Speeding Up Maximal Causality Reduction with Static Dependency Analysis*

Shiyou Huang[1] and Jeff Huang[2]

1 **Texas A&M University, College Station, USA**
`huangsy@tamu.edu`
2 **Texas A&M University, College Station, USA**
`jeff@cse.tamu.edu`

Abstract

Stateless Model Checking (SMC) offers a powerful approach to verifying multithreaded programs but suffers from the state-space explosion problem caused by the huge thread interleaving space. The pioneering reduction technique Partial Order Reduction (POR) mitigates this problem by pruning equivalent interleavings from the state space. However, limited by the happens-before relation, POR still explores redundant executions. The recent advance, Maximal Causality Reduction (MCR), shows a promising performance improvement over the existing reduction techniques, but it has to construct complicated constraints to ensure the feasibility of the derived execution due to the lack of dependency information.

In this work, we present a new technique, which extends MCR with static analysis to reduce the size of the constraints, thus speeding up the exploration of the state space. We also address the redundancy problem caused by the use of static analysis. We capture the dependency between a read and a later event `e` in the trace from the system dependency graph and identify those reads that `e` is not control dependent on. Our approach then ignores the constraints over such reads to reduce the complexity of the constraints. The experimental results show that compared to MCR, the number of the constraints and the solving time by our approach are averagely reduced by 31.6% and 27.8%, respectively.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Model Checking, Dynamic Analysis, Program Dependency Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.16

1 Introduction

Concurrent programs are error prone. Moreover, it is notoriously difficult for developers to find and reproduce those concurrency bugs because they only manifest in specific interleavings. *Stateless Model checking* [11] (which we refer to as SMC in this paper) offers a promising solution to combat this challenge by systematically exploring all the possible interleavings of the program. Since the pioneering work of VeriSoft [11, 12] and CHESS [24], SMC has been successfully applied in real-world programs and has found many deep bugs. To mitigate the *state explosion* problem in SMC, a great effort has been dedicated to reduction techniques such as *partial order reduction* (POR) [3, 10, 13] which prunes redundant executions from the state-space and search strategies such as context (or preemption) bounding [24] which prioritizes executions with fewer context switches in a given state-space.

* This work was supported by NSF award CCF-1552935.

However, POR is limited by the happens-before relation and may explore redundant executions. To maximally reduce redundancy, Huang [16] recently develops a new reduction technique called *Maximal Causality Reduction* (MCR), which gains a promising performance improvement over prior reduction techniques. To explore the maximal causality between redundant executions that lead to equivalent states, MCR takes the values of the reads and writes into consideration and constructs first-order constraints over the events in the trace to generate schedules. As the new schedule contains at least one read that returns a different value from that in the prior trace, the program reaches a new state if it is executed following the derived schedule.

However, MCR is purely dynamic and it only collects information (values and addresses, etc.) from the trace, which does not reflect the dependency relation of two events. As a result, MCR has to conservatively enforce all the reads that happen before a considered event e to return the same value (Section 2) as that in the current trace so that e is reachable in the derived schedule. Consider the following code snippet.

```
          int counter = 0;
//thread t1:              //thread t2:
while (i++ < Max)         while (i++ < Max)
  counter += 1;             counter -= 1;
```

This program contains two threads with one global variable *counter*, one thread increasing the *counter* but the other decreasing it. The loop iteration in the program is decided by Max. For ease of presentation, we extend the while loop with $Max = 2$ and execute the program in the program order. The execution by each thread is an alternation of reads and writes to the shared variable counter, *e.g.*, r1-w1-r2-w2. MCR enumerates all the reads in the trace and considers all the possible values that each read can return (more details are given in Section 2). To ensure the reachability of the considered read r, MCR enforces the reads that happen before r to return the same value (Section 3.1). For example, if MCR considers the second read r2 in the trace, it will enforce the first read r1 to return the same value to ensure the reachability of r2. This is because MCR does not know whether or not the value returned by r1 can influence the evaluation of a predicate (*e.g.*, a if statement), thus affecting the execution of a later event, such as r2. With the number of reads and writes increasing in the trace, MCR needs to construct expensive constraints to ensure the reachability of an event, which on the one hand consumes more memory and on the other more time for the solver to solve the constraints.

In light of the limitation, the main question we consider is the following: Can we skip those reads (*e.g.* r1) that happen before a target event (*e.g.* r2) in the exploration, thus reducing the constraints? Combining with the program's information, we can figure out whether a read (*e.g.* r1) affects the reachability of another (*e.g.* r2). The key contribution of this work is to integrate the static dependency analysis into the dynamic exploration to reduce the complexity of the first-order constraints. Although the dependency information provided by the static analysis may be imprecise due to the limitations of all classic static analysis, we discuss that the soundness of the dynamic exploration is not impacted by the imprecision in Section 4.3. We use the system dependency graph (SDG) of the program to identify whether a read has a control or data dependency on an event in the trace. Then in the exploration of new schedules from a given trace, we rely on such dependency information to construct constraints to only make the dependency-related reads return the same value.

Different from *program slicing* [28, 6] which computes a set of the statements that can influence the value of a given point, our approach aims to locate the reads that can impact the evaluation of a predicate that determines the execution of a given point. By our approach, the number of the constraints and the solving time of the above example (when the value of

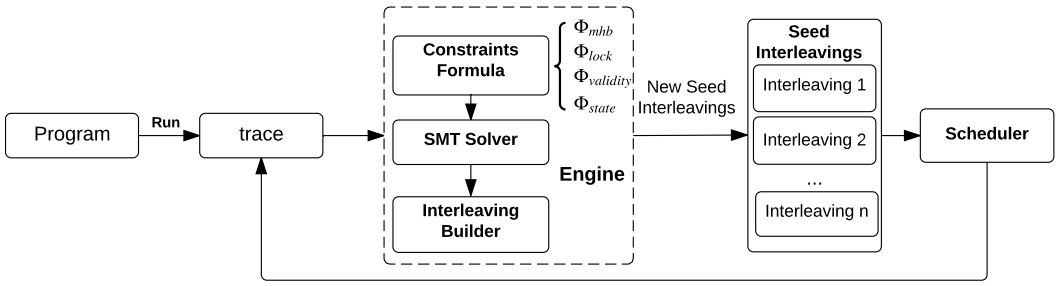

Figure 1 Workflow of MCR. The engine part of MCR constructs SMT constraints over the trace to explore new program schedules, and the new trace is generated by re-executing the program under the dynamic scheduler.

Max is 5) are reduced by 35.1% and 44.6%, respectively.

We have implemented our technique based on *JOANA* [1, 14] and *WALA* and evaluated it with a collection of multithreaded Java programs, including two large real-world applications, `Derby` and `Weblech`. On average, our approach reduces the number of the constraints and the solving time by 31.6% and 27.8%, respectively, compared to MCR. We also evaluate the total time used to search the state space by our approach. Because it takes time to check the dependency relation of two events in the exploration, the total time used to search the state space is not reduced significantly on small benchmarks, although the size of constraints for these programs is significantly reduced. But for `Derby` and `Weblech`, our approach reduces the total time by 14.1% and 43.1%, respectively, compared to MCR.

In summary, this paper makes the following contributions:

- We extend MCR with static dependency analysis to reduce the size of the SMT constraints and hence speed up the state space exploration of MCR (Section 4).
- We analyze the redundancy caused by static analysis and present a modified algorithm to avoid the redundancy (Section 5).
- We validate the effectiveness of our technique on real-world Java programs and the experimental results show promising performance improvement over MCR with respect to the size of the constraints and the solving time as well as the total time of state space exploration (Section 6).

