FAST AND EFFECTIVE APPROACHES FOR VERIFYING AND DEBUGGING CONCURRENT PROGRAMS

A Dissertation

by

SHIYOU HUANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jeff Huang
Committee Members,	Dilma Da Silva
	Lawrence Rauchwerger
	Paul Gratz
Head of Department,	Scott Schaefer

December 2019

Major Subject: Computer Science

ABSTRACT

Concurrent programs are ubiquitous, from the high-end servers to personal machines, due to the fact of multi-core hardware. Unfortunately, it is difficult to write correct concurrent programs. Stateless Model Checking (SMT) and Deterministic Replay are powerful techniques for systematic testing and reproducing concurrent failures. However, it is challenging to develop efficient and practical SMT and bug reproduction systems due to the exponentially large thread interleaving space which can be exacerbated when it comes to relaxed memory models. In this work, I introduce my research efforts to address the challenges in developing fast and effective SMT and deterministic replay techniques. I present a new model checking technique based on maximal causality reduction for verifying concurrent programs under different memory models. I also optimize the model checker by using static dependency analysis to reduce the constraints size and introducing a new equivalence for checking the seed interleavings, which I call *switch equivalence* to further reduce the redundant exploration.

To debug heisenbugs more efficiently, I presents a new concurrency failure reproduction technique, H3, that enables reproducing concurrency bugs in production runs on commercial off-the-shelf hardware for the first time. H3 integrates the hardware control flow tracing capability provided in recent Intel processors, Processor Tracing (PT), with symbolic constraint analysis. Compared to a state-of-the-art solution, CLAP, this integration allows H3 to reproduce failures with much lower runtime overhead and much more compact control-flow trace. Moreover, it allows us to develop a highly effective core-based constraint reduction technique that reduces the complexity of the generated symbolic constraints from exponential in the trace size to exponential in the number of cores.

ACKNOWLEDGMENTS

Looking back at the journey of my PhD life, I am full of gratitude. It is not a journey of myself, but a journey accompanied by many others, my advisor, labmates, roommates, friends and any other people who give me help and guidance during the past four and half years. Without them, this thesis could not be possible.

First and foremost, I want to thank my advisor Jeff Huang for giving me this opportunity to work with him and providing me tremendous help and guidance on my research. Working with Jeff is a life-time invaluable experience. I am always impressed and encouraged by his energy, optimism, persistence, diligence and eagerness for perfection. I feel quite lucky to work with him and I have learned a lot of things from him including writing, presentation and how to do research. Most importantly, he influences my way of thinking and appreciating good research works.

I would also like to express my gratitude to my committee members, Dilma Da Silva, Lawrence Rauchwerger and Paul Gratz, who spent time on my prelim and gave me the thoughtful feedback. I am especially thankful for Dilma's suggestions on my talk for the USENIX ATC conference and her reference letter for my internship.

I want to thank all the members in the ASER group, Gang Zhao, Bozhen Liu, Bowen Cai, Peiming Liu, Brad Swain, Yanze Li, Qiuping Yi, Siwei Cui, Yahui Sun, Luochao Wang and Shanshan Li. I really value the thoughtful discussion with them and group activities we have been through. I will miss the delicious food cooked by Gang and Yanze, the fun brought by Peiming, and all the other beautiful moments. I want to thank Brad for patiently answering my questions on the American culture and standard English. You guys make my PhD life not dull and hard at all. The relationship with you is a great fortune to me.

I want to thank my roommates, Wenlin Yao, Xilong Zhou and Zhongcheng Zhang. I cannot believe that I have been roommate with Wenlin for four years. Four-year is a long time and it is almost my PhD career, but I still clearly remember the first time we met each other. We often eat and play together. I really appreciate their support when I am through the hardships.

Finally, I want to thank my family for their unselfish support and understanding. I would thank my girlfriend Yi Chen for not showing up until the last year of PhD life so that I can focus on my research in the first three years. I will keep moving forward to not let you down.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Jeff Huang (advisor), Dilma Da Silva and Lawrence Rauchwerger of the Department of Computer Science and Professor Paul Gratz of the Department of Electric Engineering.

All work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by either Graduate Research Assistant or Graduate Teaching Assistant from the Department of Computer Science, Texas A&M University

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The multi-core hardware is not new to us anymore these days. Not only in the high-end servers, they also appear in our personal computers and mobile devices. In order to make full use of the multi-core CPUs, many softwares are developed using concurrency for higher performance, from our daily used browsers, database systems, to big data platforms. Unfortunately, it is difficult to write correct concurrent programs. Moreover, it is extremely hard to test and debug them. Concurrency errors such as data races, atomicity violations, non-deterministic ordering and deadlocks threaten the correctness and stability of our software and can cause severe problems such as huge economy lost [3, 5] and even real-world disaster [6]. The reasons why it is more challenging to reason about concurrent programs than sequential programs are as follows.

Concurrent Programs Are Difficult to Verify. Concurrent programs can generate a huge thread interleaving space, while the bug may only manifest in a certain interleaving. It is impossible to consider every interleaving to verify the correctness of a concurrent program. This problem is exacerbated when it comes to relaxed memory models. It is known that *sequential consistency* (SC) [7] is the most intuitive memory model, under which operations by different threads can interleave but those by the same thread should always follow the program order. It is challenging enough to verify concurrent programs under SC, because the number of different interleavings grows exponentially with the number of threads and the length of program execution. To make matters worse, most contemporary multiprocessors implement *relaxed memory models* (RMMs), such as *Total Store Order* (TSO) and *Partial Store Order* (PSO) [8, 9] to achieve better performance. For TSO and PSO, the verification problem is much more challenging because operations by the same thread may no longer follow the program order. For instance, under TSO, a write and a following read by the same thread can be re-ordered if they access different memory locations, and under PSO, which is a further relaxation of TSO, two writes by the same thread can be re-ordered if they target different locations.

The ability to re-order operations from the same thread under TSO and PSO significantly explodes the state-space over SC. Consider M concurrent threads each executing N_i operations where i=$1, 2, \ldots, M$. The total number of interleavings under SC can be calculated by the formula $\prod_{i=1}^{M} \binom{\sum_{j=i}^{M} N_j}{N_i}$ [10], and that of allowing the reordering of operations can be calculated by the formula $(\prod_{i=1}^{M} N_i)!$, *i.e.*, the number of permutations of all operations. Consider only four threads and four operations each (M=N_i=4). The number of interleavings under SC is $6 * 10^7$, whereas the number of permutations is $2 * 10^{13}$, which is 300,000 times larger.

Concurrency Bugs Are Difficult to Reproduce The behavior of a concurrent program is non-deterministic as it can be impacted by the user input, how the thread interleaves, asynchronous events and so on. Due to the non-determinism, concurrency bugs may disappear when re-executing the program. Moreover, the program can exhibit more behaviors under relaxed memory models. All these non-determinism makes the debugging of concurrent programs extremely difficult. As a result, a technique that can faithfully reproduce the concurrency failure is of great significance to the developers. However, most existing solutions either are too slow due to the high runtime overhead incurred by tracing the shared memory dependencies, introduce the observer effect that makes the Heisenbugs disappear [11, 12, 13], or require special hardware that does not exist [14, 15, 16, 17, 18].

1.1 Contributions

This dissertation presents four contributions to combat the challenges of developing more efficient and practical techniques to verify and debug concurrent programs: (1) a stateless model checker for concurrent programs under relaxed memory models with maximal causality reduction (MCR); (2) MCR-S: a work that uses static dependency analysis to reduce the complexity of the constraints built by MCR; (3) SE-MCR: a work that reduces the state space of the concurrent program by checking equivalent interleavings so that it makes the model checker more effective; and (4) H3 a *record and replay* system that combines hardware flow tracing with offline constraints analysis to reproduce concurrency failures.

1.1.1 Maximal Causality Reduction for TSO and PSO

This dissertation presents an effective model checker for programs under two types of relaxed memory models TSO and PSO. This work builds first-order constraints over an executed trace to reason about the other possible interleavings that can be derived from the given trace. To support TSO and PSO, it address two key problems: (1) How to soundly encode the semantics of TSO and PSO (specifically the write-to-read and write-to-write reorderings) by relaxing the SC constraints developed by Huang [19]? (2) How to deterministically replay TSO and PSO interleavings for concurrent programs?

This work is built based on a prior work Stateless Model Checking with Maximal causality Reduction (MCR) [19]. From a high-level perspective, this work uses constraints to reason about the semantics of TSO and PSO. Under *sequential consistency* (SC) memory model, all the read and write operations are executed following the program order. Different than SC, we build constraints to allow read and write events to be re-ordered while respecting the semantics of the memory models with store buffers (FIFO queues) under TSO and PSO. We assign one buffer to each thread for TSO and multiple for PSO (each corresponds to a dynamic memory location) to achieve the reordering. By invoking an off-the-shelf SMT solver to solve the constraints, new interleavings are generated and are used to replay the program to explore new states. We further design a novel algorithm to deterministically replay concurrent programs under TSO and PSO, where operations in the generated interleavings can be re-ordered (*i.e.*, does not follow the program order). The key insight of the algorithm is to decide when to buffer a write and when to flush it into the main memory by comparing the memory location of the executed operation with that of the operation given in the generated interleaving. This work not only supports TSO and PSO, but can be easily to be extended to support more relaxed memory models as long as the semantics can be encoded into the SMT constraints.

1.1.2 MCR-S: Speeding Up MCR with Static Dependency Analysis

Though MCR is powerful for verifying the concurrent programs, it it limited the fact that it is pure 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 as that in the current trace so that e is reachable in the derived interleaving.

In light of this limitation, this dissertation presents MCR-S to optimize the constraints constructed by MCR. The essential idea of this work is to provide the static dependency information between the events for the dynamic exploration. 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. In the exploration of new schedules from a given trace, we rely on the dependency information to decide what reads can influence the reachability of a later event, thus reducing the constraints that make those reads return the same value.

1.1.3 SE-MCR: Reduce the State Space of MCR with Switch Equivalence

MCR reduces the number of explored interleavings by DPOR and *iterative context bounding* (ICB) [20] 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). Although MCR gains a great performance improvement over the POR based approaches, it does not achieve the minimum number of explored interleavings.

This dissertation presents a new algorithm, *SE-MCR*, applying the *switch equivalence* checking to the MCR approach. The new algorithm contains two steps. After collecting the trace by executing the program along a given prefix, (1) it first computes all the seed interleavings derived from the given trace to drive the program to new states; then (2) it checks if two interleavings can result in equivalent executions in the future. If so, the algorithm makes one prefix (say $\mathcal{P}$) *remember* the other (say $\mathcal{P}'$). When we compute a new seed interleaving, suppose $\mathcal{P}''$ from the trace that begins with $\mathcal{P}$, we ignore $\mathcal{P}''$ if $\mathcal{P}''$ equals to $\mathcal{P}'$. We formally prove that *SE-MCR*does not miss any states

that MCR can produce and also avoids the redundant executions by MCR.

1.1.4 H3: Record and Replay on Commercial Hardware

In addition to verifying concurrent programs, which is sometimes limited to the scalability issue, another effective approach to fixing concurrency bugs is to improve the efficiency of debugging the concurrent program. However, as aforementioned, the bug may disappear when re-executing the concurrent program due to the non-deterministic memory races. Therefore, the ability to reproduce software bugs is crucial for debugging.

Researchers have investigated significant efforts in *record & replay* (RnR) systems aiming to eliminate the non-determinism. CLAP [21] is the most efficient software-based appraoch. It introduces the idea of recording only thread-local information (*i.e.*, thread-local *control flow* paths) and then using offline constraint solving to reconstruct the shared memory dependencies. It is a promising solution for reproducing Heisenbugs because it does not record any cross-thread communication (data or synchronization); hence it requires no synchronizations during recording, which not only reduces the runtime overhead but also minimizes the observer effect.

To enable a production-run RnR solution, however, CLAP is still unsatisfactory due to two important challenges. First, although CLAP is much faster than conventional solutions, the runtime overhead incurred by CLAP using software path-recording is as large as 3X, which is unacceptable for most production environments. Second, the constraints generated by CLAP can be too complex to solve. In the worst case, the complexity of the constraints is exponential in the trace size. Despite that SMT solvers (e.g., Z3 [22]) are becoming increasingly powerful, in practice, the constraints can become too large to solve in a reasonable time.

This dissertation presents H3, a new *record & replay* (RnR) system to reproduce Heisenbugs using commercial hardware features and offline constraints analysis. Our key observation is that both of the aforementioned challenges can be effectively addressed by hardware-supported *control-flow* tracing. Moreover, hardware-supported tracing allows us to perform a significant reduction of the constraints generated by CLAP because memory accesses executed on each core are ordered internally. We develop a core-based constraint reduction technique that reduces the complexity of

the constraints from exponential in the trace size to only *exponential in the number of cores*.

1.2 Roadmap

The remainder of this dissertation is organized as follows. Chapter 2 introduces the background knowledge and the prior work on stateless model checking and *record* and *replay* systems. Chapter 3 presents our stateless model checker for concurrent programs under two kinds of relaxed memory models. Chapter 4 addresses the scalability issue of *maximal causality reduction*, which generates too complicated constraints to be solved for existing SMT solvers. It uses static dependency analysis to provide the dependency relation for the constraints analysis. Chapter 5 reduces the state space of *maximal causality reduction* by introducing a coarser equivalence. Chapter 6 presents a new *record* and *replay* technique, H3, which is achieved by combining commercial hardware feature and off-line constraints analysis to reduce the runtime overhead of existing techniques. Last, Chapter 7 concludes the thesis and discusses the future work.

2. BACKGROUND AND RELATED WORK

With multi-core architecture prevailing, programmers develop concurrent softwares to fully take advantage of the parallel computing power. However, it is difficult to guarantee the correctness of the concurrent software because of all the possible thread interleavings. One direction to address this issue is stateless model checking, an automatic verification technique that explores all the possible interleaving of a concurrent program. As the state space explodes when the size of the program grows, it is difficult to consider every interleaving of a concurrent program. Consequently, stateless model checkers are usually wrapped with an efficient reduction technique which removes the redundant interleavings from the state space to make the model checkers more efficient and scalable.

