thread; (3) a *join* event can happen only after the *end* event of the joined thread. Clearly MHB yields a partial order over the events of τ which must be respected by any trace in $\mathit{feasible}(\tau)$. We denote MHB by $\prec$, which will be used later. We can specify $\prec$ easily as constraints Φ_{mhb} over the O variables: we start with $\Phi_{mhb} \equiv \mathit{true}$ and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2, or when e_1 is an event of the form $\mathit{fork}(t, t')$ and e_2 of the form $\mathit{begin}(t')$, etc.

Lock-mutual-exclusion constraints (Φ_{lock}) The locking semantics requires that any two code regions protected by the same lock are mutually exclusive, *i.e.*, they should not interleave. Φ_{lock} captures the ordering constraints over *lock* and *unlock* events. For each lock l, we extract the set S_l of all the corresponding pairs, (e_a, e_b), of *lock*/*unlock* events on l, following the program order locking semantics: the *unlock* is paired with the most recent *lock* on the same lock by the same thread. Then we conjunct $\Phi_{lock} \equiv \mathit{true}$ with the formula

$$\bigwedge_{(e_a,e_b),(e_c,e_d)\in S_l} (O_{e_b} < O_{e_c} \vee O_{e_d} < O_{e_a})$$

Data-validity constraints (Φ_{rw}) The data-validity constraints ensure that every event in the *considered* trace is feasible. Note that in constructing MCM for an input trace τ, the considered trace does not necessarily contain all the events in τ but may contain a subset of them, so that all the incomplete traces corresponding to partial executions of the program are considered as well. For an event to be feasible, all the events that must-happen-before it should also be feasible. Moreover, every read event that must-happen-before it should read the *same value* as that in the input trace; otherwise the event might become infeasible due to a different value read by an event that it depends on. Each read, however, may read a value written by any write, as long as all the other constraints are satisfied.

Let $\prec_e$ denote the set of events that must-happen-before an event e. Consider a read event r in $\prec_e$, say $\mathit{read}(t, x, v)$, we let W^x be the set of $\mathit{write}(\_, x, \_)$ events in τ (here '_' denotes any value), and W^x_v the set of $\mathit{write}(\_, x, v)$ events in τ. The data-validity constraint of e, denoted by $\Phi_{rw}(e)$, is written as:

$$\Phi_{rw}(e) \equiv \bigwedge_{r\in\prec_e} \Phi_{value}(r, value(r))$$

where $\Phi_{value}(r, v)$ is defined as follows:

$$\Phi_{value}(r, v) \equiv \bigvee_{w\in W^x_v} (\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{w\neq w'\in W^x} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The constraint $\Phi_{rw}(e)$ requires that every read that must-happen-before e should read the same value as that in the input trace. The constraint $\Phi_{value}(r, v)$ enforces the read event $r = \mathit{read}(t, x, v)$ to read the value v on x (written by any *write* event $w = \mathit{write}(\_, x, v)$ in W^r_v), subject to the condition that the order of w is smaller than that of r and there is no interfering $\mathit{write}(\_, x, \_)$ in between. In addition, w itself must be feasible, which is ensured by $\Phi_{rw}(w)$.

Since MCM models all the incomplete traces as well, the data-validity constraint Φ_{rw} is thus satisfiable if any event in the input trace τ is feasible, written as a disjunction of the feasibility constraints of all events in τ:

$$\Phi_{rw} \equiv \bigvee_{e\in\tau} \Phi_{rw}(e)$$

It is worth noting that the formula Φ constructed in this section encodes all the feasible interleavings, *i.e.*, $\mathit{feasible}(\tau)$, that can be inferred from the input trace τ. Each solution of the order variables to Φ corresponds to an interleaving in $\mathit{feasible}(\tau)$. The size of Φ is cubic in number of reads and writes in τ, and the size of $\mathit{feasible}(\tau)$ may be huge as the number of unique solutions to Φ can be exponential. In practice, however, we do not need to directly solve Φ to produce all the interleavings in $\mathit{feasible}(\tau)$. For example, when used for checking properties, it often suffices to find one interleaving that satisfies the property. We next show how to check assertion violation and data race properties using Φ.