The rest of the paper is organized as follows: Section 2 reviews the key insight of MCR; Section 3 introduces the background of SDG and our motivation of this work; Section 4 presents our approach to reducing constraints; Section 5 discusses the redundant exploration by our approach; Section 6 reports our experimental results; Section 7 discusses related work and Section 8 concludes this paper.

2 Maximal Causality Reduction

This section reviews the key insight of MCR [16]. As Figure 1 illustrated, given a program with a fixed input, MCR systematically explores all unique interleavings of the program in a closed loop, with each explored interleaving covering a unique program state. At first, the instrumented program is executed in a random order to generate the initial trace that is taken as the input by the engine. Then given an executed trace τ, the `engine` encodes τ into an SMT constraints formula ($\Phi^{mc} = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{validity}$) to compute a proved maximal set of traces, denoted as $\mathsf{MaxCausal}(\tau)$, which contains all the feasible schedules that can be derived from τ [17]. To prune the redundant executions in $\mathsf{MaxCausal}(\tau)$, Φ^{mc} is then

conjoined with a new state constraint Φ_{state} to generate a final formula $\Phi = \Phi^{mc} \wedge \Phi_{state}$ that is used to generate a seed interleaving. A `seed interleaving` is a feasible thread schedule that drives the program to reach a new state that is not explored before. The essential insight of Φ_{state} is to enforce the reads in the trace τ to return *different values* from that in τ allowed by the SMT constraints formula. By re-executing the program under the scheduler following the seed interleaving, the program will reach a new state and the trace generated will be the input of the engine.

In MCR, the following common types of events are considered:

- `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.

To encode Φ, for each event in the given trace τ, MCR uses an integer variable O to denote its order in a certain feasible trace in $\mathsf{MaxCausal}(\tau)$ and encodes the following constraints over the variables O:

1. must-happen-before constraints (Φ_{mhb});
2. lock-mutual-exclusion constraints (Φ_{lock});
3. data-validity constraints ($\Phi_{validity}$);
4. New state constraints (Φ_{state}).

Must-happen-before (MHB) constraints (Φ_{mhb})

The Φ_{mhb} constraint ensures a minimal set of *happens-before* relations that events in any feasible interleaving must obey. It requires that (1) All events by the same thread should happen in the program order (assuming *sequential consistency*); (2) The *begin* event of a thread should happen after the *fork* event that starts the thread; (3) A *join* event for a thread should happen after the last event of the thread.

Lock-mutual-exclusion constraints (Φ_{lock})

The Φ_{lock} constraint ensures that events guarded by the same lock are mutually exclusive. It is constructed over the ordering of the *lock* and *unlock* events. More specifically, for each lock, MCR extracts all the *lock/unlock* pairs of events and constructs the following constraints for each two pairs (l_1, u_1) and (l_2, u_2): $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

Data-validity constraints ($\Phi_{validity}$)

The $\Phi_{validity}$ constraint ensures that all events in any trace in $\mathsf{MaxCausal}(\tau)$ are reachable. For an event e to be reachable, all events that must-happen-before e must be feasible, and every read event that e depends on (excluding e itself) should read the same value as it reads in τ. A concrete example will be given to illustrate this in Section 3.1.

New state constraints (Φ_{state})

To eliminate redundant executions, MCR enforces at least one read event in each explored execution to read a new value, so that no two executions reach the same state. MCR enumerates each read event in τ on the set of all values by the writes on the same memory address. For each value that is different from what it reads in τ, a new state constraint is generated to force the read to read the new value. Consider a read *r=read(t,x,v)* on x with

value v, and a value $v' \neq v$ written by any write on x, Φ_{state} is written as $\Phi_{value}(r, v')$. Since all such state constraints are generated, MCR ensures that no non-equivalent interleaving is missed. Hence the entire state-space will be covered systematically by MCR.

Example

We use the upcoming example to illustrate how MCR works, and we assume all the examples in this paper are executed under sequential consistent (SC) memory [21]. For ease of presentation, we use e_i to denote the event at line i and O_i (an integer variable) to represent the order of e_i in the trace. For example, if $O_i < O_j$, then e_i will be executed before e_j in the generated interleaving. We keep the notations in the rest of the paper.

```
        initially x = y = 0;
thread 1:                thread 2:
1: x = 1;   /*w(x)*/      3: y = 1;   /*w(y)*/
2: a = y;   /*r(y)*/      4: b = x;   /*r(x)*/
```

Listing 1 An example illustrating how MCR works.

The program has 6 different executions, 3 of which are redundant. MCR is able to explore all the state-space via only 3 executions.

Suppose in the initial execution, MCR obtains the trace $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$ under SC, and the program reaches the state (a=0,b=1). MCR constructs the MHB constraints $\Phi_{mhb} = O_1 < O_2 \wedge O_3 < O_4$. As the trace contains two reads, $r(y)$ and $r(x)$, to generate new seed interleavings, MCR enforces each of the two reads to read a different value in future executions. For example, it adds the new state constraint $\Phi_{state} = O_3 < O_2$ to enforce $r(y)$ to read from $w(y)$ and return the value 1. By solving this constraint conjoined with Φ_{mhb}, the SMT solver will return a solution: $\{O_1 = 1, O_2 = 3, O_3 = 2\}$. From this solution, MCR generates a new seed interleaving e_1-e_3-e_2, because $O_1 < O_3 < O_2$. By re-executing the program following this seed interleaving, MCR will obtain a new execution $\tau_1 = \langle e_1, e_3, e_2, e_4 \rangle$, and reach a new state (a=1,b=1). In the new trace, the order of the event that occurs in the seed interleaving is fixed and MCR only considers the rest of the events, e_4 ($r(x)$) in this example. Because there is no new value that $r(x)$ can return, the exploration along this seed interleaving is finished. Likewise, to consider the second read event $r(x)$ in τ_0, MCR generates a new seed interleaving e_3-e_4, which produces a new execution $\tau_2 = \langle e_3, e_4, e_1, e_2 \rangle$ that reaches a new state (a=1,b=0). As there is no new state that can be generated from $e_1 - e_2$, the exploration is finished. MCR successfully explores all the three different program states – (a=0,b=1), (a=1,b=1) and (a=1,b=0) – through only three different executions.

3 Motivation and Technical Background

In this section, we discuss the importance and the complexity of $\Phi_{validity}$ constraints via a simple example. We then introduce the background of the system dependency graph (SDG), which we rely on to simplify $\Phi_{validity}$ (Section 4).