In addition to verification, the ability to reproduce the concurrence failure is also important to developers. Record and Replay (RnR) systems are useful techniques for debugging concurrency bugs. As the execution of a concurrent program is non-deterministic, RnR records the non-determinism, *e.g.*, the ordering of the memory accesses, and then replays failure execution so that the developer can analyze the root cause of the failure. Section 2.1 introduces the existing stateless model checkers with the reduction techniques behind the model checkers. Section 2.2 presents the current RnR systems from software/hardware based approaches.

2.1 Stateless Model Checking

Stateless model checking is a technique that systematically explores all the possible thread interleavings of concurrent programs. Due to the state space explosion problem, a great effort has been dedicated to reduction techniques to prune the equivalent executions from the state space. The two most popular techniques are Partial Order Reduction [23, 24] and context bounding. POR computes an equivalent class of interleavings by checking if one interleaving can be transferred to another by swapping adjacent, non-conflicting events. POR executes only one interleaving from such an equivalent class to avoid redundant executions. Context bounding, like VeriSoft

[25], prioritizes executions with fewer context switches such that it can limit the state space to be polynomial.

2.1.1 POR Based Model Checking

Dynamic Partial Order Reduction (DPOR) [26] computes the persistent sets on the fly more accurately than the static POR does. Although DPOR improves the performance, the efficiency of its reduction is heavily influenced by what process is chosen to execute at each point of the scheduling. *Optimal-DPOR* proposed by Abdulla *et al.* [27, 28] attempts to reduce the state space further by proposing a novel class of sets, called *source sets*, which are smaller than persistent sets and provably minimal. The *source sets* contain the processes that can be executed at a point of scheduling. The difference between *source sets* and *persistent sets* is that *source sets* remove processes that are equivalent to some processes already in the sets. Although it is claimed that *Source sets* obtain an optimal reduction in the number of explored interleavings, it actually misses opportunities for reducing the redundancy when there exist two writes by different threads that write the same value. This approach will swap the order of the two writes to explore new states. *Data-Centric DPOR* (DC-DPOR) proposed by Chalupa *et al.* [29]. presents a coarser equivalence(called as *Observation Equivalence*) comparing to the Mazurkiewicz equivalence. Given a trace, the *Observation Equivalence* maps every read event to the write event it observes under sequentially consistent semantics. DC-DPOR considers two traces equivalent if they contain the same read events, and every read event observes the same write event in both traces. DC-DPOR shows that observation equivalence can be exponentially more succinct than Mazurkiewicz equivalence [27].

2.1.2 Constraints Based Model Checking

Several other constraints-based approaches have also been proposed for verifying concurrent programs, such as CheckFence [30], MemSat [31] and SATCheck [32]. Checkfence verifies concurrent data structures for relaxed memory models by explicitly encoding all relevent events into boolean formulas. It first compiles the code to thread-local sequences and then encodes the sequence of instructions and the memory model into boolean constraints. It uses a SAT solver to

search the erroneous executions. This approach is less automatic and it leaves it to the programmers to decide where to insert the fences when it finds bugs. Moreover, because of its static analysis, CheckFence lazily unroll the loops and it can miss bugs which only manifest in executions that exceed the loop bounds. MemSat is a similar technqiue to CheckFence. It verifies various weak memory models by specifying the memory model as a set of constraints in relational logic. It takes as input a memory model described by a set of constraints and a test with assertions encoded as constraints as well. By solving the constraints that encode both the memory specifications and the program assertions, MemSat is able to find subtle bugs in test programs that satisfy the constraints. SATCheck is another SAT based approach for checking concurrent programs. Different than CheckFence and MemSat, SATCheck is dynamic and builds constraints over a trace. Furthermore, instead of encoding a whole program to a SMT formula, SATCheck uses relative order to encode only a single concrete execution into a SAT formula. SATCheck constructs an event graph of each execution to capture the control-flow, memory operations, conditional branches, loops and so on. Then it encode the graph to a SMT formula and searches new interleavings by using a SMT solver to solve the formula. Each new interleaving will yield new behavior, e.g., executing a new branch. The working process of SATCheck is very similar to MCR. However, SATCheck requires the user to manually specify the `store` and `load` operations. Moreover, there is no guarantee that SATCheck can cover all the behaviors of a concurrent program.

Huang [19] proposed a newly efficient reduction technique called *Maximal Causality Reduction*, which overcomes such redundancy problem by taking the values of reads and writes in each trace into consideration. MCR regards the interleaving prefixes that enforce the same read to return the same value as an equivalent class and only executes one of them. MCR computes the execution sequence by constructing SMT constraints over the trace and leverages SMT solvers to solve the constraints. It seems that MCR explores a minimal set of interleavings. But it still can explore redundant executions if two adjacent execution sequences lead to the same state by swapping the order of the two sequences. Another limitation of MCR is that it can generate very complicated constraints which make it difficult for the solver to solve them in a reasonable time.

DC-DPOR shares the similarity with MCR on maping a read event to a write event. However, DC-DPOR can explore more executions than MCR as MCR considers the values of the read and write events as well. For example, if two writes write the same value to the same memory location, DC-DPOR would take two traces as inequivalent if the writes are executed in different ordering in the traces, while MCR regards them as equivalent.

2.1.3 Model Checking for Relaxed Memory Models

The feasibility of verifying concurrent programs under relaxed memory models have been studied before [33, 34, 35]. Abdulla et al. [36] apply SMC techniques to TSO and PSO by adopting a chronological trace presentation to relax the behavior of SC. Similar to [36], Zhang et al. [37] develop an approach that extends the original DPOR algorithm [26] to support TSO and PSO. The approach refines the dependent set to allow the reordering and introduces shadow threads to simulate the non-determinism of independent events by each thread. Both of the two approaches leverage DPOR to reduce the state space. However, since DPOR is limited by the *happens-before* relation, these approaches are less effective than MCR.

CDSChecker [38] checks C/C++11 programs using a variation of the classic DPOR algorithm. It exhaustively explores the behaviors of low-level concurrent structured under C/C++11. RCMC [39] is a model checker for verifying programs running under RC11 [40], a repaired version of the C/C++11 memory model without dependency cycles. This approach works directly on execution graphs instead of enumerating thread interleavings as other POR based approaches do.

2.2 Record and Replay Systems

Researchers have proposed many different RnR systems, both at the software level [41, 42, 11, 21, 43, 44, 45, 46, 47, 48, 49] and hardware-level [14, 15, 16, 17, 18]. Most RnR systems are either order-based [42, 11, 45, 48, 49] that rely on faithfully recording the shared memory dependencies at runtime, or search-based [41, 21, 43, 44, 47] that record only partial information at runtime and rely on powerful search engines such as SMT solvers to reconstruct the memory dependencies.

A central goal of RnR systems is to reduce the runtime overhead such that they can be used

in production runs. Hardware techniques [14, 15, 16, 17, 18] are often much more efficient than software-level implementation, but most previous RnR systems rely on special hardware that is not available. Intel PT is an exciting hardware feature that opens a door for RnR systems to be applied broadly in COTS platforms.

Gist [50] introduces a bug diagnosis technique that also leverages PT to identify root causes of a failure with low overhead. Different from H3, Gist assumes the failure can be reproduced in the first place, but it may fail to do so. In addition, Gist relies on statistical analysis to identify failure causes, but it has no guarantee, i.e., it may miss real causes or report false positives. Compared to Gist, H3 solves a different problem: reproducing failures before they can be diagnosed, and H3 is sound: it guarantees to reproduce the failure as long as the constraints can be solved by the solver.

Arulraj *et al.* [51] use hardware performance counters for failure diagnosis. This technique leverages the hardware to sample predicates from a large number of successful and failing runs and then use the sampled predicates to diagnose the failure via statistical analysis.

ReCBuLC [52] uses hardware clocks that are available on modern processors to help reproducing Heisenbugs. The recorded timestamps local to each thread together with a statistical analysis for calculating the time differences among local clocks across different cores, are used to determine the global schedule of shared-resource accesses. One limitation of this approach is that the statistical analysis may fail to infer a correct global schedule.

The idea of using offline constraint analysis to infer global failure schedules was pioneered by Lee *et al.* [43, 44]. The technique uses load-based checkpoints to search for a global schedule without recording any shared memory dependencies. However, compared to PT, the load-based checkpoints are not supported by the commodity architecture.

Similar to CLAP, both ODR [41] and Symbiosis [53] rely on symbolic constraint solving to figure out schedules that can satisfy certain conditions. ODR uses constraints to reproduce failures, and Symbiosis uses constraints for reducing the schedule complexity.

PRES [46] proposes a probabilistic replay technique that uses an intelligent feedback-based replayer to reproduce failures with lightweight recording. PRES may fail to reproduce the bug in

the first attempt due to a recorded incomplete schedule. However, it can learn from the previous failing replays to rectify the schedule. Typically after a few attempts, PRES is able to find a correct schedule to reproduce the bug.

Both CoreDump [47] and ESD [54] rely on only the program coredumps to diagnose failures. CoreDump uses a technique called execution indexing to compare the differences between coredumps from failing and normal runs to identify the failing point. ESD uses static analysis and symbolic execution to synthesize both program inputs and schedule to reproduce failures. Using coredumps is promising for diagnosing real-world failures since coredumps are often available after the program crash. However, since there is no program control flow information, the technique may be difficult to reproduce failures that require complex paths and schedules to manifest.

3. MAXIMAL CAUSALITY REDUCTION FOR TSO AND PSO *

Since the pioneering work of VeriSoft [55, 56] and CHESS [57], SMC has been successfully applied in real-world programs and has found many deep bugs. However, due to the state-explosion problem, it is difficult for a model checker to explore every possible interleaving of a concurrent program. A key challenge in SMC is how to reduce the state space of the program, thus making the model checking more efficiently and practical. Maximal Causality Reduction (MCR) is a reduction technique which computes the equivalent class of interleavings by matching a read to a different write. In this Chapter, I first introduce the background of MCR and then present a new technique based on MCR to model check concurrent programs under relaxed memory models.

3.1 Maximal Causality Reduction

In this section we first introduce the Maximal Causality Model and then the constraints built by MCR over the given trace and the properties can be checked with MCR. Section 3.1.4 introduces the workflow of MCR.

3.1.1 Maximal Causal Model

A fundamental concept underpinning MCR is the Maximal Causality Model (MCM) [58, 59], which takes as input an observed execution trace of a multithreaded program, and computes the largest set of feasible traces that can be inferred from the observed trace by reordering the events. In MCM, multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces, called $\mathcal{P}$-feasible traces that $\mathcal{P}$ can produce when $\mathcal{P}$ is completely or partially executed.

3.1.1.1 Events

A concurrent system is composed of a finite set of threads or processes, which communicate by performing atomic operations on concurrent objects such as shared memory locations, locks and semaphores [60]. For instance, a shared memory location is a concurrent object with *read* and

*Reprinted with permission from "Maximal causality reduction for TSO and PSO" by Shiyou Huang and Jeff Huang, 2016. International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 51, 447-461.

write operations, whose serial specification states that each *read* yields the same value as the one of the previous *write*. A (non-reentrant) lock is an object with *acquire* and *release* operations, whose serial specification consists of operation sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for a sequence of operations, and all consecutive pairs of *acquire-release* share the same thread. It means the an operation *acquire(l)* may be blocked until a *release(l)* operation happens from another thread. As threads are light-weighted processes, we use use threads to represent both threads and processes unless otherwise specified.

In MCM, events are operations performed by threads on concurrent objects, abstracted as tuples of attribute-value pairs. For example, (*thread = t1, op = write, target = x, data = 1*) is a *write* event by thread *t1* to memory location *x* with value 1. Table 3.1 lists common types of events are considered in MCM:

Table 3.1: Events considered in MCM.

$\texttt{begin}(t)/\texttt{end}(t)$	the first/last event of thread t
$\texttt{read}(t,x,v)/\texttt{write}(t,x,v)$	read/write x with value v
$\texttt{lock}(t,l)/\texttt{unlock}(t,l)$	acquire/release a lock l
$\texttt{wait}(t,o)/\texttt{notify}(t,o)$	wait/notify an object o
$\texttt{fork}(t,t')$	fork a new thread t'
$\texttt{join}(t,t')$	block until thread t' terminates

3.1.1.2 Traces

A trace is abstracted as a sequence of events. Though MCM can be encoded as different memory consistencies, we only consider sequential consistency [7] in this paper. Given a trace τ and any set S of concurrent objects, threads, shared variables and event types, we let $\tau\restriction_S$ denote the projection of τ restricted by S. For example, if $t \in S$ is a thread, then $\tau\restriction_t$ is the projection of τ to events by thread t; if loc is the shared location specified by S, then $\tau\restriction_{loc}$ is the projection of τ to events that access loc. If e is an event in τ, let $\tau\restriction_e$ denote the prefix of τ up to and including e. We also allow multiple restrictions. For instance, $\tau\restriction_{t,read}$ refers to the projection of τ to the *read*

events by t.

Given an event e, we use the following notations to help illustrate the consistency of a trace:

- op: $op(e)$ refers to the operation type of e;
- loc: $loc(e)$ refers to the memory location of e, and $e \in (read, write)$;
- val: $val(e)$ refers to the value of e, and $e \in (read, write)$;
- $last_{op}$: $last_{op}(\tau)$ refers to the last event of τ corresponding to operation op.

A trace τ is sequentially consistent iff $\tau\restriction_o$ siatisfies o's serial specification for any object o [60]. A sequentially consistent trace τ should hold the following:

- r**ead consistency** The value returned by a read event e should be equal to the one written by the most recent *write* to the same memory location, $val(e) = val(last_{write}(\tau\restriction_{loc(e)}))$;
- l**ock mutual exclusion** Each *release* event is paired with a *acquire* event on the same lock, and there is no any other *acquire* or *release* event on the same lock between each pair.
- **program order** For any two events $e_1, e_2 \in \tau\restriction_t$, if e_1 precedes e_2 in the program, then e_1 occurs before e_2 in $\tau\restriction_t$.
- **must happen-before** A *begin* event can happen only as a first event in a thread and only after the thread is forked by another thread. An *end* event can happen only as the last event in a thread, and a *join* event can happen only after the end event of the joined thread.

3.1.1.3 Feasibility Axioms

Consistency is a property of a trace alone, stating that all the serial specifications describing the legal behaviors of the involved concurrent objects are met. The most common characterizing axiom of $\mathcal{P}$-feasible, rooted in Lamport's happens-before causality [7] or Mazurkiewicz's trace theory [61], requires that $\mathcal{P}$-feasible should be closed under consistent interleavings. However, this axiom is too strong. In maximal causality model, two weaker axioms governing $\mathcal{P}$-feasible are proposed:

1. ***prefix closedness***, if $\tau_1\tau_2 \in \mathcal{P}$-feasible traces, then $\tau_1 \in \mathcal{P}$-feasible. It means the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible.