3.3 Property Checking with Maximal Causal Model

Instead of checking properties for one interleaving at a time, which is performed at runtime by existing stateless model checkers, MCR enables checking properties against a maximal causal set of interleavings offline. Given a property ϕ defined over the order variables and the values of reads, we use a constraint solver to solve $\phi \wedge \Phi$. If the solver finds a solution, it means that there exists an interleaving satisfying the property and the corresponding interleaving will be reported, which can be extracted from the solution by ordering the events according to the value of the order variables. At a low level, the solving of $\phi \wedge \Phi$ can be significantly simplified by tailoring Φ to only the relevant events considered in ϕ.

Checking assertion violations Consider an assertion violation property $\phi_{assert}(R)$, which is defined over the program states concerning the values of a set of read events R. Firstly, since the property is only affected by the events in R, we can reduce the data validity constraint Φ_{rw} to consider only those in R, that is, $\Phi_{rw}(e)$

for all $e \in R$. Secondly, for any read in R, it may read the value written by any write on the same variable, subject to the condition that the corresponding interleaving is feasible. Let $v(r)$ denote the value that can be returned by a read event $r = read(t, x, \_)$, and V^x the set of values written by W^x, the set of writes to x. Recall $\Phi_{value}(r, v)$ denotes the constraint for r to read a value v. $v(r)$ is written as:

$$v(r) \equiv \bigvee_{v \in V^x} v \wedge \Phi_{value}(r, v)$$

With the above reduction, $\Phi \wedge \phi_{assert}(R)$ is simplified to:

$$\Phi_{sync} \wedge (\bigwedge_{e \in R} \Phi_{rw}(e) \wedge v(e)) \wedge \phi_{assert}(R)$$

As an example, suppose the property to check is $value(e) = NULL$ for a read event e (such as checking null pointer dereferences), the formula solved by the constraint solver is:

$$\Phi_{sync} \wedge \Phi_{rw}(e) \wedge (value(e) = NULL)$$

Checking data races Data races are a particularly problematic type of errors that have caused some of the worst concurrency problems in multithreaded systems today. A data race occurs when there are unordered conflicting accesses in the program without proper synchronization. Consider two read/write events, e_a and e_b, to a shared variable from different threads, and at least one of them is a write, the data race property $\phi_{race}(e_a, e_b)$ can be defined easily over the order variables corresponding to the events e_a and e_b:

$$\phi_{race}(e_a, e_b) \equiv (O_{e_a} = O_{e_b})$$

Similar to checking assertion violations, checking data races against MCM only needs to consider the data-validity constraints of $\Phi_{rw}(e_a)$ and $\Phi_{rw}(e_b)$ for the property $\phi_{race}(e_a, e_b)$ for each pair of conflicting accesses by different threads. Therefore, the formula $\phi_{race}(e_a, e_b) \wedge \Phi$ is reduced to:

$$\Phi_{sync} \wedge (O_{e_a} = O_{e_b}) \wedge \Phi_{rw}(e_a) \wedge \Phi_{rw}(e_b)$$

3.4 Seed Interleaving Generation

Our algorithm works by pivoting around the value of reads in the trace. We ensure that each generated seed interleaving has at least one new event: a read event that reads a new value (*i.e.*, a different value from that in other interleavings). All such new events are considered and their corresponding interleavings are generated as long as the interleaving is feasible, *i.e.*, satisfying the MCM formula Φ. In this way, we guarantee that no two seed interleavings are redundant. In addition, because all possible legal combinations of read values are considered, we guarantee that no seed interleaving is missed and the entire state-space will be covered eventually.

Algorithm 2 **GenerateSeedInterleavings(τ, Φ)**

```
1: Input: τ - the input trace;
2:        Φ - the maximal causal formula for τ.
3: Return: S - a set of seed interleavings.
4: for r = read(t, x, v) ∈ τ do
5:   for w = write(_, x, v') ∈ τ ∧ v' ≠ v do
6:     Φ_seed(r, w) ← ConstructSeedConstraint(r,w,Φ);
7:     s ← SolveSeedConstraint(Φ_seed(r, w));
8:     if s ≠ null then
9:       add s to S;
```