3.1 Motivation

The motivation of this work stems from the observation that when running MCR on real-world large programs, it can take a long time to solve the constraints formula even with a powerful SMT solver, like *Z3* [9]. The reason for this is that when MCR encodes long traces, especially those with lots of reads and writes, it generates extremely huge constraints and a large part of them are data-validity constraints ($\Phi_{validity}$). As illustrated in Section 2,

the constraints Φ_{mhb} and Φ_{lock} ensure that the computed interleaving obeys the semantics of the given memory model. However, to make the generated interleaving feasible, MCR also considers the reachability of an event that might be control dependent on a prior read. Consider the following program.

```
          initially x = y = 0;
thread 1:                    thread 2:
1: if (x==0)   /*r1(x)*/     3: x = 1; /*w(x)*/
2:    r = x;   /*r2(x)*/
```

Suppose initially the program is executed in the order, $e_1 - e_2 - e_3$, and the program reaches the state $r1(x) = 0$ and $r2(x) = 0$. To make $r2(x)$ return the value 1 written by $w(x)$, MCR enforces $\Phi_{state} = O_3 < O_2$ so that e_3 happens before e_2. By conjoining with $\Phi_{mhb} = O_1 < O_2$, the solver reports a possible solution $O_3 = 0, O_1 = 1, O_2 = 2$, corresponding to a concrete schedule $e_3 - e_1 - e_2$. However, this schedule is infeasible because the if predicate is not satisfied under this schedule, and hence e_2 cannot be executed. To ensure the reachability of an event, MCR encodes the data-validity constraints into the formula. In a word, all the reads that happen before the considered event should hold the same value as that in the prior execution. In this example, when we consider the value of $r2(x)$, we need to guarantee that $r1(x) = 0$. Then a correct schedule that makes $r2(x) = 1$ is $e_1 - e_3 - e_2$. Let $\prec_e$ denote the set of events that must-happen-before an event e and $r =$ `read(t,x,v)` denote a read event in $\prec_e$ on a memory location x with value v by thread t. Let W^x denote the set of all writes to x, and W^x_v the set of writes to x with value v, the data-validity constraint for e is encoded as

$$\Phi_{validity} = \bigwedge_{r \in \prec_e} \Phi_{value}(r, v),$$

where $\Phi_{value}(r, v)$ is the state constraint that ensures r to read a value v:

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

This constraint is complex because it is recursive. As we can see, to match a read r with a write w, MCR also needs to ensure the reachability of w, which requires all the reads that must-happen-before w should return the same value. It means we also need to construct constraints to match those reads with specific writes. Unfortunately, as most events in a trace are read or write, it can be very expensive to make all the reads in $\prec_e$ return the same value. The second observation of this work is that some reads in $\prec_e$ actually do not influence the reachability of e so that we can remove them from $\prec_e$ to reduce the size of the constraints. For example, for two reads `r1-r2` executed by the same thread, there is no need to consider the value returned by `r1` when constructing $\Phi_{validity}(e)$ because there is no dependency between the two reads. Our idea for reducing the size of $\Phi_{validity}$ is to only enforce the reads in $\prec_e$, which the event e is control dependent on, to return the same value. To achieve this idea, we use static analysis on the source code of the program – system dependency graph, to compute the dependencies between two events. Next we first introduce the knowledge of system dependency graph in Section 3.2 and then present the details of our approach in Section 4.

3.2 System Dependency Graph

The system dependency graph (SDG) for a program P, denoted by $G_p = (N, E)$, is a directed graph, where the nodes in N represent the statements or predicates in P and the edges in E

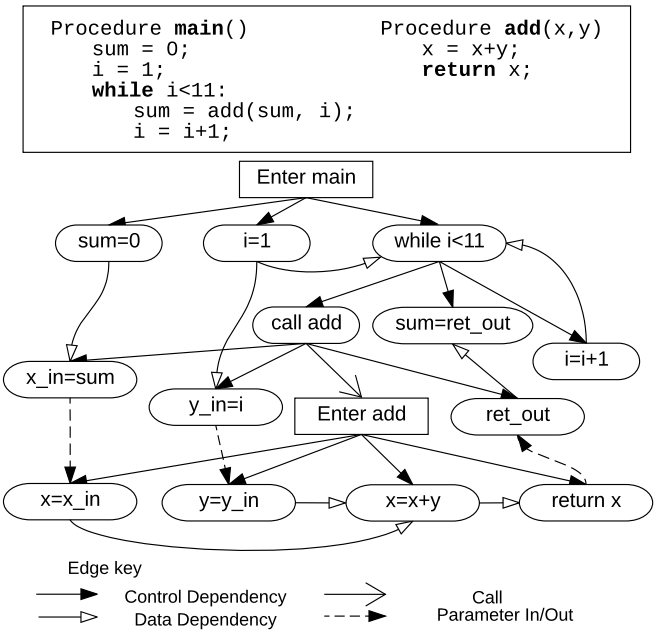

Figure 2 The System Dependency Graph of a concrete program, where the dependencies are distinguished by different edges.

represent the dependencies between the nodes [15]. Figure 2 presents an SDG of a concrete program, which includes a procedure call add in the main procedure. An SDG is made of the *procedure dependency graphs* (PDGs), which model the system's procedures. In a PDG, all the nodes are connected by either *control dependency* edges or *data dependency* edges. A node m is control dependent on the node n if the evaluation of n controls the execution of m. The source of a *control dependency* edge is either an *enter* node or a *predicate* node. A *data-dependency* between two nodes indicates that the program's computation might be changed if the relative order of the two events represented by the two nodes are reversed. In the SDG, all the PDGs are connected by the edges between the *call sites* nodes and the *enter* nodes of the called procedures. For example, in Figure 2, there exists a procedure call add in the main procedure. The two PDGs are connected by a *call edge* from *call add* node to the entry node *Enter add* of the procedure *add*. In SDG, for each parameter passing, there exists an *actual-in* node and *formal-in* node, which are connected by a *parameter-in* edge. For instance, when passing parameter x to the procedure *add*, the *actual-in* node *x_in=sum* is connected to the *formal-in* node *x=x_in* by a *parameter-in* edge (the dashed arrow). For each modified parameter and returned value, there also exists a *parameter-out* edge connecting the *formal-out* node and the *actual-out* node. *Formal-in* and *-out* nodes are control dependent on the *entry* node and the *Actual-in* and *-out* nodes are control dependent on the *call* node. The SDG permits us to analyze the dependency between two events presented by nodes in the graph by traversing the graph.

4 Our Approach

This section introduces how our approach leverages the SDG to reduce the *data-validity* constraints ($\Phi_{validity}$). We first present the overall algorithm and then the detailed dependency analysis.

4.1 Constraints Reduction

The essential idea for reducing $\Phi_{validity}$ is to reduce the number of the reads that are required to return the same value by MCR. We begin with the definition of the set of reads that an event is control dependent on to help illustrate the algorithm.

▶ **Definition 1.** Given an event e in a trace τ, $\prec_\tau (e)$ denotes the set of the reads that must-happen-before e, and $\prec_\tau^D (e) \subseteq \prec_\tau (e)$ denotes the set of reads that e is dependent on.

The main algorithm of our approach is presented as follows.

Algorithm 1: $\Phi_{validity}(e)$ Reduction

Input : τ - a trace and e - a given event in τ
Output : $\Phi_{validity}(e)$ - data-validity constraints related to e

1 $\Phi_{validity} = \emptyset$
2 $\prec_\tau (e) \leftarrow$ **Happens-before**(τ, e)
3 $\prec_\tau^D (e) \leftarrow$ **DependencyComputation**($\prec_\tau (e)$, e)
4 **foreach** *read* $r \in \prec_\tau^D (e)$ *with value* v **do**
  // $\Phi_{value}(r, v)$ recursively call *DataValidityConstraints()*
5 $\Phi_{validity} \wedge = \Phi_{value}(r, v)$
6 **end**
7 **return** $\Phi_{validity}$

Algorithm 1 shows how to compute data-validity constraints of a given event e. It takes as input the current executed trace τ and the considered event e. It first computes the set of reads that must-happen-before e (line 2) based on the constraints Φ_{mhb} in Section 2. Then our algorithm computes a subset of reads $\prec_\tau^D (e) \subseteq \prec_\tau (e)$, and all the reads in $\prec_\tau^D (e)$ have a dependency on e (line 3). We will give the details of the function **DependencyComputation**() in Section 4.2.3. The algorithm finally enforces that all the reads return the same value as that in the current trace τ according the encoding of $\Phi_{value}(r, v)$. The detailed expression of $\Phi_{value}(r, v)$ is presented in Section 3.1.