2. ***local determinism***, each event is only determined by the previous events in the same thread.

These two axioms allow us to associate a maximal set of traces $\mathsf{MaxCausal}(\tau)$ to any consistent trace τ, which comprises precisely the traces that can be generated by any program that can generate τ. Given a trace τ which is $\mathcal{P}$-feasible, we can generate a set of *prefixes* $\mathscr{P}$ which consist of events (partial or complete) from τ. The axiom **prefix closedness** provides us with the biggest possibility to change the order of the events in the original executed trace and guarantees that each new possible prefix from $\mathscr{P}$ is also $\mathcal{P}$-feasible. Therefore, the execution of the program along the *prefix* is feasible. The axiom **local determinism** implies that when we attempt to make an event e appear in the trace, we can include all the events that occur before e by the same thread in the prefix ,and make all the reads among such events return the same value as that in the original trace.

3.1.2 Constraints Encoding of MCR

In this Section, we introduce how MCR encodes a trace to a SMT constraint model that not only satisfies the maximal causal model but also makes the program yield new behaviors. The constraints model of MCR basically includes two parts: ❶ constraints (Φ^{mcm}) encoding MCM, which represent all the possible and valid interleavings that can be derived from a trace; ❷ constraints (Φ_{state}) enforcing that each interleavings is different from each other. Different than DPOR, MCR uses SMT constraints to reason about the maximal causality of the events in a trace. If a new behavior can be derived from the trace, MCR generates an interleaving that guides the re-execution of program to yield this new behavior.

Given a trace τ MCR encodes τ into a formula $\mathsf{MaxCausal}(\tau) = \Phi^{mcm} \wedge \Phi_{state}$, which contains all the different interleavings that can be dericed from τ. Φ^{mcm} consisting of three types of first-order logical constraints: (1) must-happen-before constraints (Φ_{mhb}); (2) lock-mutual-exclusion constraints (Φ_{lock}); (3) data-validity constraints ($\Phi_{validity}$). Φ^{mcm} is then conjoined with a new state constraint Φ_{state}. Φ_{state} enumerates all the reads in τ and the values observed by the

reads. To make the program yield a new behavior, Φ_{state} enfoeces at least one read to return a different value than what it observes in τ. MCR uses a SMT solver to solve the constraints to search different interleavings to make the program yield new behaviors. For implementation, we assign each event in the trace with an order variable O to denote its order in the generated interleaving. MCR builds the constraints over the order variables O to reason about the maximal causality of the events.

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 (obeying SC); (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. Clearly MHB yields a partial order over the events of τ which must be respected by any trace in $feasible(\tau)$. 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 $\Phi_{mhb} \equiv$ *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 $fork(t, t')$ and e_2 of the form $begin(t')$, etc.

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 feasible. For an event e to be feasible, 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. Let $\prec_e$ denote the set of events that must-happen-before an event e, and consider a read event *r=read(t,x,v)* in $\prec_e$ on a memory address x with value v by thread t. Let W^x

denote the set of all writes to x, and W_v^x the set of writes to x with value v, the $\Phi_{validity}$ constraint for e is encoded as $\bigwedge_{r \in \prec_e} \Phi_{value}(r, v)$, where $\Phi_{value}(r, v)$ is the state constraint that ensures r to read a value v:

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

Since MCM models all the incomplete traces as well, the data-validity constraint $\Phi_{validity}$ 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:

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

The constraint $\Phi_{validity}$ enforces the control flows. However, it is not trivial to figure out how the constraint $\Phi_{validity}$ is needed. Let us consider the example in Figure 3.1.

Initially x := y := 0	
p: *if (x == 0)* *r := x*	*q:* *x := 1*

Figure 3.1: An example showing that $\Phi_{validity}$ is necessary.

Suppose initially the program is executed in the order, p, p, q, and the program generates $r = 0$. To make $r := x$ return the value 1 written by $x := 1$, MCR enforces $\Phi_{state} = O_q < O_{p2}$ ($p2$ refers to the second statement in thread p) so that $x := 1$ happens before $r := x$. By conjoining with $\Phi_{mhb} = O_{p1} < O_{p2}$, the solver returns a possible solution $O_q = 0, O_{p1} = 1, O_{p2} = 2$, corresponding to a concrete schedule q, p, p. However, this schedule is not feasible because the if predicate is not satisfied under this schedule, and hence $r := x$ cannot be reached. To ensure the reachability of an event, MCR encodes the *data-validity* constraints into the formula. In other words, 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 returned by $r := x$ in the trace p, p, q, we need to guarantee that $x == 0$ is satisfied so that we have $\Phi_{validity} = O_{p1} < O_q$. Then we get the correct order from the solver $O_{p1} = 0, O_q = 1, O_{p2} = 2$ and construct the feasible schedule p, q, p, making $r = 1$.

Initially x := y := 0			
p: *x := 1*	*q:* *r1 := x*	*r:* *y := 1*	*s:* *r2 := y*

Figure 3.2: A simple example.

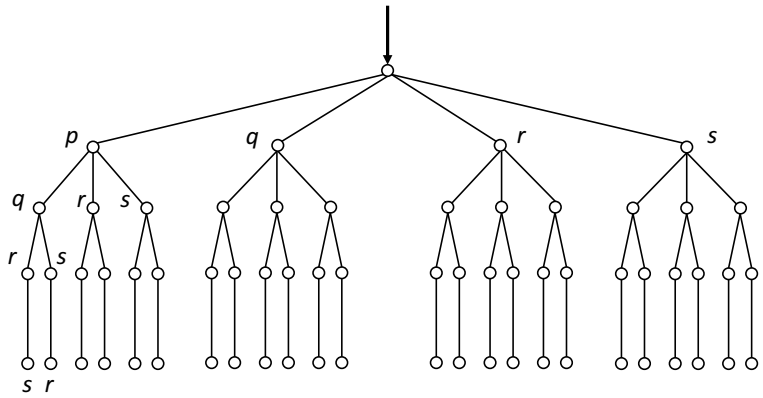


Figure 3.3: $feasible(\tau)$ inferred from a random trace τ of the program in Figure 3.2.

It is worth noting that the formula: $\Phi^{mcm}(\tau) = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{validity}$ encodes all the feasible interleavings, *i.e.*, $feasible(\tau)$, that can be inferred from the input trace τ. Each solution of the order variables to Φ^{mcm} corresponds to an interleaving in $feasible(\tau)$. The size of Φ^{mcm} is cubic in number of reads and writes in τ, and the size of $feasible(\tau)$ may be huge as the number of unique solutions to Φ^{mcm} can be exponential. Consider the program $\mathcal{P}$ in Figure 3.2. Given a random trace of the program $\tau = p.q.r.s$, then $feasible(\tau)$ contains all the feasible traces that

can be inferred from τ as shown in Figure 3.3. In this case, because all the four events can be executed in any order, in total $feasible(\tau)$ contains 24 possible traces including τ itself, but 20 of them are redundant. One execution is redundant if it generates the same output as that in another execution. For example, executions $p.q.r.s$ and $r.s.p.q$ are equivalent in this case to each other because both of them produce $r1 = 1 and r2 = 1$. As a result, similar to DPOR, we can apply a reduction algorithm to $feasible(\tau)$ to remove the redundant traces. In practice, MCR does not need to directly solve Φ^{mcm} to produce all the interleavings in $feasible(\tau)$. When used for checking properties, it often suffices to find one interleaving that satisfies the property. We show how to check assertion violation and data race properties using Φ^{mcm} in Section 3.1.3. It is shown in [58, 59] that $feasible(\tau)$ is both sound and maximal: any program which can generate τ can also generate all traces in $feasible(\tau)$, and for any trace τ' not in $feasible(\tau)$ there exists a program generating τ which cannot generate τ'.

New state constraints (Φ_{state}) It is impossible to consider every possible interleaving produced by a concurrent program. Model checkers (e.g., DPOR) are usually wrapped with a reduction technique to reduce the state space by removing equivalent interleavings from the state space. The key idea of MCR to eliminate redundant executions lies in enforcing 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 ensure 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.

3.1.3 Property Checking with Maximal Causal Reduction

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 $\phi \wedge \Phi^{mcm}$. 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 $\phi \wedge \Phi^{mcm}$ can be significantly simplified by tailoring Φ^{mcm} 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, MCR reduces the data validity constraint Φ_{rw} to consider only those in R, that is, $\Phi_{validity}(e)$ 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 $\nu(r)$ denote the value that can be returned by a read event $r = \mathit{read}(t, x, \_)$, 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. $\nu(r)$ is written as:

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

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

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

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

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

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

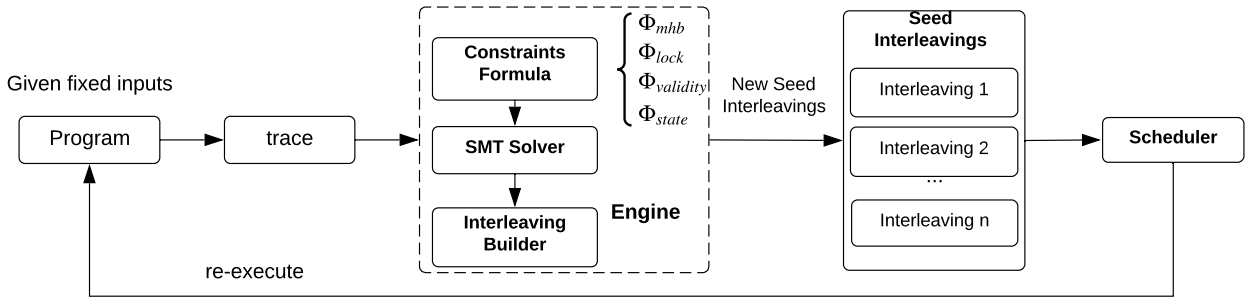


Figure 3.4: Workflow of MCR. The engine part of MCR constructs SMT constraints over the trace to explore new program schedules and the new trace is generated by re-executing the program under the dynamic scheduler. Reprinted with permission from [1].

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:

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

3.1.4 MCR Workflow

In this section, we introduce the system design of MCR. Figure 3.4 presents the workflow of MCR. For each program to be explored, MCR instruments the code to collect the interesting events depicted in Section 3.1.1. Given a multithreaded program (we assume that the input is fixed), MCR systematically explores the state space of the program in a closed loop by executing the program along the seed interleavings. In the rest of the paper, we use *prefix* and *seed interleaving*

interchangeably.

Initially, MCR executes the instrumented program following a random interleaving and generates an initial trace τ. Then τ is used to compute the maximal casual traces $\mathsf{MaxCausal}(\tau)$, and from which MCR generates new *seed interleavings* $\mathcal{P}$ (if there exists any).

Definition 3.1.1 ($\mathcal{P}$)**.** Given a trace τ, a *seed interleaving* $\mathcal{P}$ is the shortest sequence of scheduling choices that matches a read with a write which writes a different value from that in τ. Suppose the read r in τ is enforced to return a different value, $\mathcal{P} = \tau' r$ and $\forall e \in \tau'$ happen before r, and $val(r) \neq val(r')$, $r' \in \tau$ is the same read as $r \in \mathcal{P}$.

Each seed interleaving is produced by encoding

$$\mathsf{MaxCausal}(\tau) = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{validity}$$

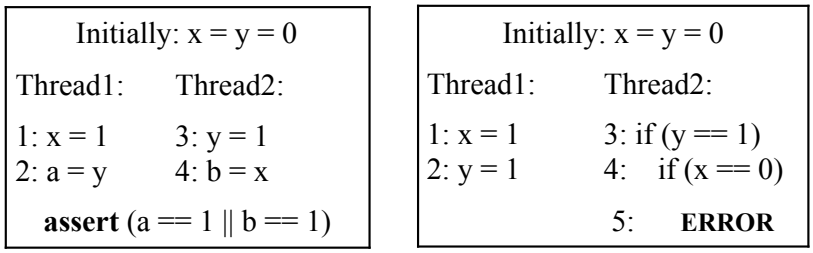
together with a *new state* constraint Φ_{state} over a read event in τ enforcing it to read a new value, such that the seed interleaving will drive the program to reach a new state, *i.e., at least one read will read a new value*. The seed interleavings are then explored as the prefix of an execution to cover new states, and to generate new seed interleavings to explore. In this work, we use seed interleavings and prefixes interchangeably.

Let us give formal the definition of the state during the exploration and what two different states are.

Definition 3.1.2 (S)**.** A state S of a concurrent program in MCR reflects the values returned by the read accesses to shared variables after the program is executed in a given order. Formally, given a trace τ, $S = \{r \mapsto val(r) | r \leftarrow \tau\restriction_{read}\}$. $S \neq S'$ iff $\exists r \in S \cap S'$, $val(r) \neq val(r')$, r and r' correspond to the same read from S and S', respectively.

The state-space of a concurrent program is a combination of the reads and the values from which the reads can read. For instance, if a program contains two reads and each of them can return two different values, then the number of the total states of the program is 4.

Under the control of a dynamic scheduler, MCR re-executes the program along the seed interleaving and collects the trace to generate more seed interleavings. MCR terminates when all



(a) A TSO example (b) A PSO example

Figure 3.5: (a) shows a program with error under TSO, but correct with SC (b) shows a program with error under PSO, but correct with SC. Reprinted with permission from [2].

seed interleavings have been explored and no new seed interleavings can be generated. For a new value constraint, there can be multiple interleavings in $\mathsf{MaxCausal}(\tau)$ that satisfy the constraint. To avoid generating redundant seed interleavings, MCR ensures that the prefix of each new explored interleaving is always preserved and the generated seed interleaving is the shortest among all satisfiable interleavings.

3.1.5 A Running Example for MCR

We use the example in Figure 3.5(a) to illustrate MCR. The program has 6 different executions (3 are redundant) under SC, but 24 different executions under TSO (20 are redundant). MCR is able to explore all the state-space under SC via only 3 executions, but it fails to expose the assertion violation that is only possible under TSO.

Let e_i denote the event at the line number i. Given a trace $\tau = \langle e_1, \cdots, e_n \rangle$, MCR uses n integer variables $\langle O_1, \cdots, O_n \rangle$ to denote the order in which the events happen in a certain execution. The value of O_i represents the position of e_i in a trace. If $O_i < O_j$, then e_i will be executed before e_j in the generated interleaving.