Algorithm 2 shows our seed generation algorithm. Given the input trace τ, our algorithm enumerates each read event in τ on the set of all values by the writes on the same variable. For each value that is different from what it reads in τ, we construct a formula Φ_{seed} that constrains the read to read the value. Specifically, consider a read $r = read(t, x, v)$ and a write $w = write(\_, x, v')$ such that $v \neq v'$, and recall $\Phi_{value}(r, v)$ denotes the constraint for r to read a value v, we construct the following formula:

$$\Phi_{seed}(r, w) \equiv \Phi_{sync} \wedge \Phi_{rw}(r) \wedge \Phi_{rw}(w) \wedge \Phi_{value}(r, v')$$

For each $\Phi_{seed}(r, w)$, we invoke a constraint solver. If the solver returns a solution, the solution represents a new interleaving which is feasible and in which the read will read that new value. Note that each read only concerns about the distinct values but not distinct writes. If there are multiple writes writing the same value, it suffices to generate only one new interleaving for all of them. This explains why MCR is insensitive to N in our example in Figure 2.

Termination Our algorithm terminates when no new seed interleaving can be generated. Unlike ICB or DPOR, our algorithm does not need a search stack because each seed interleaving (*i.e.*, the prefix of each new explored interleaving) is already keeping track of the search progress. To avoid generating duplicated seed interleavings from different τ, we ensure that the prefix of each new explored interleaving is always preserved. Figure 4 shows an intuitive view of this process. Each table illustrates the read-write mappings corresponding to a seed interleaving. A table grows as new reads and writes are discovered but never shrinks.

An important property of our algorithm is that it will cover the entire state-space with the following theorem:

THEOREM 1. *Suppose the program is terminating, our approach presented in this section will eventually cover the whole state-space of the program that can be driven by all the possible interleavings with respect to a given input.*

Proof sketch: By contradiction. Suppose there exists an interleaving s not covered, then it must be the case that s contains a new event. There could be two possibilities only: (1) the new event is a previously observed event, but reads a new value; (2) it is a previously unseen event. Case (1) is impossible, because our algorithm generates a new seed interleaving for every such read. For (2), it must be the case that the event depends on a branch, the condition of which cannot be satisfied by any of the states driven by our generated seed interleavings. However, this also means that the branch condition depends on at least one previous read event reading a new value, contradicting to the fact that our algorithm generates a seed interleaving for each read with a different value.

Algorithm 3 **Parallel-MaxCausalExplore(s)**

```
1: Input: s - a seed interleaving, initially empty.
2: τ ← Execute(s);
3: Φ ← ConstructMaxCausalModel(τ);
4: async PropertyCheck(Φ);
5: parfor r = read(t, x, v) ∈ τ do
6:   parfor w = write(_, x, v') ∈ τ ∧ v' ≠ v do
7:     Φ_seed(r, w) ← ConstructSeedConstraint(r,w,Φ);
8:     s' ← SolveSeedConstraint(Φ_seed(r, w));
9:     if s' ≠ null then
10:      Parallel-MaxCausalExplore(s');
```

3.5 Parallel Maximal Causality Reduction

Unlike ICB and DPOR which are completely online and are hard to parallelize, MCR opens the door for massive parallelism. By separating offline interleaving generation from online exploration, parallelizing MCR is mostly straightforward, as the only dependence between different iterations is the seed interleaving. Inside each iteration, multiple seed interleavings can be generated in parallel. In addition, the online exploration for each seed interleaving is independent.

Algorithm 3 shows our parallel MCR algorithm. To maximize the degree of parallelism, the input to the procedure *Parallel-MaxCausalExplore* is a single seed interleaving (initially empty). At any time of our algorithm's execution, there can be many parallelly executing *Parallel-MaxCausalExplore* procedures each working on a different seed interleaving. Inside the procedure, the MCM formula Φ corresponding to the input trace (produced by executing the program following the seed interleaving) is first constructed. Then property checking and seed interleaving generation are performed in parallel based on Φ. At line 4, `async` means creating a new concurrently executing task. Lines 5-13 describe the parallel seed interleaving generation. `parfor` means executing the `for` loop in parallel, for each pair (r, w) of a read event r and a matching write event w (which writes a different value). In each parallel subtask corresponding to (r, w), the seed constraint $\Phi_{seed}(r, w)$ is constructed and solved with an SMT solver. If the constraint is satisfiable, a new seed interleaving will be returned and started by a new instance of *Parallel-MaxCausalExplore*.