Because the number of the reads in $\prec_\tau (e)$ that e is dependent on takes a small portion of the total number of the reads in $\prec_\tau (e)$, our algorithm reduces the size of $\Phi_{validity}$ greatly. Meanwhile, the reduction will not lead to the missing of any executions explored by MCR.

Proof. To prove the correctness of this approach, it only needs to prove that our new constraints model $\Phi'_{validity}$ is equivalent to $\Phi_{validity}$ presented in Section 2 and 3.1 because all the rest part of Φ^{mc} remain the same. Consider a trace $\tau = e_1, e_2, \cdots, e_n$. To guarantee the reachability of an event $e_i \in \tau$ in a new schedule, we only need to make a read event $e \in \tau$ to return the same value and e is the last read that e_i is control dependent on. Since e is forced to return the same value, it guarantees that e is reachable and the path containing e_i is evaluated. Then no matter what values returned by the read between e and e_i, e_i is always executed. Therefore, our algorithm will not cause any infeasible executions or miss any executions. ◀

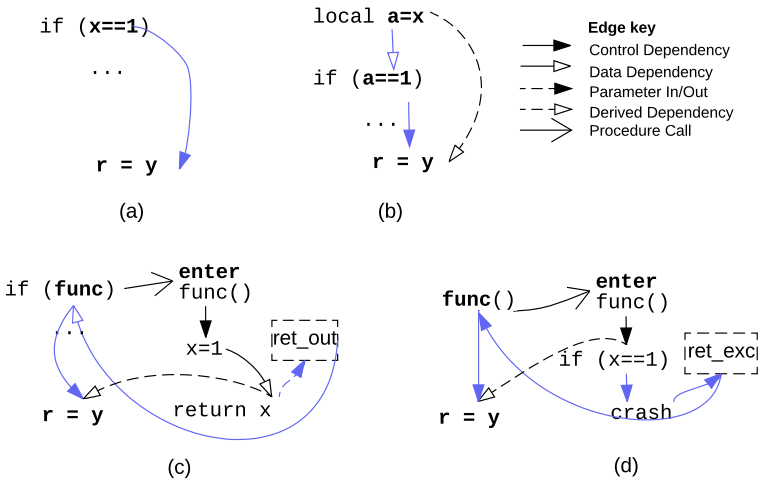

Figure 3 Four different cases where a read is control dependent on another marked by the blue edges.

4.2 Dependency Analysis

In this subsection, we present how we compute $\prec_\tau^D (e)$ based on the program's SDG from two parts, *control dependency* and *data dependency*. The insight for identifying that an event is dependent on another is to check if it exists a path in the SDG between the two events and the path satisfies a specific pattern. For the rest of the paper, we will abbreviate *control dependency CD*, *data dependency DD*, *call CL* and *parameter in/out PI/PO*. The reason why we distinguish *PI/PO* and *DD* is that the SDG that we construct via an existing tool *JOANA* [1, 14] contains these edges, and we use the type of the edge labeled by the graph to find the dependency relation. We use $n1 \xrightarrow{e^*} n2$ to denote that there is a path $p = e^*$ in SDG from node $n1$ to node $n2$, and each edge e in p belongs to one of *CD, DD, PI, PO* and *CALL*.

4.2.1 Control Dependency

We first discuss several situations where a read can influence the execution of a later event and then derive a rule of how to decide that an event is control dependent on a prior read from the general cases. In SDG, an event is control dependent on a predicate event that is either a *if* condition or *procedure call* related events. But the evaluation of the predicate is determined by the values returned by some reads. Our goal is to find those reads. We give the definition of a read that an event is control dependent on as following.

▶ **Definition 2.** An event **e** is control dependent on a read **r** if **r** is a read access to a shared variable, and **r** has data dependency on the predicate that decides the reachability of **e**.

We present four different cases in Figure 3 to help understand the definition and then summarize the rules to help identify the dependency between two events. The variables x and y in the figure are shared and all the others are local.

Case 1. Figure 3(a) shows the most direct control dependency between two events. The read $r = y$ is control dependent on the *if* predicate, which is data dependent on $x == 1$. As

$$n1\ \delta^c\ n2 \Leftrightarrow n1 \xrightarrow{e^*CD} n2,$$
$$e := \varepsilon \mid CD \mid DD \mid PI \mid PO \mid CL$$

Figure 4 Rule 1: the condition that a node has control dependency on another in SDG.

a result, the read $r = y$ is control dependent on the read $x == 1$ and the path between the two events is $x == 1 \xrightarrow{DD \cdot CD} r = y$.

Case 2. Besides direct control dependency, the evaluation of a predicate may depend on a prior read. As Figure 3(b) shows, although the evaluation of the *if* predicate is determined by the value of a, the read access to local variable a is data dependent on a prior read $a = x$. Therefore, according to Definition 2, $a = x$ is control dependent on $r = y$ and $x == 1 \xrightarrow{DD \cdot DD \cdot CD} r = y$

Case 3. Figure 3(c) illustrates the propagation of the control dependency between different procedures. The computation of the `if` predicate depends on the return value of the procedure `func()`. It implies that the reachability of a read operation might be decided by a read in another procedure. In this case, $r = y$ is control dependent on the read *return* x in `func()` and *return* $x \xrightarrow{PO \cdot DD \cdot DD \cdot CD} r = y$. Likewise, the dependency can also be transmitted by a PI edge in the graph. We omit the discussion of this case in the paper.

Case 4. In this case, the event `func()` has a special control dependency on `r=y`. As a procedure may crash (program exits abnormally) during the execution, all the executions that occur after the procedure call are not executed if the crash happens. SDG adds a control dependency edge, also denoted as CD, from the node `func()` to the node `r=y`. Through this edge, we derive $r = y$ is control dependent on $x == 1$ and $x == 1 \xrightarrow{CD \cdot CD \cdot CD \cdot CD} r = y$

As all the other cases are either the combination of the four basic cases above or can be derived using the same way, we only analyze the four basic cases in this paper. From the analyses on the four basic situations, now we summarize the rule to decide if an event is control dependent on a prior read in the program. We denote the *control dependency* between two events as δ^c: given two nodes $n1$ and $n2$ in an SDG, we use $n1\ \delta^c\ n2$ to denote that $n2$ is control dependent on $n1$. By analyzing the patterns of the paths in the four cases above, we derive that given any event e and a read r, to check $r\ \delta^c\ e$ is equivalent to check that if there is a path p ending with a *control dependency* edge from r to e, and each edge e in p belongs to one of *CD, DD, PI, PO* and *CALL*. We present the rule in Figure 4 to formalize this process.