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 $\Phi_{mhb} = O_1 < O_2 \wedge O_3 < O_4$. Since the trace contains two reads, e_2 ($R(y)$=0) and e_4 ($R(x)$=1), to generate new seed interleavings, MCR tries to enforce each of the two reads to read a different value in future executions. For example, for e_2, it adds the new state constraint $\Phi_{value} = O_3 < O_2$ to enforce

$R(y)$ to read value 1 (written by e_3) instead of 0. By solving this constraint conjoined with Φ_{mhb}, the SMT solver will return a solution such as $\{O_1 = 1, O_2 = 3, O_3 = 2\}$. From this solution, MCR will generate 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 trace $\tau_1 = \langle e_1, e_3, e_2, e_4 \rangle$, and reach a new state (a=1,b=1). Then the exploration along this seed interleaving is finished, because there is no new value that can be read by any read event in τ_1. Similarly, the read event e_4 in τ_0 generates a new seed interleaving e_3-e_4, which produces a new trace $\tau_2 = \langle e_3, e_4, e_1, e_2 \rangle$ that reaches a new state (a=1,b=0).

As we can see, MCR successfully explores all the three possible program states under SC – (a=0,b=1), (a=1,b=1) and (a=1,b=0) – through only three different executions. However, MCR misses the assertion violating state (a=0,b=0), which is feasible under TSO and PSO. To reach this state, there must be at least a reordering between (e_1, e_2) or (e_3, e_4). Neither of them is possible in the formulation of MCR, because both of them violate the Φ_{mhb} constraint. Similarly, MCR cannot trigger the PSO assertion violation in Figure 3.5(b), because e_1 must-happen-before e_2 under SC. Next, we first present the semantics of TSO and PSO in Section 3.2.1. We will show how our approach enables finding the errors under TSO and PSO in Section 3.3.

3.2 Relaxed Memory Models: TSO and PSO

MCR works efficiently for checking concurrent programs under sequential memory model, but fails to detect bugs under relaxed memory models, *e.g.*, TSO and PSO. Figure 3.6 shows a real bug extracted from a large program (with over 40K lines of code) running on an electron microscope [3]. The program runs safely under SC and TSO. However, an error (lines 15-17 of Figure 3.6) is triggered when it runs under PSO and unfortunately caused a loss of $12 million of equipment. The root cause of the error is that the write to the object *curPosition* can happen before the write to the field of the object, which is allowed under PSO. What is worse is that this error can hardly be reproduced. On average, the error appears only once in every $500,000$ loop iterations of the program[2].

[2]Interestingly, our approach takes only three runs to find this PSO bug.

```
1.class A {
2.    static Point currentPos = new Point(1,2);
3.    static class Point {
4.        int x;
5.        int y;
6.        Point(int x, int y) {
7.            this.x = x;
8.            this.y = y;
9.        }
10.    }
11.    public static void main(String[] args) {
12.        new Thread() {
13.            void f(Point p) {
14.                synchronized(this) {}
15.                if (p.x+1 != p.y) {
16.                    System.out.println(p.x+" "+p.y);
17.                    System.exit(1);
18.                }
19.            }
20.            @Override
21.            public void run() {
22.                while (currentPos == null);
23.                while (true)
24.                    f(currentPos);
25.            }
26.        }.start();
27.        while (true){
28.           currentPos =
29.          new Point(currentPos.x+1, currentPos.y+1);
30.        }
31.    }
32.}
```

Figure 3.6: A real PSO bug in an electron microscope software [3]. This bug caused a $12 million loss of equipment. Reprinted with permission from [2].

3.2.1 TSO and PSO

We present the operational semantics of hardware memory models TSO and PSO [62, 8] following the same spirit as previous work [33, 36]. We also discuss the relation of *Java Memory Model* (JMM) to TSO and PSO at the end of this section.

Total Store Ordering (TSO) TSO allows a read to complete before an earlier write to a different memory location, but maintains a total order over writes and operations accessing the same memory location. There are four kinds of operations:

- **Store** Whenever a thread t_i executes a store operation, it does not update it to the shared main memory immediately. Instead, the store is buffered to the store buffer B_i (which is a FIFO queue).

- **Load** When a thread t_i executes a load to a memory location x, it will first check its buffer B_i. If the buffer contains the store to x, then the load gets the latest value written to x in the buffer; otherwise the load gets the value from the main memory.

- **Update** An update operation flushes the store buffer into the main memory. It can happen at any point as long as the store buffer is not empty. The memory model allows any thread to non-deterministically perform the *update* operation any number of times at any state of the execution.

- **Fence** Fences are special machine instructions that prevent reordering between the operations before and after the fence. A fence operation can only be executed when the buffer is empty.

Consider a concurrent program with n threads $\boldsymbol{T} = t_1 \times t_2 \times \cdots \times t_n$ and each thread t_i is associated with a store buffer B_i, forming a set of store buffers $\boldsymbol{B} = B_1 \times B_2 \times \cdots \times B_n$. Let $\mathbf{M} = m_1, \cdots, m_k$ be the memory locations in the program, and each memory location can take value from a data domain. We define a system configuration as a tuple $\mathcal{C} = \langle T, M, B \rangle$, and the local configuration of thread t_i as $\mathcal{C}_i = \langle M, B_i \rangle$ where M is the current value in each memory location, and B_i is the current value in the store buffer of thread t_i.

For two system configurations $\mathcal{C} = \langle T, M, B \rangle$ and $\mathcal{C}' = \langle T', M', B' \rangle$, we use the notation $\mathcal{C} \xrightarrow{op} \mathcal{C}'$ to denote the transition from $\mathcal{C}$ to $\mathcal{C}'$ by executing the operation op, where op is one of the four operations (store/load/update/fence) defined above by a certain thread. Consider that op is executed by thread t_i. The transition on the system configuration is the same as that on the local configuration of t_i: $\mathcal{C}_i \xrightarrow{op(t_i)} \mathcal{C}'_i$. We use $w(t_i, x, v)$/$r(t_i, x, v)$/$u(t_i, x, v)$/$fence(t_i)$ to denote these four operations respectively, meaning that thread t_i writes/reads value v to/from memory location x, updates the value v to x from the store buffer to the main memory, or performs the fence operation, respectively.

Let $B \oplus (x, v)$ denote buffering the write (x, v) to the store buffer B, $B \ominus (x, v)$ flushing the write (x, v) to the main memory from B, and $B = \varepsilon$ denote that B is empty. Let $B(x)$ denote retrieving the value of the *most recent* buffered write to x in B. Note that $B(x)$ can be *null* when there is no buffered write to x in B. We use $\varnothing$ to denote the null value.

The operational model is defined as follows:

1. ***Store***: $\mathcal{C}_i \xrightarrow{w(t_i,x,v)} \mathcal{C}'_i$ *iff* $M' = M$ and $B'_i = B_i \oplus (x, v)$.

2. ***Load***: $\mathcal{C}_i \xrightarrow{r(t_i,x,v)} \mathcal{C}'_i$ *iff* $M' = M$, $B'_i = B_i$ and either one of the following two cases:

   (a) **Load from buffer**: $B_i(x) \neq \varnothing$ and $v = B_i(x)$.

   (b) **Load from memory**: $B_i(x) = \varnothing$ and $v = M[x]$.

3. ***Update***: $\mathcal{C}_i \xrightarrow{u(t_i,x,v)} \mathcal{C}'_i$ *iff* $B'_i = B_i \ominus (x, v)$ and $M' = M[x \hookleftarrow v]$.

4. ***Fence***: $\mathcal{C}_i \xrightarrow{fence(t_i)} \mathcal{C}'_i$ *iff* $B_i = \varepsilon$ and $M' = M$.

Partial Store Ordering (PSO) PSO is similar to TSO except that it allows reordering writes on different memory locations. The operational model of PSO can be defined by slightly modifying the TSO model defined above. Under PSO, each thread has *multiple* store buffers, each of which corresponds to one unique memory location. In other words, each memory location is assigned with a store buffer. Two consecutive write operations on different memory locations can be buffered into different store buffers, allowing them to be executed out of the program order.

3.2.2 JMM on TSO/PSO platforms

The motivation of our work stems from the real bug exhibited in Figure 3.6. Readers may concern that Java has its own memory model (Java Memory Model or JMM) [63] and the compiler and hardware reorderings need to respect the JMM. However, this does not impair the validity of our motivation, because hardware memory models are orthogonal to language models. For any language, as long as the compiler does not insert fences to prohibit reorderings, the hardware may exhibit TSO/PSO behaviors. Because the JMM allows the delayed stores as that in TSO and PSO [64, 65], the JVM inserts no barriers to disable the reorderings on TSO/PSO platforms. As a consequent, the reordering can cause the bug in Figure 3.6 to occur. To avoid this bug, one solution is to declare both the fields x and y in Figure 3.6 as final. For final fields, to respect JMM, the JVM inserts a barrier after the initialization of final fields. Thus, once an object is constructed, the values assigned to the final fields of the object are visible to all other threads. This prevents the bug from occurring in Figure 3.6.

3.3 MCR for TSO and PSO

Our approach builds upon MCR but enables it to work for both TSO and PSO. There are two crucial differences between our approach and the original MCR [19]:

1. We relax the *must-happens-before* (MHB) relation between events to capture the semantics of TSO and PSO when producing the seed interleavings.

2. We develop novel replay algorithms for TSO and PSO interleavings that allow the reordering of events by the same thread.

In this section, we first describe how to relax the MHB constraints to allow the semantics of TSO and PSO defined in Section 3.2.1. We then present our replay algorithms. Finally, we discuss the limitation of our approach.

3.3.1 Relaxation on MHB Constraints

To encode the semantics of TSO and PSO, we relax the MHB constraints Φ_{mhb} of MCR (recall Section 3.1.2). Specifically, we decompose Φ_{mhb} into two components:

$$\Phi_{mhb} = \Phi_{mem} \wedge \Phi_{sync}$$

where (1) the memory operation constraint (Φ_{mem}) captures the reordering semantics allowed by different memory models (TSO or PSO); (2) the synchronization constraint (Φ_{sync}) captures the *happens-before* relation entailed by synchronizations. Φ_{sync} is common for all memory models (e.g., SC/TSO/PSO).

Constraints on memory operations (Φ_{mem}) Under TSO, following the operational semantics defined in Section 3.2.1, we construct Φ_{mem} with four rules: (1) *write-to-write constraints* (Φ_{ww}). For all writes by the same thread, their order should be consistent with the program order. (2) *memory location constraints* (Φ_{addr}). For all the reads and writes by the same thread that access the same memory address, they should follow the program order. (3) *read-to-read constraints* (Φ_{rr}). All read operations from the same thread should follow the program order. (4) *read-to-write*

MCR		$O_1<O_2, O_3<O_4$
Our Approach	SC	$O_1<O_2, O_3<O_4$
	TSO	O_1, O_2, O_3, O_4
	PSO	O_1, O_2, O_3, O_4

(a)

MCR		$O_1<O_2, O_3<O_4$
Our Approach	SC	$O_1<O_2, O_3<O_4$
	TSO	$O_1<O_2, O_3<O_4$
	PSO	$O_1, O_2, O_3<O_4$

(b)

Figure 3.7: The *must-happen-before* constraints constructed by MCR and our approach on the TSO and PSO examples in Figure 3.5(a) and Figure 3.5(b), respectively. Reprinted with permission from [2].

constraints (Φ_{rw}). Any read operation and its following write operation from the same thread should follow the program order. Together, Φ_{mem} is represented as the conjunction of these four constraints:

$$\Phi_{mem} = \Phi_{ww} \wedge \Phi_{rr} \wedge \Phi_{rw} \wedge \Phi_{addr}$$

PSO is a further relaxation of TSO. PSO not only allows the write-to-read reordering allowed by TSO, but also the reordering of write-to-write to different memory locations. Therefore the only difference between the Φ_{mem} constraint under TSO and PSO is on the rule Φ_{ww}. In PSO, Φ_{ww} ensures only that all writes to the same memory location from the same thread should follow the program order.

Constraints on synchronizations (Φ_{sync}) For all synchronizations (*i.e.*, *lock*/*unlock* and *begin*/*end*) by the same thread, they should always be executed in the program order. Moreover, for each synchronization, all its proceding reads and writes should always happen before it, and all its following reads and writes should always happen after it.

3.3.2 New States under Relaxed MHB

To show the difference brought by the relaxed MHB constraints compared to the original MCR, we use the example in Figure 3.5(a) again to illustrate how it enables exposing the TSO and PSO errors which MCR fails to expose. Same as in Section 3.1.5, let us assume that $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$ is observed as the initial trace. Figure 3.7 shows a comparison between the MHB constraints on

τ_0 constructed by MCR and by our approach for TSO and PSO, respectively. Consider the read e_4 ($R(x)$=1). To make $R(x)$=0 in the new seed interleaving, MCR enforces e_4 to happen before e_1 by the constraint $O_4 < O_1$. Under TSO, because e_3 does not necessarily happen before e_4, our approach does not enforce $O_3 < O_4$ (as shown in Figure 3.7a) compared to MCR. As a result, the generated new seed interleaving by our approach will be just e_4, while it is e_3-e_4 by MCR. By replaying the program with the new seed interleaving e_4, our approach will explore a new execution and generate a new trace τ_1 starting with e_4, such as $\tau_1 = \langle e_4, e_1, e_2, e_3 \rangle$. In this case, τ_1 reaches the state (a=0,b=0), which violates the TSO assertion.

Likewise, under PSO, to expose the error in Figure 3.5b, our approach can generate an execution $\tau_1 = \langle e_2, e_3, e_4, e_1 \rangle$ because of the reordering between e_1 and e_2 under PSO.

3.3.3 Deterministic Replay

A key challenge in extending MCR from SC to TSO and PSO lies in how to replay the TSO and PSO interleavings. Under the original MCR, the interleaving is abstracted as a sequence of schedule choices, with each choice representing a thread ID by the corresponding operation on a shared variable. Before a thread executes an operation on a shared location, it is blocked first, and then the scheduler queries the seed interleaving to decide which thread to execute next. For SC, since the global order in the generated seed interleaving is consistent with the program order, this replay strategy guarantees that the operation chosen by the scheduler exactly matches with the event in the interleaving. However, because operations can be executed out of the program order under TSO/PSO, the simple global-ordering based replay approach in the original MCR no longer works. To realize the reordering, we rely on a store buffer (a FIFO queue) assigned to each thread to delay the execution of a store. The difficulty comes from the non-determinism of the *update* operation (recall Section 3.2.1) since it can happen any time at any point of the execution to flush the store buffer. To deterministically enforce a seed TSO/PSO interleaving, there are two key issues to be addressed: (1) *when to buffer a write*; and (2) *when to flush the buffered write to the main memory*.