4. Implementation

To evaluate our algorithms, we have developed a stateless model checker called ASER using ASM [2] and Z3 [7]. In ASER, we implemented both the basic MCR algorithm and the parallel algorithm. To compare with the state-of-the-art, we have also implemented the ICB algorithm in the original CHESS model checker [23, 25] and its integration with DPOR [10]. In addition, for these algorithms we have implemented the detection of two safety violation properties: null pointer dereference (NPE) and data race.

Three main components ASER consists of a runtime tracer, an offline constraint analyzer, and a special scheduler. The runtime tracer captures critical events in the execution including all shared data accesses and thread synchronizations. Our current implementation dynamically instruments Java programs using bytecode rewriting. Nevertheless, our algorithm is general to different languages. The offline constraint analyzer formulates the MCM constraints from the events and generates seed interleavings by solving the constraints using Z3. Since all MCM constraints are simple ordering comparisons over integer variables, we use the Integer Difference Logic (IDL) in Z3 to solve them efficiently. The special scheduler controls the thread execution to follow the seed interleavings by intercepting the critical events with application-level conditional variables and semaphores.

ICB/DPOR Our implementation of ICB follows the original algorithm [23]. The only difference is that we preempt not only prior to thread synchronizations but also before every shared data access, because we want to evaluate on programs with data races as well. For ICB+DPOR, naively combining ICB with the original DPOR algorithm [10] is unsound. We follow the bounded partial order reduction [6] to implement it. Both of these two algorithms are implemented as plugins to the special scheduler as they are purely dynamic. For ICB, the scheduler checks before every critical event the number of thread preemptions in the current schedule. All schedules with preemption number less or equal to a pre-defined bound, N, will be explored. ICB starts with zero preemption. After all such interleavings are explored, it increases the preemption number by one and starts a new iteration. This process is repeated until reaching N. For ICB+DPOR, we maintain vector clocks to track happens-before following the optimization in [10]. When a dependence is detected, we create new schedules to explore by adding backtracking points following [6].

Data race and NPE detectors For ICB and DPOR, we implement dynamic NPE and data race detection since they both perform property checking online. For NPE, the scheduler simply tracks `NullPointerException` at runtime. For race detection, we implement the happens-before (HB) based algorithm using vector clocks. Note that classical happens-before tracks HB edges on synchronization events only and is only precise up to the first race. We also track HB on shared data reads and writes to ensure all detected races are real. For MCR, we implement the property checking algorithms for NPE and data race in the constraint analyzer according to Section 3.3. It is worth noting that neither any NPE nor data race has to occur in the explored executions before it can be detected by MCR. The offline property checking on the MCM formula enables precisely predicting these property violations in all the maximal causal set of interleavings. Moreover, once the seed interleaving corresponding to a property violation is generated, it will drive the program to deterministically expose the violation.

5. Evaluation

We have compared MCR with ICB and DPOR on a variety of popular multithreaded benchmarks (shown in Figure 1) collected from recent concurrency studies [9, 16] including two real-world large applications. In this section, we focus on answering three questions: 1. How efficient and effective is MCR in finding concurrency errors? 2. How efficient and effective is MCR in exploring state-spaces? 3. How scalable is MCR for real programs?

Evaluation Methodology For the first two questions, we use the same set of benchmarks. Each benchmark has at least one known concurrency error that only manifests rarely at runtime as assertion violations, exceptions, etc. We compare the time and the number of executions for each technique to find the known error. To evaluate MCR for exploring state-spaces, we further run the fixed version of these benchmarks (by either fixing the bug or disabling the runtime exception). In addition, during exploration we perform data race and NPE detection to evaluate the effectiveness of each technique for exposing these two types of bugs.

We use `Jigsaw` and `Weblech` to evaluate the scalability of MCR. `Jigsaw` is a web server application from W3C and `Weblech` is a website download tool. Both programs have a test driver that starts the server, performs client requests, and terminates. The executions of these two programs have many more events than the other benchmarks. For example, the number of events in `Jigsaw` is more than 36K executed by 12 threads. No prior study of stateless model checking has evaluated on such a large scale. We run `Jigsaw` and `Weblech` with each technique and set a time bound of an hour. Although no technique can finish in an hour, we compare the number of detected data races and NPEs to show the improvement of MCR over existing techniques. We also run our parallel algorithm in these experiments to assess the scalability of MCR with parallelization.