4.2.2 Data Dependency

So far we have only considered the control dependency of the nodes. In this Section, we will point out that under some cases, the reads on which an event is data dependent on should also be added to the read set $\prec_\tau^D (e)$. Recall that when MCR maps a read to a certain write w, the data validity constraints in Section 2 also need to guarantee the reachability of w. We have illustrated in Section 4.2.1 that to ensure the reachability of an event e in the trace τ, we only need to ensure the reads in $\prec_\tau^D (e)$ to return the same value. However, we also

$$n1\ \delta^d\ n2 \Leftrightarrow n1 \xrightarrow{e^*DD} n2,$$
$$e := \varepsilon \mid DD$$

Figure 5 Rule 2: the condition that a node has data dependency on another in SDG.

need to guarantee that the value written by w matches with the one expected by the read in $\prec_\tau^D (e)$. Take the following program as an example.

```
                    int x = y = 0;
//thread 1:          //thread 2:                 //thread 3:
1: r = y; /*r1(y)*/  2: x = 1; /*w1(x)*/        4: x = 2; /*w2(x)*/
                     3: y = x; /*w(y),r2(x)*/
```

Suppose initially the program is executed along the program order: `1-2-3-4`. The state of the program is $r1(y) = 0$ and $r2(x) = 1$. Next, to make $r(y) = 1$ (return the value of $w(y)$), we encode $O_3 < O_1$. Because there is no event that is control dependent on a read in this program, we do not consider the data-validity constraints. Then a feasible schedule generated by our constraints can be `2-4-3-1`, making $r1(y) = 2$ and $r2(x) = 2$ instead of $r1(y) = 1$. This is because our constraints only ensure the reachability of $w(y)$ and does not constrain the value returned by $r2(x)$, which has a data dependency on $w(y)$. Hence the value written to $w(y)$ can be any one returned by $r2(x)$.

When considering the reachability of a write w, we also need to ensure that w writes the same value to the shared address as it does in the original trace. To guarantee this, we force a read r to return the same value if r is a read access to the same address accessed by w and has a data dependency on w. Similar to δ^c, we denote the data dependency between two events as δ^d: given two nodes $n1$ and $n2$ in an SDG, we use $n1\ \delta^d\ n2$ to denote that $n2$ is data dependent on $n1$. Then we can derive the data dependency rule following the spirit of Rule 1. Given a write w and a read r, to check $r\ \delta^d\ w$ is equivalent to check that if there is a path p ending with a *data dependency* edge from r to w. We present the rule in Figure 4.

The reason why the path may contain several *DD* edges is that the dependency can be transmitted via the operations on local variables, similar to *Case 2* presented in Section 4.2.1.

4.2.3 Dependency Reads Computation

After the discussion about the *control* and *data dependency*, we now present the algorithm of the function **DependencyComputation()** in Algorithm 1 to give the details about how to compute the set of reads that an event is dependent on in the program.

Algorithm 2 takes as input a given event e and the set of the reads $\prec_\tau (e)$, containing all the reads in τ that must-happen-before e. The algorithm analyzes two situations. If event e is a read, it only chooses the reads from $\prec_\tau (e)$ that e is control dependent on and adds them to the set $\prec_\tau^D (e)$. If e is a write, the algorithm adds the reads from $\prec_\tau (e)$ that e is control or data dependent on to $\prec_\tau^D (e)$.

4.3 Discussion

Challenges of static analysis for object-oriented languages, such as Java, stem from object- and filed- sensitivity, dynamic dispatch and objects as parameters problems and so on. These statically undecided problems are usually approximated relying on points-to analysis, or pointer analysis. However, it is difficult to make precise points-to analysis, and even the

Algorithm 2: Computation of $\prec_\tau^D (e)$

```
1  Function DependencyComputation(≺_τ (e), e):
2      ≺_τ^D (e) = ∅ ;
3      foreach read r in ≺_τ (e) do
4          if e is a read then
5              if r δ^c e then
6                  add r to ≺_τ^D (e) ;
7          else
8              if r δ^c e or r δ^d e then
9                  add r to ≺_τ^D (e) ;
10         return ≺_τ^D (e) ;
```

precise points-to analysis has to approximate certain undecidable situations which lead to may-alias. Due to the limitations of all static analysis, it is difficult for us to build fully precise SDGs so that an SDG may contain false or approximated dependency information. However, the soundness of our approach is not threatened by the unsound dependency. In this section, we use two cases to explain why our approach is not affected by imprecise static analysis.

Case 1: Problem with may-alias

Imprecise points-to analysis may lead to the may-alias problem between two pointers of the same type. In the construction of the SDG, the may-alias problem may lead to that a later read is data dependent on several writes to the same memory location. Let us consider the following example:

```
1: p.o = 1;          //w1
2: q.o = 2;          //w2
3: if (p.o == 1);    //r
```

where p and q are pointers of the same type and o is the field that p and q can access. When we construct the SDG for the program above, both $w1$ and $w2$ have a data dependency on r (*i.e.*, $(w1, w2)\ \delta^d\ r$) because p and q may alias. However, this does not affect our algorithm to decide which write that r is exactly data dependent on. This is because when the program is executed and generates the trace $e_1 - e_2 - e_3$, our algorithm is aware of the field information accessed by each event. From the trace, we can identify exactly what event has a dependency on e_3.

Case 2: Problem with path-insensitivity

Because the generated SDG considers all the possible paths of the program, the dependency read set $\prec^D$ computed from the SDG contains reads in all the paths, which leads to imprecise dependency. Consider the following program as an example.

```
1: if (exp)   r = x;  //r1
2: else       r = x;  //r2
3: y = r;             //w
```

If we use the SDG to compute the read set that write $y = r$ is data dependent on, both of the reads $r1$ and $r2$ have a data dependency on $y = r$ (*i.e.*, $(r1, r2)\ \delta^d\ w$) because the

SDG is path-insensitive. But this can be avoided by our approach because we combine static analysis with the dynamic information. Our algorithm for computing $\prec_\tau^D (e)$ is based on a concrete executed trace, *i.e.,* only $e_1 - e_3$ or $e_2 - e_3$ can be generated. As a result, only one read, either $r1$ or $r2$ has data dependency on w in an concrete execution.

5 Redundant Executions

Extending MCR with static dependency analysis reduces the size of the constraints for exploring new program's states, and it will not miss any executions. However, our approach may explore redundant executions. In this section, we use a simple example to illustrate how the redundant executions are introduced and explain the root reason that causes the redundancy. We also propose a solution to the redundancy problem.

```
          initially x = 0;
thread 1:                thread 2:
1: x = 1; /*w(x)*/       2: r1 = x;   /*r1(x)*/
                         3: r2 = x;   /*r2(x)*/
```

Listing 2 An example that shows redundant explorations by our approach.

Consider the example above. Following the procedure in Section 2, MCR generates only three different executions to explore the state space of this program.

- $\tau_0 =< e_1, e_2, e_3 >, (r1 = 1, r2 = 1)$;
- $\tau_1 =< e_2, e_1, e_3 >, (r1 = 0, r2 = 1)$;
- $\tau_2 =< e_2, e_3, e_1 >, (r1 = 0, r2 = 0)$.

However, using static dependency analysis, our approach generates one more execution $\tau_1' =< e_2, e_3, e_1 > (r1 = 0, r2 = 0)$, which is equivalent to τ_2. We explain how the same state is explored twice as follows.