To solve this problem, we first extend the original abstraction of the SC interleaving in MCR

by adding the memory location information – $addr$ – to each operation. The new abstraction of *interleaving* is defined as follows:

Definition 3.3.1. An interleaving is a sequence of schedule choices, with each schedule choice $c(tid, addr)$ consisting of a thread ID tid and a memory location $addr$ that is expected to be accessed by the corresponding operation.

The key idea of our TSO and PSO replay algorithms is to use the accessed memory location to decide whether to buffer a write by checking it against the information in the seed interleaving.

Store Buffering/Updating Before performing a store operation, we first check if the memory location accessed by this operation is the same as the one in the seed interleaving. If yes, we can flush the store to the main memory. Otherwise, we buffer the store in the store buffer (a FIFO Queue). At this point, we do not update the schedule choice since the operation has not been executed from the view of the interleaving. Later, when the address by the event in the interleaving matches with the one buffered in the FIFO queue, we flush the value to the main memory and also update the schedule choice to the next one.

Fence The operational model of TSO and PSO requires that before performing a fence operation, all the buffered stores in the *store buffer* should be flushed into the memory. However, in our approach, the re-execution of a program is controlled by a given interleaving, and our replay algorithm guarantees that when the scheduler meets a fence operation, the buffer must be empty. The reason is that all events that occur before a fence should happen before the fence and we have already constrained all such events to happen before the fence in the formula (see Section 3.3.1). Therefore, when the scheduler is about to execute a fence of an interleaving, all the events before this fence have already been executed (all buffered writes have already been flushed), and thus the buffer is empty at this moment.

Based on the new abstraction above, we can prove the following two theorems to guide our replay algorithms and to guarantee their correctness. Theorem 1 guides our algorithm to buffer writes and Theorem 2 guides our algorithm to flush the buffered writes to memory.

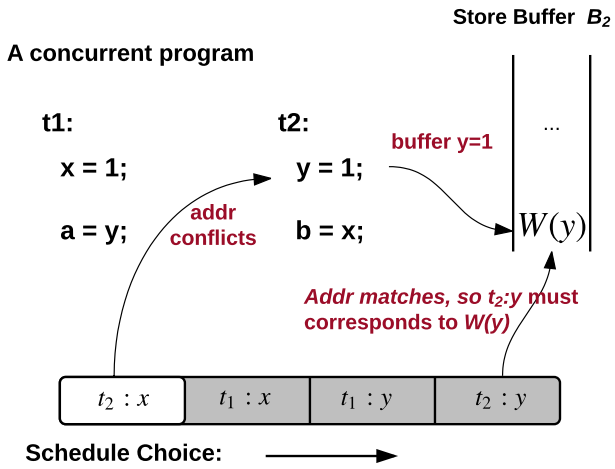


Figure 3.8: An example to illustrate Theorem 1 and Theorem 2. B_2 is the store buffer associated with thread t_2. Reprinted with permission from [2].

Theorem 1. At replay, when the *program counter* (PC) points to an event, say e_i, corresponding to the choice of the schedule, say c_j, if $addr(e_i) \neq addr(c_j)$, then e_i must be a write operation and it needs to be buffered.

Proof sketch: If e_i is a read or a fence operation, it implies that a later operation c_j (later according to program order) is allowed to happen before a read/fence operation. This contradicts the TSO and PSO operational models defined in Section 3.2.1. Hence e_i must be a write, and a certain operation matching with c_j that accesses a different location should be executed before e_i. Therefore, e_i must be buffered.

Theorem 2. When considering a schedule choice c_j in the seed interleaving, if $addr(c_j)$ equals to the memory location of the write at the head of the store buffer, then c_j must be corresponding to that buffered write.

Proof sketch: Proof by contradiction. Suppose that c_j is a read or a fence. Since there is a write w in the store buffer by the same thread that accesses the same address as c_j does, it means that the read (or fence) is allowed to happen before the write w which is before it. This again contradicts the operational models of TSO and PSO. Therefore, c_j must be a write. Similarly, c_j cannot correspond to any other write in the store buffer, otherwise c_j would be allowed to be

executed before its preceding writes. Hence, c_j must correspond to the buffered write w and w should be flushed.

Example Figure 3.8 illustrates the two theorems above. When considering the schedule choice $t_2 : x$, the instruction $y = 1$ is to be executed. However, their addresses do not match, which implies that the write $y = 1$ should be buffered (Theorem 1). When considering the last schedule choice $t_2 : y$, its address matches with that of the write at the head of thread t_2's store buffer (B_2), which implies that $t_2 : y$ is a buffered write and has to be flushed to memory (Theorem 2).

3.3.4 TSO Replay

To replay TSO interleavings, we associate each thread t_i with a FIFO queue (to simulate the store buffer B_i) and we assume the queue is unbounded. The *interleaving* here is a seed interleaving generated based on the solution given by the SMT solver. Each interleaving (recall Definition 3.3.1) consists of a sequence of schedule choices $c(tid, addr)$. The program is executed under the control of an application-level scheduler to enforce the schedule choices specified in the interleaving.

Algorithm 1 shows how we replay a TSO interleaving. The key idea is to determine whether to buffer or update a *store* by comparing its memory location with that specified in the schedule choice. We use a variable *index* to indicate the current position of the interleaving. The index is initialized to 0 and incremented by 1 each time when an instruction is executed (except the store buffer operation). Before the program executes a load or a store on a shared variable, the program is blocked and the schedule choice c given by the interleaving is queried to decide the next instruction. The next instruction is chosen by the program counter via the schedule choice (see the statement $PC(c)$ at line 4). Before executing an instruction, the algorithm proceeds depending on its type. If the instruction is a *store*, the algorithm first checks whether the memory location of this store and that specified in the schedule choice are equal or not. If they are equal, we write the value to the memory (see line 10). Otherwise we buffer the store into the thread's FIFO queue B_i. If the instruction is a *load*, the algorithm will first check if there is a buffered write in B_i that writes to the same address. If yes, the most recent buffered value will be returned; otherwise, the value from

Algorithm 1: TSO replay algorithm

Input : A seed interleaving S – schedule choices
Return: a new trace by logging the instruction executed

```
1  Initial: index = 0 // global
2  while index < S.length do
3      c ← S[index]
4      Inst ← PC(c) // guided by the schedule choice
5      i = tid(Inst)
6      x = addr(Inst)
7      if Inst is a store then
8          v = value(Inst) // the value of the store
9          if addr(Inst) == addr(c) then
10             Write(x, v)
11             index = index+1
12             updateCheck_TSO(S)
13         else
               // buffer the store
14             B_i ← B_i ⊕ (x, v)
15         end
16     else if Inst is a load then
17         if x in B_i then
18             v = B_i(x) // read the most recent value from buffer
19         else
20             v = mem(x) // read the value from memory
21         end
22         index = index+1
23         updateCheck_TSO(S)
24     else
           // fence - the buffer should be empty
25         Fence()
26         index = index+1
27     end
28 end
```

the main memory will be returned. For *fence* instructions, we simply proceed without the need to flush the buffer because the buffer should already be empty as discussed in 3.3.3.

Each time afte a *load* or *store* is executed, our algorithm will check if there are stores in the FIFO queue that need to be updated to the memory. The function $UpdateCheck\_TSO$ in Algorithm 2 shows the process for updating the buffered stores for TSO and PSO. Recall Theorem 2

Algorithm 2: After a load/store, check whether there are pending stores in the buffer that need to be updated.

```
1  Function UpdateCheck_TSO(S):
2      c ← S[index] // return if index out of bound
3      i = tid(c)
4      while (addr(c) == addr(B_i[0])) do
5          B_i ← B_i ⊖ (x, v)
6          flush(x, v) // flush to memory
7          index = index+1
8          c ← S[index] // return if index out of bound
9          i = tid(c)
10     end
11 Function UpdateCheck_PSO(S):
12     c ← S[index] // return if index out of bound
13     i = tid(c)
14     j = varId(c)
15     while (addr(c) == addr(B_i^j[0])) do
16         B_i^j ← B_i^j ⊖ (x, v)
17         flush(x, v) // flush to memory
18         index = index + 1
19         c ← S[index] // return if index out of bound
20         i = tid(c)
21         j = varId(c)
22     end
```

that when the current schedule choice has the same memory location as that of the store at the head of the buffer, then the expected operation must be a store that has been buffered. We hence follow this condition to detect all such stores and update them to the memory.

Termination Note that our algorithm just replays the instructions within the interleaving, it terminates when $index \geq S.length$. For those outside of the interleaving, they are executed following the program order. The number of while-loop iterations (line 2) is determined by the index, which specifies the schedule choice. Although the index keeps unchanged when buffering a store (line 13) it will eventually be increased to the size of the schedule and terminates the algorithm. Each time after we execute or buffer an instruction, the program counter will be updated to point to the next instruction controlled by the schedule choice (line 4). Although the index is not changed, the

address or the type (read/write) of the operation will change, which leads to the execution of a read/write, and eventually the execution of the update function which increases the index.

The correctness of our algorithm is guaranteed by the following theorem.

Theorem 3. Algorithm 1 correctly replays all the events in the given TSO interleaving.

Proof sketch: Theorem 1 and Theorem 2 guarantee that all events in the given interleaving will be replayed in the order as specified in the interleaving. To prove Theorem 3, we only need to prove that all the events in the interleaving will be replayed, i.e., no event will be missed. In our replay algorithm, each time after an event is executed, the $updateCheck\_TSO(S)$ subroutine will update (i.e., execute) the buffered events until the index points to an event that is not buffered. Suppose there exists an event e that is not executed when the replay algorithm is finished. If e corresponds to a non-buffered event, e should be executed directly when it is chosen by the index. On the other hand, if e corresponds to a buffered store, there must exist a nearest non-buffered event e' preceding e in the interleaving. After e' is executed, $updateCheck\_TSO(S)$ subroutine will execute e. Thus, in any case, no event will be missed.

Example To illustrate the algorithm, consider a TSO interleaving of the program in Figure 3.5a: e_4, e_1, e_2, e_3, which corresponds to the sequence of schedule choices: (t_2,x),(t_1,x),(t_1,y),(t_2,y). When replaying this interleaving, the schedule choice (t_2,x) guides the program to execute the instruction $y = 1$ at line 3. Since the addresses do not match, $y = 1$ is buffered and the schedule index does not change. When the program reaches the instruction $b = x$ at line 4, since it is a load operation, $b = x$ is executed directly. Similarly, $x = 1$ and $a = y$ at lines 1 and 2 are executed under the schedule choices (t_1,x),(t_1,y). After $a = y$ is executed, the algorithm detects that the schedule choice (t_2,y) corresponds to the buffered write $y = 1$ in the FIFO queue. Therefore, $y = 1$ is updated to the memory.

3.3.5 PSO Replay

Replaying PSO interleavings is similar to that for TSO. The only difference is that under PSO, each thread is associated with multiple FIFO queues with each queue corresponding to one unique

Algorithm 3: PSO replay algorithm

Input : A seed interleaving S – schedule choices
Return: a new trace by logging the instruction executed

Initial: $index = 0$ `// global`
while *(*$index < S.length$*)* **do**
  $c \leftarrow S[index]$
  $Inst \leftarrow PC(c)$ `// guided by the schedule choice`
  $x_j = addr(Inst)$ $i = tid(Inst)$
  $j = varId(x_j)$
  if $Inst$ *is a store* **then**
    $v = value(Inst)$ *//the value of the store* **if** $addr(Inst) == addr(c)$ **then**
      $Write(x_j, v)$ $index = index + 1$ $updateCheck\_PSO(S)$
    else
      `// buffer the store`
      $B_i^j \leftarrow B_i^j \oplus (x_j, v)$
    end
  else if $Inst$ *is a load* **then**
    if x_j *in* B_i^j **then**
      $v = B_i^j(x_j)$ `// read the most recent value from buffer`
    else
      $v = mem(x_j)$ `// read the value from memory`
    end
    $index = index + 1$
    $updateCheck\_PSO(S,index)$
  else
  end
  //fence – the buffer should be empty $Fence()$
  $index = index+1$
end

memory location. For a thread t_i accessing memory locations $m_1, m_2 \ldots, m_k$, we assign a FIFO queue B_i^k for memory location m_k. Algorithm 3 shows the replay process. The key difference from Algorithm 1 is that when buffering a store under PSO, the algorithm needs to buffer the store to the FIFO queue corresponding to the memory location accessed by the store. We use $varId()$ to get the unique ID assigned to each variable by each thread, and the buffer B_i^j correlates to a variable which belongs to thread i and has variable ID j. The function $UpdateCheck\_PSO$ in Algorithm 2 shows the process for updating the buffered stores under PSO (similar to the process

for TSO).

3.3.6 Discussion

We note that our approach is not optimal for minimizing the redundancy under TSO and PSO, albeit MCR is optimal for SC. The root problem is that under TSO and PSO, the generated seed interleavings are shorter than that under SC, which results in the possibility that two distinct seed interleavings may reach the same state. Consider again the example in Figure 3.5(a). In the initial execution $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$, there exists two reads e_2 ($R(y)$=0) and e_4 ($R(x)$=1). If we force the read on y (e_2) to read value 1 which requires that e_3 to happen before e_2, our approach will generate a seed interleaving e_3-e_2. Likewise, if we force the read on x (e_4) to read value 0, which requires that e_4 to happen before e_1, our approach will generate a seed interleaving e_4.

If we continue with the seed interleaving e_3-e_2, we will generate two more executions:

- $\tau_1 = \langle e_3, e_2, e_1, e_4 \rangle$ (a=1,b=1);
- $\tau_2 = \langle e_3, e_2, e_4, e_1 \rangle$ (a=1,b=0).

And if we continue with the seed interleaving e_4, we will generate another two executions:

- $\tau_3 = \langle e_4, e_1, e_2, e_3 \rangle$ (a=0,b=0);
- $\tau_4 = \langle e_4, e_3, e_2, e_1 \rangle$ (a=1,b=0).