All experiments were conducted on an 8-processor 32-core 3.6GHz Intel i7 Linux with 8GB memory and JDK 1.7 8GB heap space. All data were averaged over three runs.

5.1 Benchmark Bug Finding Results

Table 2 summarizes the results for finding the known errors in the benchmarks. Overall, MCR takes significantly fewer executions than ICB and ICB+DPOR. In most cases, the number of runs taken by MCR is orders of magnitude smaller than ICB and ICB+DPOR. For example, for `BubbleSort`, ICB took 592 executions to trigger the runtime assertion violations, and ICB+DPOR took 400 executions, whereas MCR only took 4 executions. In particular, for `MTList` and `MTSet` (two multithreaded tests for Java synchronized `LinkedList` and `HashSet`), because both of them contain many more threads and events than the other benchmarks, ICB ran out of memory and ICB+DPOR took more than 5000 executions

Table 1: Benchmarks. Each benchmarks has at least one known error that causes runtime exceptions under certain interleavings.

Program	LoC	#Thrd	#Evt	Description
Example	79	3	32	The example program shown in Figure 2 with N=1.
Account	373	5	51	Concurrent account deposits and withdrawals suffering from atomicity violations.
Airline	136	6	67	A race condition causes the tickets sold more than the capacity.
Allocation	348	3	125	An atomicity violation causes the same block allocated or freed twice.
BubbleSort	175	5	133	Bubble sorting an array without proper synchronization causing incorrect sort results.
MTList	5979	27	685	Buggy synchronized LinkedList library test throwing ConcurrentModificationException.
MTSet	7086	22	724	Buggy synchronized HashSet library test throwing ConcurrentModificationException.
PingPong	388	6	44	The player is set to null by one thread and dereferenced by another throwing NPE.
Pool107	10K	3	170	Concurrency bug in Apache Commons Pool causing more instances than allowed in the pool.
StringBuf	1339	3	70	An atomicity violation in Java StringBuffer causing StringIndexOutOfBoundsException.
Weblech	35K	3	2045	A tool for downloading websites and emulating standard web-browser behavior.
Jigsaw	380K	12	36K	A web server application from W3C providing full HTTP 1.1 functionality.

Table 2: Results on finding known errors in benchmarks.

Program	#Executions/Total time		
	ICB	ICB+DPOR	MCR
Example	77322/20s	3782/3s	**46**/2s
Account	111/0.2s	20/0.2s	**2**/0.3s
Airline	669/1.8s	19/0.8s	**9**/3s
Allocation	15/0.1s	8/0.3s	**2**/0.3s
BubbleSort	592/1.2s	400/2.7s	**4**/4.8s
MTList	**OOM**/-	5173/290s	**8**/97s
MTSet	**OOM**/-	5480/267s	**21**/159s
PingPong	648/3s	37/0.5s	**2**/0.7s
Pool	24/0.3s	6/0.3s	**3**/0.4s
StringBuf	12/0.1s	10/0.5s	**2**/0.4s

to find the error. Nevertheless, MCR took only 8 and 21 runs, respectively, because it eliminated a large space of redundant interleavings.

Comparing the overall performance (including both the online exploration and offline analysis time), MCR took even less time than ICB and DPOR in almost half of the benchmarks, though MCR requires building and solving constraints which takes extra time. For instance, MCR took one third of the time taken by ICB+DPOR (97s vs 290s) for `MTList` and nearly half (159s vs 267s) for `MTSet`. The reason is that in these benchmarks the gain by reducing redundant interleavings outweighs the cost of constraint analysis. Moreover, the performance of MCR can be improved significantly through parallelism. We report the results of our parallel algorithm for the two real programs in Section 5.3.

5.2 Benchmark State Space Exploration Results

Table 3 summarizes the state-space exploration results for the benchmarks. For each benchmark, a technique may either finish the execution normally, meaning that the state-space has been completely explored (✔) in an hour, or terminate early by running out of memory (✘), or timeout (⛀). Columns 2-4 report the execution outcome of each technique. Columns 5-7 report the number of explored executions and the total time. Columns 8-10 report the number of data races and NPEs detected during the exploration.