First, the program is executed in the program order and the execution $\tau_0 =< e_1, e_2, e_3 > (r1 = 1, r2 = 1)$ is generated. Then the two read events in the trace, $r1(x)$ and $r2(x)$, will be considered to return a different value. To make $r1(x)$ return a different value 0, $r1(x)$ should read from the initial write. Then e_2 is required to happen before the write e_1 and thus we generate a new execution $\tau_1 =< e_2, e_1, e_3 > (r1 = 0, r2 = 1)$. Then the analysis on τ_0 is done because $r2(x)$ cannot read from the initial write if we use MCR to model check the program. The reason is that when considering the second read $r2(x)$ in τ_0, MCR enforces that $r1(x) = 1$ because $r1(x)$ happens before $r2(x)$ according to the data validity constraints. This implies that $r1(x)$ should read from $w(x)$ so that $e1$ should happen before e_2. As e_2 happens before e_3 by the program order, then e_1 happens before e_3 because of the transitive relation. Therefore $r2(x)$ is only able to read from $w(x)$ from the analysis on τ_0. But by our approach, we assume that $r(1)$ does not affect the reachability of $r2(x)$. As a consequence, we do not enforce $r1(x) = 1$ when considering different values that $r2(x)$ can return. Then a new execution is allowed by our approach,

- $\tau_1' =< e_2, e_3, e_1 > (r1 = 0, r2 = 0)$.

This execution is equivalent to the state of τ_2. And τ_2 can be derived from τ_1. The root reason why MCR does not generate such a redundant execution is that enforcing the read to hold a value implicitly causes a happens-before order between the write and the read (*e.g.* $w(x)$ and $r1(x)$), thus indirectly affecting the value by a later reader (*e.g.* $r2(x)$). Now that we do not require those reads to hold the same value, the implicit happens before order imposed on some writes and reads that access the same memory locations and reside in different threads is removed.

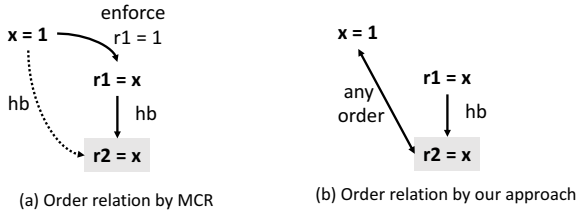

Figure 6 Removed happens-before between $x = 1$ and $r2 = x$ by our approach.

Figure 6 shows the difference of the order relation by MCR and our approach on the example above. The dashed arrow represents the implicit happens-before relation and the shadowed box represents the read we consider. As we can see in Figure 6(b), $x = 1$ and $r2 = x$ can be in any order by our approach, while $x = 1$ happens before $r2 = x$ in MCR.

5.1 Redundancy Elimination

According to the analysis on the example presented in Listing 2, we observe that when MCR explores the new values that a considered read r can return, enforcing all the reads that happen before r, on the one hand, guarantees the reachability of r and on the other hand, restricts the writes that r can read from. But for the rest of the reads and writes, we are only concerned about the reachability of them. We address the redundancy problem by adding constraints to make all the reads that happen before r return the same value. This is a trade-off between the original MCR and Algorithm 1. We present our algorithm as follows.

Algorithm 3: DataValidityConstraints′(τ, e)

Input : τ - a trace and e - a given event in τ
Output : $\Phi_{validity}(e)$ - data-validity constraints related to e

1 $\Phi_{validity} = \emptyset$
2 $\prec_\tau (e) \leftarrow$ **Happens-before**(τ, e)
`// target read: read considered to return new values`
3 **if** *e is not a* TARGET READ **then**
4 $\quad \prec^D_\tau (e) \leftarrow$ **DependencyComputation**$(\prec_\tau (e), e)$
5 **end**
6 **foreach** *read* $r \in \prec^D_\tau (e)$ *with value* v **do**
`// `$\Phi_{value}(r, v)$` recursively call DataValidityConstraints()`
7 $\quad \Phi_{validity} \wedge = \Phi_{value}(r, v)$
8 **end**
9 **return** $\Phi_{validity}$

The only difference between Algorithm 3 and Algorithm 1 lies in *line 3*. In our new algorithm, we decide whether to add the reads that happen before e to $\prec^D_\tau (e)$ based on the type of e. If e is a read expected to return a new value, we put all the reads that happen before e into $\prec^D_\tau (e)$ to avoid the redundant behavior. For the example, in Listing 2, as we want to explore what values $r2(x)$ can read, we also put $r1(x)$ into $\prec^D_\tau (e)$ to make $r1(x)$ return the same value as that in τ_0 so that τ'_1 will not be generated by our approach. If e is an event that we only care about if it will be reached in the next schedule, we handle e in the way of Algorithm 1. Although this expands $\prec^D_\tau (e)$ and increases the size of the constraints, it still generates less constraints than MCR does but with no redundancy. Moreover, if the solving of the constraints takes much more time than what the execution of the program needs, we can keep the redundant executions to reduce the overall checking time. We will have more discussions about this in Section 6.

Algorithm 3 can remove all the redundancies caused by Algorithm 1, and it will not miss any executions.

Proof. The proof on the latter part follows the same analysis on Algorithm 1 in Section 4.1. To prove that Algorithm 3 reduces all the redundancies, we show that by using Algorithm 3, our approach explores the same executions as MCR does. Given a trace τ, MCR considers only one read $r \in \tau$ each time when exploring new schedules. Consequently, the number of the new executions derived from r depends on the number of the writes that r can read from in τ. Because we force all the reads that happen before r to return the same value as that in τ, which remains completely the same as how MCR handles such a read, r reads from the same writes as that it can read from in MCR. Therefore, our approach explores the same executions as MCR does. ◀

6 Implementation and Evaluation

This section presents the implementation of integrating static dependency analysis into MCR and evaluates the performance improved by using static analysis.

6.1 Implementation

SDG construction

The SDG of the program has been well studied for a long time and there are many framework that can compute SDG, such as *WALA* [2] and *Soot* [27] for Java programs. In this work, we build the SDG of Java programs based on two existing framework, *JOANA* [1, 14] and *WALA*. *JOANA* is a information flow tool based on *WALA* for Java programs. *JOANA* implements flow-sensitive, context-sensitive and object-sensitive analysis and it minimizes false alarms. Considering that *JOANA* supports full Java bytecode and refines the SDG by *WALA*, we choose *JOANA* as our framework to construct the SDG.

Path Finding

Before the dynamic analysis on the executed trace, we first generates the SDG of the program and use a map structure to store the information of the graph. Because the SDG of a large system contains thousands of nodes, we use a distinct integer ID to represent each node to save the memory space of the map. During the dynamic exploration, we match the event in the trace with its corresponding node in SDG, and decide the dependency relation of two events by checking whether the path (if it exists) between the two nodes matches the rule defined in Figure 4 or 5.

6.2 Methodology

In the rest of this section, we refer to as MCR-S and MCR-S+ the approach that implements Algorithm 1 and 3, respectively. We evaluate the effectiveness of MCR-S and MCR-S+ by testing the three approaches on various benchmarks, including two large Java programs. Our evaluation aims to answer the following three research questions:
RQ1: How many reads and constraints can be reduced by our approach, compared to MCR?
RQ2: To what extent can the solving time be improved after the constraints are reduced, compared to MCR?
RQ3: How does the redundancy by MCR-S affect the total time spent on the state-space exploration?

Table 1 Benchmarks.

Program	time(s)	memory(M)	#nodes	#edges
Counter	2.00	69	289	1,440
Airline	2.10	79	809	4,902
Pingpong	2.52	83	914	5,244
BubbleSort	2.14	81	911	5,710
Pool	3.67	75	2,848	17,586
StringBuf	2.96	111	2,129	12,310
Weblech	8.01	219	22,094	167,492
Derby	69.67	1,385	115,658	2,409,784

In Section 6.3, we address RQ1 by comparing MCR-S and MCR-S+ with MCR, with respect to the number of the reads, constraints and the solving time. In Section 6.4, we consider RQ3 via evaluating the total time spent in exploring the state space of the program by the three approaches. We expect to see how the overall performance is improved by the static analysis and meanwhile the influence by the redundant executions. The comparison between MCR-S and MCR-S+ reveals which improves the performance more, the maximal constraints reduction with redundant executions or the partial constraints reduction with no redundancy.