As we can see, under TSO, our approach explores five executions to cover the whole state-space. However, the optimal solution should only explore four executions, because there are only four unique states. In our approach, τ_2 and τ_4 are equivalent to each other, both of which reach the state (a=1,b=0).

The only difference between these two redundant executions is the permutation of the two seed interleavings: e_3-e_2 and e_4, where e_3-e_2 targets the value read from y and e_4 the value read from x. Since these two seed interleavings are non-overlapping and are permutable, they lead to the same state. However, it is difficult to prune this type of redundancy in the current MCR, because the seed interleavings are generated independently without considering their permutations. A potential way

to eliminate this redundancy for TSO and PSO would be to merge multiple independent seed interleavings into a single one. Nevertheless, this type of redudancy only accounts for a minor portion of the explored executions, because the space of seed interleavings is significantly smaller than the whole interleaving space. As we will show in our experiments in Section 3.5, even with this redundancy, MCR under TSO and PSO is much more effective than existing approaches on both popular benchmarks and real programs.

3.4 Case Study

In this section, we present a case study of our approach on the real PSO bug (shown in Figure 3.6). We also compare our approach with the DPOR algorithm for PSO by Zhang et al. [66] implemented in the *rInspect* tool. We show that our approach is much more effective than [66] in both state-space exploration and bug finding.

To make the problem more clear, we simplify the program to its equivalent form shown in Figure 3.9a. Note that the simplified example in Figure 3.9a is slightly different from the program in Figure 3.6, but it exactly presents how the PSO bug occurs in the original program does. Lines 2 and 3 of the example in Figure 3.9a simulate the initialization instructions when constructing a new *Point* object. Lines 4 and 5 write the initial values to the fields of the object. We use an integer variable z to indicate whether or not the object is constructed. If $z = 1$, it means that the object is constructed. In our case, we do not simulate the *while* statements and we just update the values of x and y once, which is enough to reveal the bug. To show the power of our approach we use two *for* loops with N times to change the complexity of the state-space.

Suppose that in the initial execution the two threads run sequentially following the program order, we will obtain a trace as below[3]:

$$\tau_0 = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8^1, e_8^2 \rangle$$

where e_i corresponds to an event performed at line i, and e_8^1 and e_8^2 corresponds to the first and second (read to x and y) events at line 8, respectively. In the initial trace, there are three reads: e_7

[3]We set N to 1 in this case to simplify the presentation.

```
      Initial x = y = z =0

Thread 1:            Thread 2:
fork();              LOOP N:
LOOP N:              7.if(z==1)   //R(z)
1. z = 0;  //W(z)
2. x = 0;  //W(x)    8. if (x+1 != y) //R(x), R(y)
3. y = 0;  //W(y)    9.   print(x, y) //R(x), R(y)
4. x = 2;  //W(x)              ERROR!
5. y = 3;  //W(y)
6. z = 1;  //W(z)
```

(a) A simplified version of the program in Figure 3.6. An execution 1-2-6-7-8-3-4-5-8-9 can trigger this error under PSO.

$\Phi_{mhb(\text{SC, TSO})}$	$O_1<O_2<O_3<O_4<O_5<O_6$ $O_7<O_8^1<O_8^2$
$\Phi_{mhb(\text{PSO})}$	$O_1<O_6\ \ O_2<O_4\ \ O_3<O_5$ $O_7<O_8^1<O_8^2$
$\Phi_{validity}$	$O_6<O_7$
Φ_{state}	$O_2<O_8^1 \wedge (O_4<O_2 \vee O_8^1<O_4)$

$$\Phi=\Phi_{mhb} \wedge \Phi_{validity} \wedge \Phi_{state}$$

(b) The generated constraints by our approach under three different memory models for the example in Figure 3.9a.

Figure 3.9: A simplified example from Figure 3.6 and the SMT constraints. Reprinted with permission from [2].

($R_7(z)$=1), e_8^1 ($R_8(x)$=2) and e_8^2 ($R_8(y)$=3). The index of R corresponds to the line number of the statement.

Consider the read e_8^1 ($R_8(x)$=2). To generate a new seed interleaving, we first try to enforce e_8^1 to read a different value 0 (by e_2 which writes 0 to x). Figure 3.9b shows the corresponding constraints constructed by MCR with our approach for the three different memory models (SC, TSO and PSO). Φ_{state} ensures that $R_8^1(x)$ reads 0 by enforcing e_2 to happen before e_8^1 and e_4 to either happen before e_2 or after e_8^1. $\Phi_{validity}$ ensures that $R_7(z)$ reads the same value as it reads in the trace τ_0.

For SC and TSO, the generated constraint formula is not satisfiable. However, for PSO, the SMT solver returns a solution $\{O_1 = 0, O_2 = 0, O_6 = 1, O_7 = 2, O_8^1 = 3\}$. Based on this solution, we can generate a new seed interleaving to continue with: e_1-e_2-e_6-e_7-e_8^1.

By re-executing the program with this seed interleaving, we can obtain a new trace:

$$\tau_1 = \langle e_1, e_2, e_6, e_7, e_8^1, \mathbf{e_3}, \mathbf{e_4}, \mathbf{e_5}, \mathbf{e_8^2}, \mathbf{e_9^1}, \mathbf{e_9^2} \rangle$$

The events after the seed interleaving are newly explored events. Among these new events there are again three reads: e_8^2 ($R_8(y)$=3), e_9^1 ($R_9(x)$=2) and e_9^2 ($R_9(y)$=3). Since $R_8(x) = 0$ and $R_8(y) = 3$ at line 8, the *if* condition is satisfied and hence the error is triggered. However, we

Table 3.2: Experimental results between our approach and DPOR on the program in Figure 3.9a with N from 1 to 4. The numbers indicate the number of executions explored by each approach. The symbols indicate timeout in one hour ($*$), found the PSO bug (✔), not found the bug (✘), and threw exception (⊖). Reprinted with permission from [2].

Loop times	DPOR				MCR (our approach)			
	SC	TSO	PSO	Error found?	SC	TSO	PSO	Error found?
N=1	4	4	⊖	✘	2	2	10	✔
N=2	105	105	⊖	✘	43	43	89	✔
N=3	4282	4282	⊖	✘	296	296	819	✔
N=4	14840*	14840*	⊖	✘	2767	2767	8420	✔

note that this bug is quite elusive. Before the read $R_9(x)$ at line 9, the buffered write to x has already been flushed to the memory. When the program executes line 9, we have $R_9(x) = 2$ and $R_9(y) = 3$, which contradicts with the *if* condition.

Table 3.2 reports the results comparing our approach with the DPOR algorithm on the number of explored executions and on whether the approach is able to trigger the error under PSO. We set the number of loop iterations from 1 to 4. Our results show that as the number of the loop iterations increases, the number of executions explored by both of the two approaches increases dramatically (2 to 8420 for MCR and 4 to more than 14840 for DPOR). The reason is that the state-space of the program significantly increases as more reads and writes are executed. However, MCR is able to finish exploring the state-space under all memory models in a few seconds, whereas when N=4 DPOR fails to finish the exploration in an hour after exploring 14840 executions under SC and TSO, and under PSO the *rInspect* tool terminates early by throwing an exception (likely due to an implementation bug in the tool). Moreover, our approach takes only 3 executions to trigger the PSO error, whereas DPOR fails to find the error by throwing an exception.

3.5 Evaluation

We have implemented our approach based on the original MCR [19] for multithreaded Java programs with ASM [67] for dynamic bytecode instrumentation and Z3 [68] for constraint solving. We extended MCR from SC to TSO and PSO by relaxing the *must-happen-before* constraints and implementing the TSO and PSO replay algorithms presented in section 4.2. We have evaluated our

Table 3.3: Benchmarks. Reprinted with permission from [2].

Program	LoC	#Thrd	#Evt	Description
Dekker	119	3	56	Two critical sections with 3 shared variables.
Lamport	162	3	40	Two critical sections with 4 variables.
bakery	119	3	27	n critical sections using 2n shared variables. We take n=2.
Peterson	94	3	72	Two critical sections with 3 variables
StackUnsafe	135	3	34	Unsafe operations on a stack by two threads, which cause the stack underflow.
RVExample	79	3	32	An example from original MCR [19], which contains a very tricky error
Example	73	2	44	The example program from Figure 3.9a with loop number from 1 to 4.
Account	373	5	51	Concurrent account deposits and withdrawals suffering from atomicity violations.
Airline	136	6	67	A race condition causing the tickets oversold.
Allocation	348	3	125	An atomicity violation causing the same block allocated or freed twice.
PingPong	388	6	44	The player is set to null by one thread and dereferenced by another throwing NPE.
StringBuf	1339	3	70	An atomicity violation in Java StringBuffer causing StringIndexOutOfBoundsException.
Weblech	35K	3	2045	A tool for downloading websites and enumerating standard web-browser behavior.

approach on a collection of popular benchmarks and real applications shown in Table 3.3. *Dekker*, *Lamport*, *Bakery* and *Peterson* are four classic solutions to mutual exclusion problems from the previous work [36, 66, 69], all of which are intensively racy programs. *StackUnsafe* contains improper operations on the stack collected from [66]. *RVExample* is the motivating example in the original MCR paper [19]. *Example* is the real PSO bug example in Figure 3.6. The other six benchmarks are real programs used in previous concurrency studies [70, 19], including a large application – *Weblech*.

In the rest of this section, we first describe our evaluation methodology and then report our experimental results.

3.5.1 Evaluation Methodology

Our evaluation aims to answer the following three research questions:

1. How effective is our approach for exploring the state-space of concurrent programs?

2. How effective is our approach for finding TSO and PSO errors?

3. How scalable is our approach on real programs?

For the first question, we compared our approach with the most recent development of DPOR by Zhang et al. [66], which extended the original DPOR algorithm [26] with sleep-set reduction for

TSO and PSO. Because their *rInspect* tool is implemented for C/C++, we carefully transformed seven standard benchmarks from Java to C/C++ or in reverse for the comparison.

For the second question, we compared the number of executions needed by different approaches to expose the injected or known errors in each benchmarks. We injected assertion violations in the critical sections of four mutual exclusion programs for different memory models. For those benchmarks with known errors (e.g., *StackUnsafe* and *RVExample*), we directly used those errors for the evaluation. Besides DPOR, we also compared our approach with SATCheck [32], a recent SAT-based stateless model checking approach. SATCheck is a branch-driven approach that aims to cover all branches and all the unknown behaviors of the uninterpreted functions by systematically exploring thread schedules under SC and TSO. However, we found that the SATCheck tool missed executions during testing, especially when the benchmarks become more complicated, e.g., when the program has more conditional paths. Also, since SATCheck runs on C/C++ programs that use primitive reads/writes to access the shared memory, it needs sophisticated instrumentations to identify operations on shared variables, which is done manually in SATCheck. For comparison, we carefully transformed the seven benchmarks to the required format.

For the third question, we tested our approach on the six real programs. We evaluated our approach on these benchmarks under TSO and PSO, in addition to SC which is supported in the original MCR. Because none of the other two tools can support complex real applications and both of them work for C/C++ programs, we were not able to compare our results with the other approaches.

All experiments were conducted on an Apple MacBook Pro machine with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory and Java JDK 1.7. All results were averaged over three runs.

3.5.2 Results of State Space Exploration

Table 3.4 summarizes the results of state-space exploration for the first seven benchmarks in Table 3.3. The first three columns report the results of DPOR, the three columns in the middle report the results of our approach, and the last three columns report the comparison between the two approaches. On average, MCR takes 5X to 10X (as much as 30X) fewer executions than DPOR

Table 3.4: Results of state-space exploration between our approach and DPOR. $*$ means timeout in one hour and $\ominus$ indicates that an exception happened before finishing the experiment. Reprinted with permission from [2].

Program	**DPOR**			**MCR (our approach)**			**SpeedUp**		
	SC	TSO	PSO	SC	TSO	PSO	SC	TSO	PSO
Dekker	248	252	508	62	98	155	4.0X	2.6X	3.3X
Lamport	128	208	2672	14	91	102	9.1X	2.3X	29.4X
Bakery	350	1164	2040	77	158	165	4.5X	7.1X	12.4X
Peterson	36	95	120	13	18	19	2.8X	5.3X	6.3X
StackUnsafe	252	252	252	29	46	108	8.7X	5.5X	2.3X
RVExample	1959	$\ominus$	$\ominus$	57	64	70	34.4X	$\ominus$	$\ominus$
	4	4	$\ominus$	2	2	10	2.0X	2.0X	$\ominus$
Example	105	105	$\ominus$	43	43	89	2.4X	2.4X	$\ominus$
(N = 1 to 4)	4282	4282	$\ominus$	296	296	819	14.5X	14.5X	$\ominus$
	14840*	14840*	$\ominus$	2767	2767	8420	5.4X	5.4X	$\ominus$
Avg.	435	394	1118	42	79	103	10.4X	5.0X	10.9X

to explore the entire state-space. For *RVExample*, which contains a very tricky error with loops, DPOR takes almost 2,000 executions, while MCR only takes 57 executions under SC. Moreover, the *rInspect* tool cannot finish under TSO and PSO by throwing a socket exception. For our *Example* in Figure 3.6, MCR takes 2,767 executions under SC and TSO, and 8,420 executions under PSO. Because the tools are implemented in different languages, it is difficult to compare the runtime speed between them. We hence focused on evaluating the effectiveness of our approach in reducing the number of executions but not the runtime performance. In the original MCR paper [19], it has shown that MCR outperforms DPOR in terms of runtime speed.

3.5.3 Results of Bug Finding

Table 3.5 summarizes the results of the bug finding for the first seven benchmarks in Table 3.3. Overall, our approach takes much fewer executions than the other two approaches for finding the errors. Moreover, our technique is able to find all the known errors and injected assertion violations, whereas DPOR fails to find the errors in *Example* and *RVExample* by throwing exceptions, and SATCheck cannot find any error by either throwing segmentation faults or repeating the same execution forever. Because SATCheck is a branch-driven approach, when there exists a certain

Table 3.5: Results of bug finding between our approach, DPOR and SATCheck. ⊖ indicates the tool failed to run on the benchmark. ! means the tool finished the exploration without finding the bug. $*$ means the tool repeats the same execution and did not terminate. Since SATCheck does not support PSO, we only report its results on SC and TSO. Reprinted with permission from [2].