Overall, MCR is much more effective and efficient than ICB and ICB+DPOR in exploring state-space and finding bugs. For most benchmarks, ICB did not finish in an hour, because ICB has to explore all the possible interleavings, the size of which is huge even for small programs. For half of the benchmarks, ICB+DPOR was able to finish. For the others, however, it either ran out of memory or did not finish. Comparatively, MCR finished exploration for most benchmarks except `BubbleSort` and `MTList`/`MTSet`. We next discuss the results on several interesting benchmarks.

Table 4: Results on real applications. * means OOM.

Program		ICB	ICB+DPOR	MCR	MCR-P
Jigsaw	#Race	2	7	20	38
	#NPE	1	2	6	10
	#Run	307*	425*	32	769
Weblech	#Race	4	4	6	7
	#NPE	0	0	1	1
	#Run	1229*	1072*	185	3311

Airline MCR took 8 executions and 4.5s to explore the entire state-space, while ICB explored 325,891 executions in an hour and did not finish, and ICB+DPOR explored 3000 executions. Notice that the online execution of this benchmark is much faster than offline constraint analysis, so ICB+DPOR took less time than MCR even it explored many more executions. The number of executions explored by MCR is even smaller than that reported in Table 2 in finding the error. The reason is that the bug (an atomicity violation) is fixed by adding synchronizations, which reduces the number of possible interleavings.

BubbleSort This benchmark has more than 10 million interleavings. None of the techniques was able to finish within an hour. ICB ran out of memory, ICB+DPOR explored 326,647 executions, and MCR explored 13,981. Although MCR explored fewer online executions than ICB+DPOR, the interleavings in these executions are much more valuable, as they are all maximal causally distinct to each other. This is further validated by the fact that MCR detected more data races than ICB+DPOR (7 vs 6) in these executions.

MTList/MTSet The state-spaces of these two benchmarks are much larger than the others. Both ICB and ICB+DPOR ran out of memory and MCR explored 382 and 457 executions respectively in an hour. However, MCR detected many more data races and NPEs than the other two techniques. In `MTList`, MCR detected 8 data races and one NPE, while both ICB and ICB+DPOR detected only one data race and none NPE. In `MTSet`, MCR detected 6 data races and 4 NPE, while both ICB and ICB+DPOR detected 5 data races and none NPE. Moreover, MCR found a new exception (`NoSuchElementException`) in both benchmarks.

PingPong This benchmark has a known NPE error. All techniques found the NPE and both MCR and ICB+DPOR detected 6 data races (one more than ICB). However, neither ICB nor ICB+DPOR was able to finish exploration in an hour, though they explored 343,728 and 972,799 executions respectively. Comparatively, MCR finished exploration in only 13s with 411 online executions. The advantage of MCR over ICB and DPOR exposed in this benchmark is not the number of bugs found, but the verification confidence that MCR has explored the entire state-space of this benchmark (corresponding to the given input), and there is no more data race or NPE other than those 7 data races and 1 NPE.

Table 3: Results on state-space exploration of benchmarks. MCR found new exceptions (tagged with *) in MTList/MTSet.

Program	Finished(✔), Timeout(⛔), OOM(✘)			#Executions/Total time			#Race|#NPE		
	ICB	ICB+DPOR	MCR	ICB	ICB+DPOR	MCR	ICB	I+D	MCR
Example	⛔	✔	✔	3294109/1h	25522/10s	50/2s	7|0	10|0	10|0
Account	⛔	✔	✔	1499507/1h	875/2s	3/0.5s	3|0	3|0	3|0
Airline	⛔	✔	✔	325891/1h	3000/3.5s	8/4.5s	0|0	0|0	0|0
Allocation	✘	⛔	✔	–	1354979/1h	30/5.6s	0|0	0|0	0|0
BubbleSort	✘	⛔	⛔	–	326647/1h	13981/1h	4|0	6|0	7|0
MTList	✘	✘	⛔	–	–	382/1h	1|0	1|0	8|2*
MTSet	✘	✘	⛔	–	–	457/1h	5|0	5|0	6|5*
PingPong	⛔	⛔	✔	342728/1h	972799/1h	411/13s	6|1	7|1	7|1
Pool	⛔	✔	✔	509852/1h	1547/1.9s	3/0.9s	0|0	0|0	0|0
StringBuf	⛔	✔	✔	1340718/1h	427/0.8s	3/0.4s	0|0	0|0	0|0

5.3 Real Application Exploration Results

Table 4 reports our results on `Jigsaw` and `Weblech`. The rows *#Race*, *#NPE*, and *#Run* report the number of data races, NPEs, and executions detected and explored by each technique. MCR-P corresponds to our parallel algorithm. As expected, no technique was able to finish exploration within an hour. ICB and ICB+DPOR even ran out of memory on both of these two programs. For `Jigsaw`, ICB explored 307 executions, ICB+DPOR 425, and MCR 32 before they terminated or timed out. For `Weblech`, ICB explored 1229 executions, ICB+DPOR 1072, and MCR 185.