The experiments were run on a MacBook with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory and JDK 1.7. All results were averaged over three runs.

Benchmarks

To show the effectiveness improved by our hybrid analysis, we run our approach on the same benchmark set used by prior work [16] so that we can make a direct comparison. Table 1 summarizes the benchmarks evaluated in this work. **Counter** is the example introduced in Section 1, and we take $Max = 5$ during the evaluation. **Airline** is a program that can sell more tickets than the capacity. **Pingpong** can arouse an NPE error on the shared variable player. **BubbleSort** is a small but read-write intense program with more than 10 million interleavings. **Pool** contains a concurrency bug in Apache Commons Pool causing more instances than allowed in the pool. **StringBuf** contains an atomicity violation. **Weblech** and **Derby** are two large real-world programs with long trace and complicated constraints. We present the time and memory used to construct the program's SDG in the second and third column, respectively. The last two columns show the number of the nodes and edges in the graph generated.

6.3 Reduction Analysis

Table 2 reports the results by MCR, MCR-S and MCR-S+ on the benchmarks. Column *#reads* lists the number of the reads the three approaches considered totally when constructing constraints to explore new interleavings. Column *#constraints* gives the total number of data-validity ($\Phi_{validity}$) constraints that map a read to a certain write. The number is the sum of the constraints generated by each exploration in the whole state-space search. As the other constraints remain the same for MCR and the new approaches, we just discuss the read-write constraints in the evaluation. Column *time* shows the time used by the solver to solve the constraints.

Figure 7 presents the reduction results by MCR-S and MCR-S+ compared to MCR on the number of the reads and constraints as well as the solving time. The figure is best viewed in color. The blue bar represents the results by MCR, green for MCR-S and yellow for

Table 2 Results of the number of the reads and constraints as well as solving time generated by MCR, MCR-S and MCR-S+ to explore the state-space of the benchmarks, respectively. one hour.

Program	MCR			MCR-S			MCR-S		
	#reads	#consts	time(*sec*)	#reads	#consts	time(*sec*)	#reads	#consts	time(*sec*)
Counter	55,886	202,039	22.11	37,515	108,270	7.41	45,972	131,053	12.25
Airline	15,632	24,643	2.43	15,328	24,475	2.39	15,599	24,625	2.38
Pingpong	1,905	5,225	1.42	1,376	3,684	1.38	1,906	5,227	1.32
BubbleSort	5,583,561	3,487,802	679.27	3,574,528	2,158,422	546.75	5,087,528	3,046,852	586.42
Pool*	143	68	< 1	94	12	< 1	117	36	< 1
StringBuf*	102	30	< 1	102	30	< 1	102	30	< 1
Weblech	120,161	5,676	13.75	103,155	3,920	6.39	90,096	4,217	5.24
Derby	46,222,858	22,008,512	477.13	22,530,501	12,184,850	347.98	36,461,542	17,412,201	300.58
Avg.	8,666,667	4,288,982	199.35	4,377,067	2,413,936	151.03	6,950,440	3,437,362	151.26

* The exploration time on these two benchmarks is far less than 1 *second* and we ignore them when we compute the average results.

MCR-S+, respectively. For comparison, we normalize MCR's results to 1 as the baseline and length of the green and yellow bars represents the ratio of the results of MCR-S and MCR-S+ to that of MCR.

Number of reads reduced.

Figure 7(a) summarizes the comparison on the number of the reads reduced by MCR and our approaches. Averagely, MCR-S reduces the number of the reads by 27.1% and MCR-S+ by 12.1% compared to MCR. And the reduction percentage by MCR-S ranges from 14.2% to 51.3%, and MCR-S makes the greatest reduction on the `Derby` benchmark. Comparing to MCR-S, MCR-S+ makes less reduction because it needs to constrain more reads into the formula to avoid the redundant executions (Section 5). But MCR-S+ still makes a reduction that ranges from 8.9% to 25.0% compared to MCR. Among the 6 benchmarks, neither MCR-S or MCR-S+ makes a reduction on `Airline`. The reason is that in the routine `run()` of `Airline`, all the reads and writes are control dependent on a read in the `if` predicate. As introduced in Section 4, we can't reduce any reads for this benchmark. In addition to `Airline`, the other benchmark that MCR-S+ fails to reduce the reads is `Pingpong`, while MCR-S reduces the reads by 28.2%. Note that for benchmark `Weblech`, MCR-S considers more reads than MCR-S+ does. This is because that MCR-S explores more executions than MCR-S+ does due to the redundancy, and we take as the final result the total number of reads the approaches have considered in the whole state-space exploration.

Number of constraints reduced.

Figure 7(b) reports the reduction of the data validity constraints by MCR-S and MCR-S+. As the reads are reduced by our approaches, we do not need to constrain those reads to return the same value, and thus reduce the size of the constraints. Given a read r that returns the value by the write w, we count the constraint as one, and the constraint enforces another write that writes a different value from that by w to the same location to either occur before w or after r. On average, MCR-S reduces the number of constraints by 31.6%, while MCR-S+ by 15.7%. As Figure 7(b) shows, the reduction on the constraints is consistent with that on the reads in Figure 7(a).

Solving time reduced.

Figure 7(c) presents the results of the solving time by each method. From Figure 7(b) and (c), we can see that though MCR-S approximately makes two times as much constraints reduction

Figure 7 Reduction on the number of the reads and constraints as well as the solving time achieved by MCR-S and MCR-S+ comparing to MCR. The results generated by MCR are normalized to one as the baseline.

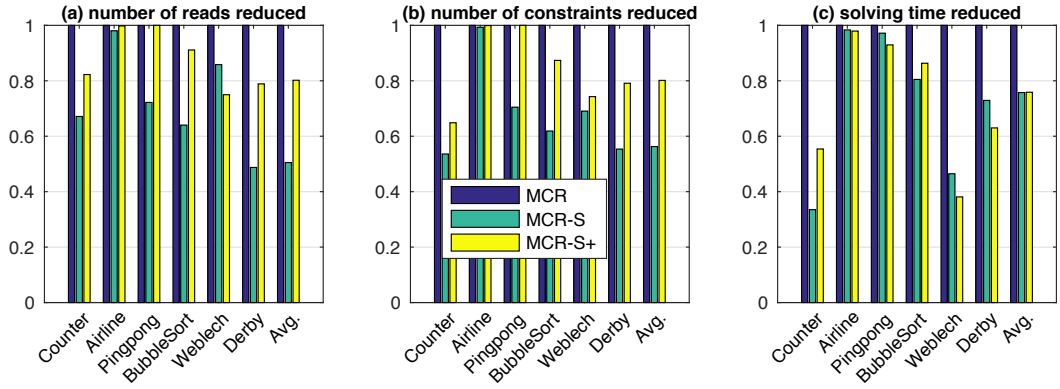

Table 3 The total number of executions and time taken by the three methods to explore the state-space of the benchmarks.