Program	**DPOR**			**SATCheck**		**MCR (our approach)**		
	SC	TSO	PSO	SC	TSO	SC	TSO	PSO
Dekker	22	28	29	32*	68735*	10	4	5
Lamport	6	8	24	⊖	⊖	2	2	3
Bakery	12	15	15	⊖	⊖	8	8	15
Peterson	4	5	6	19!	34282*	7	2	3
StackUnsafe	6	6	6	⊖	⊖	2	2	2
RVExample	301	⊖	⊖	60564*	70365*	53	54	39
Example	14840!	14840!	⊖	1!	1!	2767!	2767!	3

path that it cannot cover, the tool will run forever, for example in *Dekker*, *Peterson* and *RVExample*. For *RVExample*, DPOR takes 301 executions to find that tricky error, while our approach takes only 53 executions.

3.5.4 Results on Real Programs

Table 3.6 reports the number of executions taken by MCR to explore the state-space of the six real programs under SC, TSO and PSO as well as the number of data races found during the exploration. MCR stops exploration when all state-space of the program has been explored, or it triggers a bug in the program that leads to a runtime exception. Overall, MCR scales well to these real programs, and it is highly effective in exploring the state-space and finding bugs including data races in these programs. For example, for *Account*, MCR took only 7 executions to explore the whole state-space under SC, and 9, 11 under TSO and PSO, and found 3 data races. For *Weblech*, which contains over 2K critical events, MCR finished after explorating 185, 106 and 113 executions, respectively, under SC, TSO and PSO, and found 6 data races. The reason that MCR explored fewer executions under TSO and PSO than that under SC is that bugs in *Weblech* that lead to runtime exceptions are revealed faster under TSO and PSO.

Table 3.6: Results of MCR under SC, TSO and PSO on real programs for state-space exploration and bug finding. Reprinted with permission from [2].

Program	#Executions			# Data Races		
	SC	TSO	PSO	SC	TSO	PSO
Account	7	12	12	3	3	3
Airline	8	11	11	0	0	0
StringBuf	3	3	3	0	0	0
Allocation	30	30	30	0	0	0
PingPong	411	483	527	7	7	7
Weblech	178	103	116	6	6	6

3.6 Summary

We have presented an extension of MCR for stateless model checking of concurrent programs under TSO and PSO. Our approach solves two key technical challenges. First, how to generate new unique interleavings by formulating the operational semantics of TSO and PSO as first-order logical constraints. Second, how to deterministically execute the program following the generated TSO and PSO interleavings. By relaxing the must happen-before constraints in MCR to allow TSO and PSO reorderings, and by developing novel replay algorithms that allow executions out of program order, our approach enables MCR to effectively verify concurrent programs for TSO and PSO. We have also presented our experimental results of applying MCR on both popular benchmarks and real applications and comparing MCR with DPOR and SATCheck. Our results show that our approach is much more effective than the other approaches for both state-space exploration and bug finding.

4. MCR-S: SPEEDING UP MCR WITH STATIC DEPENDENCY ANALYSIS *

Maximal Causality Reduction (MCR) 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 3.1.2) 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. To ensure the reachability of the considered read r, MCR enforces the reads that happen before r to return the same value (3.1.2). For example, if MCR considers the second read r2 in the trace, it will enforce the first read r1 to return the same

*Reprinted with permission from "Speeding Up Maximal Causality Reduction with Static Dependency Analysis" by Shiyou Huang and Jeff Huang, 2017. 31st European Conference on Object-Oriented Programming (ECOOP), 74, 16:1-16:22.

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 MCR-S 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 4.2.3. 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.

4.1 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 represent the dependencies between the nodes [71]. Figure 4.1 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

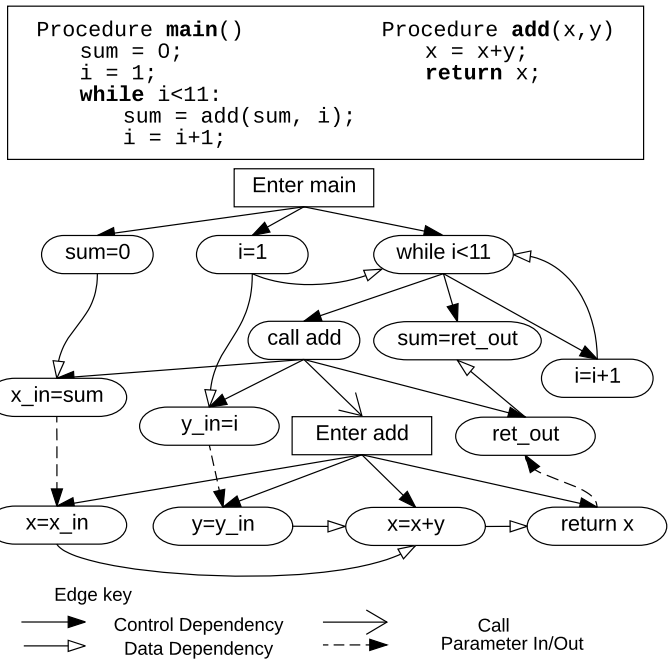


Figure 4.1: The System Dependency Graph of a concrete program, where the dependencies are distinguished by different edges. Reprinted with permission from [1].

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 4.1, 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.

Algorithm 4: $\Phi_{validity}(e)$ 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** $\Phi_{validity}$

4.2 MCR with Static Dependency Analysis

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.

4.2.1 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 4.2.1. 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 4 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} introduced in Section 3.1.2. 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 4.2.2.3. 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 3.1.2.

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 3.1.2 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.

4.2.2 Dependency Analysis

In this subsection, we present how we compute $\prec^D_\tau (e)$ 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* [72, 73] 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*.

4.2.2.1 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

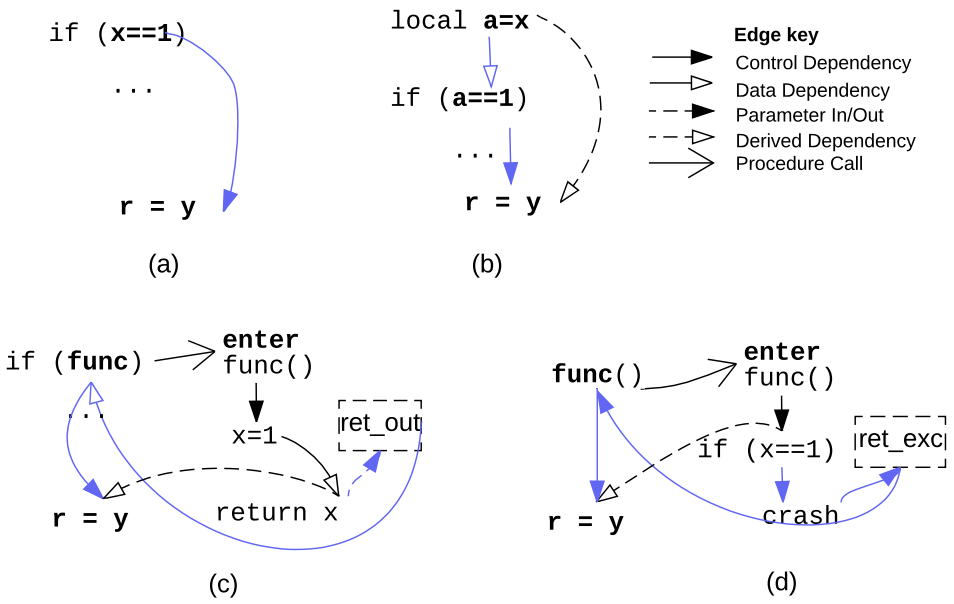


Figure 4.2: Four different cases where a read is control dependent on another marked by the blue edges. Reprinted with permission from [1].

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 4.2.2. 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 4.2 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 4.2(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 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 4.2(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 4.2.2, $a = x$ is control dependent on $r = y$ and $x == 1 \xrightarrow{DD \cdot DD \cdot CD} r = y$

Case 3. Figure 4.2(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 4.3 to formalize this process.

4.2.2.2 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

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

Figure 4.3: Rule 1: the condition that a node has control dependency on another in SDG. Reprinted with permission from [1].

data validity constraints in Section 3.1.2 also need to guarantee the reachability of *w*. We have illustrated in Section 4.2.2.1 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 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 4.3.

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

Figure 4.4: Rule 2: the condition that a node has data dependency on another in SDG. Reprinted with permission from [1].

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 4.2.2.1.

4.2.2.3 Dependency Reads Computation

After the discussion about the *control* and *data dependency*, we now present the algorithm of the function **DependencyComputation()** in Algorithm 4 to give the details about how to compute the set of reads that an event is dependent on in the program.

Algorithm 5 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 $\prec_\tau^D (e)$.

4.2.3 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 precise points-to analysis has to approximate certain undecidable situations which lead to may-alias. Due to the limitations