Although MCR explored fewer executions than ICB and DPOR (because the offline analysis takes more time for longer executions), it detected many more data races and NPEs. For `Jigsaw`, MCR detected 20 data races (13 more than ICB+DPOR and 18 more than ICB) and 6 NPEs (4 more than ICB+DPOR and 5 more than ICB). For `Weblech`, MCR detected 6 data races and 1 NPE, while both ICB and ICB+DPOR detected 4 data races and none NPE. Note that all the reported data races and NPEs are distinct (on different program locations). Moreover, by parallelizing our algorithm on a 32-core machine, MCR-P was able to explore many more executions and detect more data races and NPEs within the same time. For `Jigsaw`, MCR-P was able to explore 769 executions and detected 38 data races and 10 NPEs, and for `Weblech`, MCR-P explored 3311 executions and detected one more data race than MCR.

New data races and NPEs found Both `Jigsaw` and `Weblech` have been studied frequently in previous research [16, 29]. In our experiments, MCR and MCR-P also found two new races and two new NPEs in `Jigsaw` and one new race in `Weblech`. These races and NPEs were not detected by ICB or DPOR and have not been reported before. One race in `Jigsaw` is on field `http.CommonLogger.errlog` in class `CommonLogger` between two statements in methods `errlogmsg` and `openErrorLogFile`. The other race is on `http.httpd.finishing` in class `httpd` between methods `run` and `shutdown`. Both NPEs are on field `http.httpd.logger`, which is dereferenced in methods `log` and `errlog` but set to `NULL` in method `cleanup`. The race in `Weblech` is on field `downloadInProgress` in method `run`.

5.4 Discussion

Our experimental results clearly demonstrate the superior performance of MCR over the state-of-the-art. Although for large programs it is still hard for MCR to explore the whole state-space within a reasonable time, MCR serves as a valuable augmentation to existing predictive techniques for practical bug finding and testing of concurrent programs. To scale to even larger real-world programs, we have also identified a few challenges that we plan to address in future.

Scalable constraint solving As noted in Section 3.2, the size of MCM constraints is cubic in the number of shared data accesses. For long executions, the corresponding constraints can be very large. Even with a high performance Z3 solver and an efficient IDL decision procedure, the constraints may still be hard to solve in a reasonable time. In fact, the majority of execution time in our experiments with MCR on `Jigsaw` and `Weblech` is spent within the solver. One direction to improve the scalability of MCR is to develop a customized solver tailored to the MCM constraints.

Non-terminating programs and fairness Most realistic programs are non-terminating if thread-fairness is not considered. For instance, a thread may spin forever on a loop condition `while(!flag);` if the scheduler continuously runs the thread without giving a chance for other threads to execute. Because every iteration of the loop produces a new read event, MCR will generate new seed interleavings for it, hence the exploration process will never terminate. However, for the purpose of effective bug finding, we should deprioritize exploring the seed interleavings for repetitive read events. Prior work [24] has explored fairness in stateless model checking. We plan to integrate the technique to make MCR thread-fairness aware.

Input space and non-determinism Real programs have both large scheduling and input spaces. MCR only reduces redundant interleavings wrt the fixed input. It is a known challenge to explore for all inputs, which will enable the full verification. In addition, stateless model checking generally assumes the input is deterministic across runs. If the input is non-deterministic, the special scheduler may fail to enforce the seed interleavings. To address this problem, we can leverage existing work on record and replay [14, 27] to capture all non-deterministic input sources. We plan to further investigate efficient solutions for ensuring input-determinism and identifying redundant interleavings across different inputs.