Program	MCR		MCR-S		MCR-S+	
	#executions	time(*sec*)	#executions	time(*sec*)	#executions	time(*sec*)
Counter	4,523	181	6,550	247	3,485	133
Airline	14	4	14	5	14	5
Pingpong	394	13	535	16	394	15
BubbleSort	5,823	OOT	1,828	OOT	6,885	OOT
Weblech	967	677	756	511	668	385
Derby	15	787	16	797	15	676

as MCR-S+ does, the solving time taken by the two approaches is quite close to each other. Among the 6 benchmarks, MCR-S reduces the solving time by 27.8% compared to MCR, on average, while 26.2% by MCR-S+. Moreover, for benchmarks `Weblech` and `Derby`, it takes more time for MCR-S to solve the constraints than MCR-S+. This is because MCR-S explores more executions than MCR-S+ does, and thus the size of the total constraints generated by MCR-S actually is greater than that by MCR-S+. Likewise, though MCR-S reduces the size of constraints by 29.5% on the benchmark `Airline`, it takes almost the same time for MCR-S to solve the constraints as that for MCR.

6.4 Overall Checking Performance Comparison

Table 3 summarizes the state-space exploration results by the three approaches, in terms of the number of executions explored and time (*seconds*) taken to finish the exploration. Note that we do not report the results of `Pool` and `StringBuf` because the execution time for these two benchmarks is too small to be tracked. We run `BubbleSort` with an input which contains four integers. Because `BubbleSort` is a read and write intensive benchmark, none of three methods can finish the exploration in a reasonable time. Therefore, we set one hour as an upper bound for the exploration and use OOT to represent that the exploration runs out of time. As discussed in Section 5, MCR-S may introduce some redundant executions into the exploration. Consider the `Counter` and `Pingpong` benchmarks. It takes $6,550$ and 535 executions for MCR-S to explore the state-space, respectively. But it only takes $4,553$ and 394 executions for MCR and $3,485$ and 394 for MCR-S+. Although MCR-S reduces

more reads and constraints than MCR-S+ does, it also introduces redundant executions. As a result, it takes more time for MCR-S to check the two benchmarks. But MCR-S+ reduces the total time of the exploration of `Counter` by 48 seconds, compared to MCR. For the `BubbleSort` benchmark, all of the three methods fail to finish the exploration in one hour. MCR-S+ explores the most executions while MCR-S explores the least among the three methods in the bounded time, meaning that the average time of MCR-S+ spent on each execution is the least. MCR-S+ fails to reduce the total exploration time on `Pingpong` and `Airline` for two reasons: (1)First, the two benchmarks generates light constraints and the solving time of the constraints only takes a small portion of the total time. (2) Second, it takes time for MCR-S+ to check the dependency between two events in the dynamic exploration.

For the benchmark `Weblech`, both MCR-S and MCR-S+ reduce the exploration time by about 3 and 5 minutes, respectively. Although MCR-S and MCR-S+ explores less executions on `Weblech`, interestingly, all of the three methods expose the `null pointer exception` in the benchmark. For `Derby`, MCR-S+ reduces the checking time by about 2 minutes, compared to MCR and MCR-S, and MCR-S spent 10 more seconds than MCR does. Among the six benchmarks, MCR-S+ achieves the best effect. This is because MCR-S+ reduces the size of the constraints, and meanwhile it does not introduce any redundant executions.

7 Related Work

Stateless Model Checking

SMC is a powerful systematic testing technique that can verify the correctness of concurrent programs by automatically exploring all the possible interleavings by the program. SMC prevails since the pioneering work of VeriSoft [11]. To mitigate the state explosion problem, a great effort has been dedicated to reduction techniques to prune the equivalent executions from the state space. The most popular techniques known are Partial Order Reduction (POR) [7, 10] and context bounding [24, 23], while context bounding does not reduce redundancy but limits the search space to polynomial. A number of techniques [8, 23, 4] based on POR or combining them have been proposed to improve and optimize the performance of POR. However, as pointed out in the MCR work [16], the effectiveness of POR is limited by *happens-before*: it can not reduce the redundant interleavings that have different *happens-before* relations.

MCR [16] is new reduction technique to explore new program states by using SMT or SAT solvers to search new interleavings. The new interleaving is produced by solving the constrains over the order variables of the events. As discussed before, the size of the constraints can be arbitrarily large and complicated, in general cubic in the size of the trace. Huang *et al.* [20] recently extended MCR from SC [21] to TSO and PSO [5, 26]. Our work can also be applied to optimize the constraints in this technique.

Program Slicing

Our work is closely related to program slicing technique, originally defined in [28], which aims to compute a slice consisting of all statements and predicates that can influence the value of a certain point in the program. Ottenstein et al. [25] brought *program dependence graph* (PDG) into slicing and pointed out that PDG is well-suited for representing the procedures in software development environment. Horwitz et al. [15] addressed interprocedural-slicing problem by introducing the *system dependence graph* (SDG) to represent the whole program.

To find the statements that influence the value of a point under specific input instead of all inputs, Agrawal and Horgan [6] proposed the notion of the dynamic slicing based on a dynamic dependence graph to narrow the slice.

Different from the above techniques, our work is only interested in the reads which influence the evaluation of a predicate, and thus influencing the reachability of a certain point. Moreover, our slice is based on the executed trace. As a result, although the dependence graph is statically computed, we only include the statements that do affect the occurrence of a specific event because all the statements are from the executed trace.

Other Works

Another work that our approach shares partial similarities with is TAME [18] by Huang and Rauchwerger. TAME tries to find what branches in the given trace have the chance to explore a different path due to the program's schedule. It is feasible to combine our work with TAME. We can first run TAME on the trace to exclude those branches that will not take a different path no matter how the program schedules and then only consider reads that relate to schedule-sensitive branches.

Cortex [22] is an extension on CLAP [19] that helps expose and understand schedule- and path-dependent concurrency bugs. Cortex is able to synthesize failure executions from correct production runs by flipping branches and alternating the order of concurrent events. It leverages symbolic execution to identify the path conditions and inverts the path condition to synthesize a different control flow. Our approach can also first instrument those reads related to path conditions and record them in the trace. Then we can directly identify those reads when we construct constraints over the trace.

8 Conclusion

In this work, we present a new technique to reduce the size of the constraints formula to speed up MCR via static dependency analysis. We use system dependency graph to capture the dependency between a read and an event e in the trace and exclude those reads that e is not control dependent on. We then can ignore the constraints over such reads to make them return the same value and thus reducing the complexity of the formula. The experimental results show that comparing to MCR, the number of the constraints and the solving time by our approach are averagely reduced by 31.6% and 27.8%, respectively.

Acknowledgements. We would like to thank our shepherd, Anders Møller, and the anonymous reviewers for their valuable feedback.

References

1 Joana: Information flow control framework for java. `http://pp.ipd.kit.edu/projects/joana/`.

2 Wala. `https://github.com/wala/WALA`.

3 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2014.

4 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

5 Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

6 Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI, 1990.

7 Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

8 Katherine E. Coons, Madanlal Musuvathi, and Kathryn S. Mckinley. Bounded partial-order reduction. In *In Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 833–848, 2013.

9 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

10 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

11 Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.

12 Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 2005.

13 Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

14 Jurgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM, 2010.

15 S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI, 1988.

16 Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

17 Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

18 Jeff Huang and Lawrence Rauchwerger. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2015.

19 Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

20 Shiyou Huang and Jeff Huang. Maximal causality reduction for tso and pso. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2016.

21 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

22 Nuno Machado, Brandon Lucia, and Luís Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

23 Madanlal Musuvathi and Shaz Qadeer. Partial-order reduction for context-bounded state exploration. Technical report, Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

24 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008.

25 Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, 1984.

26 Scott Owens, Susmit Sarkar, Peter Sewell, and A Better. x86 memory model: x86-tso. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

27 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON, 1999.

28 Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE, 1981.