6. Related Work

Stateless model checking has been an active research area since the pioneering work of VeriSoft [13]. Since then a large effort has been invested in reduction techniques to combat the explosion of interleaving space. Partial order reduction (POR) [5, 10] and context bounding [23, 25] are the two most effective approaches known so far and algorithms combining them are also proposed [6, 22, 31]. Various techniques based on persistent set [5], sleep set [10], and source set [3] have been proposed to optimize the performance and effectiveness of POR. Fundamentally, the reduction effectiveness of POR is limited by happens-before relation: it cannot reduce redundant interleavings that have different happens-before relation. Differently, MCR overcomes happens-before by exploring the maximal causality between schedules to achieve the maximal reduction.

The maximal causal model (MCM) was first presented as a theoretical result in [30]. In our prior work RVPredict [16], we extended MCM with control flow and encoded it with SMT constraints for race detection. This work is built upon RVPredict with two key improvements. First, we extend RVPredict to perform model checking with maximal causality reduction. A crucial step is the generation of seed interleavings to explore the full state space, which is not addressed by prior work. In addition, we realize the full MCM with our data-validity constraint in Sec 3.2, while RVPredict does not but is specialized for predicting races.

Another related approach is monotonic partial order reduction (MPOR) [18]. MPOR also formulates concurrent program executions as constraints and solve them with SMT solvers. Differently, MPOR does not achieve maximal reduction and cannot be used for stateless model checking as it does not generate seed interleavings. The constraint-based approach has also been used to find concurrency bugs [28, 32] and to reproduce concurrency failures [15].

A direct application of stateless model checking is systematic concurrency testing. Unlike conventional testing techniques that may end up repeatedly executing the same interleaving, stateless model checking systematically explores all legal but distinct interleavings. To improve the scalability of testing realistic concurrent programs, various approaches such as coverage-driven [33, 34], priority-based [4, 17, 26], fairness-based [24], and assertion-guided techniques [19] have been proposed and shown effective in practice finding concurrency bugs. Nevertheless, these techniques only try to select or prioritize schedules but do not reduce redundant interleavings. Moreover, they do not provide any guarantee of state converage and may miss bugs. MCR is orthogonal to these techniques and can be combined with them to improve scalability.

7. Conclusion

We have presented maximal causality reduction (MCR), which minimizes the number of explored executions for stateless model checking concurrent programs based on the foundation of maximal causal model. We have designed and implemented MCR using a constraint-based approach and shown that it significantly improves the efficiency and effectiveness of existing techniques for state-space exploration and bug finding on both benchmarks and real programs. Moreover, MCR shifts the runtime exploration cost to embarrassingly parallel offline analysis, a promising approach to scale model checking to large concurrent programs.

Acknowledgment

Part of this work was done while the author was at University of Illinois at Urbana-Champaign and supported by the DARPA HACMS program as SRI subcontract 19-000222. The author wishes to thank Grigore Rosu for his valuable advice and Qingzhou Luo for many useful discussions on early idea of this work, and anonymous PLDI reviewers for helpful suggestions that have improved this paper. Special thanks go to Lawrence Rauchwerger, Jaakko Järvi, and Dilma Da Silva at Texas A&M University for their review of the manuscript and insightful comments.

References

[1] A. Mazurkiewicz. Trace theory. *Advances in Petri Nets*, 1987.

[2] ASM bytecode analysis framework. `http://asm.ow2.org/`.

[3] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *POPL*, 2014.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[5] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *STTT*, 1998.

[6] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *OOPSLA*, 2013.

[7] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[8] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, 2006.

[9] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

[10] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, University of LiÃÍge., 1996. ISBN 3540607617.

[12] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.

[13] P. Godefroid. Software model checking: The verisoft approach. *Form. Methods Syst. Des.*, 2005.

[14] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

[15] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *PLDI*, 2013.

[16] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[17] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[18] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, 2009.

[19] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *ASE*, 2014.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[21] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *ESEC-FSE*, 2007.

[22] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. In *Tech. Rep. MSR-TR-2007-12*, 2007.

[23] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[24] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI*, 2008.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[26] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, 2012.

[27] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, 2010.

[28] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NFM*, 2011.

[29] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[30] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.

[31] J. van den Hooff. Fast bug finding in lock-free data structures with cb-dpor. In *Master thesis, Massachusetts Institute of Technology*, 2014.

[32] C. Wang, R. Limaye, M. K. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.

[33] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE*, 2011.

[34] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.