Algorithm 5: 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) ;
```

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 $\prec^D_\tau\ (e)$ 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.

4.3 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 4.1: An example that shows redundant explorations by our approach.

Consider the example above. 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.

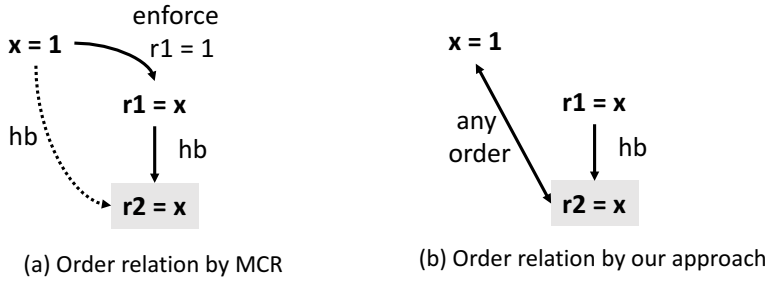


Figure 4.5: Removed happens-before between $x = 1$ and $r2 = x$ by our approach. Reprinted with permission from [1].

Figure 4.5 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 4.5(b), $x = 1$ and $r2 = x$ can be in any order by our approach, while $x = 1$ happens before $r2 = x$ in MCR.

4.3.1 Redundancy Elimination

According to the analysis on the example presented in Listing 4.1, 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 4. We present our algorithm as follows.

The only difference between Algorithm 6 and Algorithm 4 lies in *line 3*. In our new algorithm, we decide whether to add the reads that happen before e to $\prec_\tau^D (e)$ 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 $\prec_\tau^D (e)$ to avoid the redundant behavior. For the example, in Listing 4.1, as we want to explore what values $r2(x)$ can read, we also put $r1(x)$ into $\prec_\tau^D (e)$ 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 4. Although this expands

Algorithm 6: DataValidityConstraints′(τ, e)

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

$\Phi_{validity} = \emptyset$
$\prec_\tau (e) \leftarrow$ **Happens-before**(τ, e)
`// target read: read considered to return new values`
if e *is not a* TARGET READ **then**
  $\prec_\tau^D (e) \leftarrow$ **DependencyComputation**$(\prec_\tau (e), e)$
end
foreach *read* $r \in \prec_\tau^D (e)$ *with value* v **do**
  `// `$\Phi_{value}(r, v)$` recursively call DataValidityConstraints()`
  $\Phi_{validity} \wedge = \Phi_{value}(r, v)$
end
return $\Phi_{validity}$

$\prec_\tau^D (e)$ 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 4.4.

Algorithm 6 can remove all the redundancies caused by Algorithm 4, and it will not miss any executions.

Proof. The proof on the latter part follows the same analysis on Algorithm 4 in Section 4.2.1. To prove that Algorithm 3 reduces all the redundancies, we show that by using Algorithm 6, 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.

4.4 Implementation and Evaluation

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

4.4.1 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* [74] and *Soot* [75] for Java programs. In this work, we build the SDG of Java programs based on two existing framework, *JOANA* [72, 73] 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 4.3 or 4.4.

4.4.2 Methodology

In the rest of this section, we refer to as MCR-S and MCR-S+ the approach that implements Algorithm 4 and 6, 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 explo-

ration?

In Section 4.4.3, 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 4.4.4, 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 [19] so that we can make a direct comparison. Table 4.1 summarizes the benchmarks evaluated in this work. **Counter** is the example introduced in the beginning of this chapter, 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.

4.4.3 Reduction Analysis

Table 4.2 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

Table 4.1: Benchmarks. Reprinted with permission from [1].

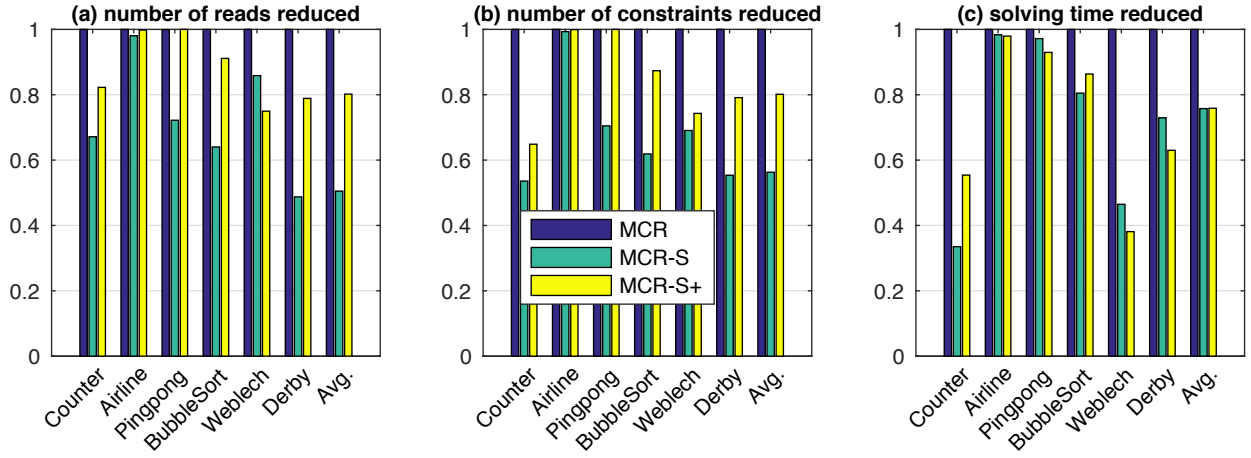
Program	time(s)	memory(M)	#nodes	#edges
Counter	2.00	69	289	1,440
Airline	2.10	79	809	4,902
Pingpong	2.52	83	914	5,244
BubbleSort	2.14	81	911	5,710
Pool	3.67	75	2,848	17,586
StringBuf	2.96	111	2,129	12,310
Weblech	8.01	219	22,094	167,492
Derby	69.67	1,385	115,658	2,409,784

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 4.6 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 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 4.6(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 4.3). 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 4.2, 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%.

Figure 4.6: 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. Reprinted with permission from [1].



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.

Table 4.2: 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. Reprinted with permission from [1].

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.

Table 4.3: The total number of executions and time taken by the three methods to explore the state-space of the benchmarks. Reprinted with permission from [1].

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

Number of constraints reduced. Figure 4.6(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 4.6(b) shows, the reduction on the constraints is consistent with that on the reads in Figure 4.6(a).

Solving time reduced. Figure 4.6(c) presents the results of the solving time by each method. From Figure 4.6(b) and (c), we can see that though MCR-S approximately makes two times as much constraints reduction 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.

4.4.4 Overall Checking Performance Comparison

Table 4.3 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 4.3, 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.

4.5 Summary

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.

5. SE-MCR: REDUCE THE STATE SPACE OF MCR USING SWITCH EQUIVALENCE

Stateless Model Checkers suffer from state explosion problem. As a result, the reduction technique behind a model checker is critical to make the model checkers efficient and more scalable. MCR uses SMT constraints to reason about the maximal causality of a given execution trace and partitions the traces based on the values seen by the read events in each trace. Compared with DPOR, MCR reduces the number of executions 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.

Although MCR gains a great performance improvement over the POR based approaches, it does not achieve the minimum number of explored interleavings. Consider four threads, p, q, r and s, performing read and write accesses to shared variables x and y:

$$p{:}\ \textit{write}\ x; \quad q{:}\ \textit{read}\ x; \quad r{:}\ \textit{write}\ y; \quad s{:}\ \textit{read}\ y;$$

Given the execution trace $p.q.r.s$, MCR considers all the reads and tries to make them return different values. Suppose it considers q first and then s, the execution $\boldsymbol{q.s}.p.r$ can be generated. If it considers s first and then q, $\boldsymbol{s.q}.p.r$ is explored. However, these two executions produce the same output. The fundamental reason for this is that after MCR computes two interleaving prefixes $\{q,s\}$ from the trace $p.q.r.s$ to make the reads happen before the writes, it assumes that the explorations starting with q and s are independent and will not lead to redundant executions. But executions starting with $q.s$ and $s.q$ are actually equivalent to each other. We call such equivalence as ***switch*** equivalence. Even worse, the future exploration along $q.s$ and $s.q$ will also be equivalent, which causes a performance loss to MCR. In this cahpter, I first illustrate how MCR explores redundant interleavings with a concrete example and then present the new algorithm, called *SE-MCR*, applying the switch equivalence checking to the MCR approach.

5.1 MCR Explores Redundant Interleavings

MCR reduces the number of explored interleavings by DPOR and *iterative context bounding* (ICB) [20] by orders of magnitude. In this section, we use the program in Figure 3.2 as an example to interpret how MCR works, and then show that MCR is limited to redundant executions. The program contains four threads *p,q,r* and *s*, and two shared variables x and y. Figure 5.1 shows the exploration process by MCR. In this paper, for ease of presentation, we use the thread label (*e.g.*, *p*) to refer to the corresponding event. MCR uses integer variables $\langle O_1, \cdots, O_n \rangle$ to denote the order in which the events happen in a certain execution. The value of O_p represents the position of the event p in a trace. If $O_p < O_q$, then p will be executed before q in the generated interleaving.

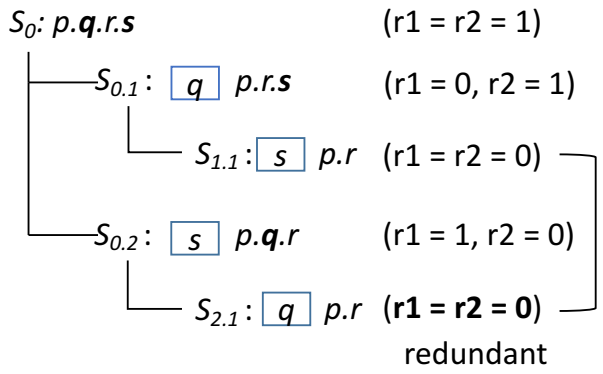


Figure 5.1: The exploration for the program in Figure 3.2 by MCR. The prefix is wrapped by a box. A trace is collected by re-executing the program starting with the prefix, and bold letters corresponds to read events in the trace.

Initially, suppose the program is executed in the order *p.q.r.s* and reaches the initial state S_0, where $r1 = 1$ and $r2 = 1$,

$$S_0 : p.q.r.s \quad \{r1 = 1, r2 = 1\}.$$

To explore new possible execution interleavings from the trace *p.**q**.r.**s***, MCR enumerates all the reads in the trace, *e.g.*, $r1 = x$ and $r2 = y$ by threads q and s (marked as bold in the figure), respectively. In S_0, MCR detects that *q* reads from *p*, and the program can generate a new state if *q*

reads from the initial value, implying that *q* happens before *p*. MCR generates $\Phi_{state} = O_q < O_p$ to encode the order. Therefore, MCR generates a new schedule prefix, $\boxed{\text{q}}$, wrapped by a box in the figure. Likewise, to make *s* read from the initial value, MCR generates another prefix $\boxed{\text{s}}$ from the constraints $O_s < O_r$. Then by running the program starting with $\boxed{\text{q}}$, a new trace *p.r.s* is collected, leading to state

$$S_{0.1} : q.p.r.s \quad \{r1 = 0, r2 = 1\}.$$

Similarly, the program reaches the state

$$S_{0.2} : s.p.q.r \quad \{r1 = 1, r2 = 0\}$$

by the execution along $\boxed{\text{s}}$. Using the same strategy on trace *p.r.s* in $S_{0.1}$ and *p.q.r* in $S_{0.2}$, MCR generates two more prefixes *s* and *q*, respectively. By re-executing the program starting with the two prefixes, two new states

$$S_{1.1} : q.s.p.r \quad \{r1 = 0, r2 = 0\}$$

and

$$S_{2.1} : s.q.p.r \quad \{r1 = 0, r2 = 0\}$$

are generated.

Figure 5.2 shows the reduction made by MCR. The interleaving highlighted in blue are executed by MCR, while the dashed are redundant executions identified by MCR. The number of explored interleavings is reduced from 24 to 5. However, as we can see, states $S_{1.1}$ and $S_{2.1}$ are equivalent to each other, both of which making $r1 = r2 = 0$. MCR fails to identify $\boxed{\text{q}}.\boxed{\text{s}}$ and $\boxed{\text{s}}.\boxed{\text{q}}$ as redundant executions because when MCR first computes the prefixes $\boxed{\text{q}}$ by $S_{0.1}$ and $\boxed{\text{s}}$ by $S_{0.2}$, it takes $\boxed{\text{q}}$ and $\boxed{\text{s}}$ as two totally independent prefixes. MCR then assumes that the sub-state spaces of the program starting with these two prefixes will not overlap, meaning that executions from different sub-space (*i.e.*, starting with different prefixes) cannot generate the same

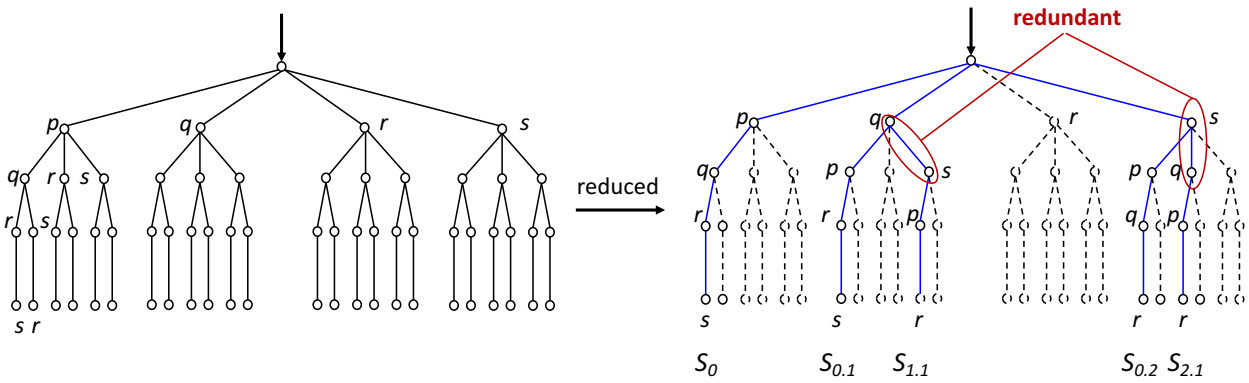


Figure 5.2: State-space reduced by MCR.

state. The redundancy problem can harm the efficiency of the model checker severely because the explorations along two redundant prefixes are equivalent to each other.

Limitation To avoid generating duplicated seed interleavings from different τ, MCR ensures that the prefix of each new explored interleaving is always preserved. Figure 5.1 shows an intuitive view of this process. From state $S_0 : p.q.r.s$, MCR generates two prefixes q and s. Let us take q as an example. By executing the program starting with q, MCR collects the trace p, r, s. Note that the events corresponding to the prefix are not included in the trace any more. A new prefix s can be generated from $p.r.s$. Since the new prefix s is generated under the sub-space by q, MCR combines s with q (*i.e.*, $q.s$) to guide the new execution. Obviously, given a trace τ collected by executing the program along a prefix $\mathcal{P}$, no two prefixes generated from τ are equivalent. Suppose any two prefixes $\mathcal{P}_1$ and $\mathcal{P}_2$, derived from τ. As each prefix enforces only one read to return a different value, assume that $\mathcal{P}_1$ targets read r_1 and $\mathcal{P}_2$ targets r_2. Because both of $\mathcal{P}_1$ and $\mathcal{P}_2$ are derived under $\mathcal{P}$, prefixes $\mathcal{P}.\mathcal{P}_1$ and $\mathcal{P}.\mathcal{P}_2$ must guide the program to generate different states.

However, as Figure 5.1 shows, MCR explores redundant states, $q.s.p.r$ and $s.q.p.r$. Let us assume that from $\mathcal{P}_1$ (here corresponds to q), MCR can further generate prefix $\mathcal{P}'_1$ (s) and from $\mathcal{P}_2$ (s), MCR generates $\mathcal{P}'_2$ (q). Then there is no guarantee that $\mathcal{P}.\mathcal{P}_1.\mathcal{P}'_1$ is different from $\mathcal{P}.\mathcal{P}_2.\mathcal{P}'_2$, where $\mathcal{P}$ is the prefix of $\mathcal{P}_1$ and $\mathcal{P}_2$. Because $\mathcal{P}'_1$ and $\mathcal{P}'_2$ may target different reads. Figure 5.2 shows where the redundancy happens in the reduced space by MCR for the example in Figure 3.2.

Initially x := y := 0			
p: *x := 1*	*q:* *r1 := x*	*r:* *y := 1*	*s:* *x := 2* *r2 := y*

Figure 5.3: An example that shows swapping the order of two seed prefixes does not always work.

In this work, we also present an optimized MCR and show that the new algorithm can explore a provable minimum number of interleavings.

5.2 Solution Overview

In this section, we present the basic idea for our solution. To identify the redundancy, we check if there exists such a situation, (1) there is an execution sequence that makes the program produce a state S, and MCR explores another state S' following S; (2) there exists an alternative execution sequence that makes the program produce S' first, and then MCR explores another state S following S'. If such a situation exists, MCR potentially explores redundant executions.

Consider the exploration in Figure 5.1. After we collect the trace from the execution S_0 : $p.q.r.s$, we compute two prefixes $\boxed{\text{q}}$ and $\boxed{\text{s}}$ in the same way as MCR does. Then our approach checks if the combination of $\boxed{\text{q}}$ and $\boxed{\text{s}}$ in a different order will lead to an equivalent execution. We find that if the program reaches state $S_{0.1}$ along $\boxed{\text{q}}$ first, the model checker will explore the prefix $\boxed{\text{s}}$ next. If the program reaches $S_{0.2}$ along $\boxed{\text{s}}$ first, $\boxed{\text{q}}$ will be considered next. Our approach detects that $\boxed{\text{q}}.\boxed{\text{s}}$ and $\boxed{\text{s}}.\boxed{\text{q}}$ make the program produce the same outcome statically (*i.e.*, without execution). Then when we reach state $S_{0.1}$ by $\boxed{\text{q}}.p.r.s$ we can skip the prefix $\boxed{\text{s}}$ generated from the trace $p.r.s$, and state $S_{1.1}$ is avoided.

Challenges 1 Swapping the order of two seed prefixes does not always generate the equivalent execution sequence. Take the program in Figure 5.3 as an example. This program slightly modifies the example in Figure 3.2 by adding a write $x := 2$ before $r2 := y$ in thread s. Figure 5.4 shows the exploration process of this program. The redundancy occurs between the executions starting with

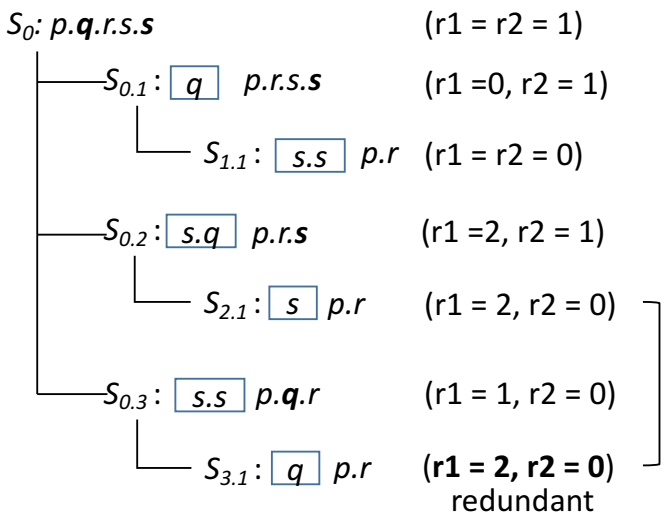


Figure 5.4: The exploration for the program in Figure 5.3 by MCR.

$S_{0.2}$: s.q s and $S_{0.3}$: s.s q, both of which drive the program to generate $r1 = 2, r2 = 0$. If we check the equivalency by just reordering the prefixes, the equivalent execution sequence should be $S_{0.1}$: q.s.s and $S_{0.3}$: s.s.q, because one can be transformed to the other by just swapping q and s.s. But obviously, q.s.s and s.s.q are not equivalent to each other because the former drives the program to the state $r1 = r2 = 0$, while the latter $r1 = 2, r2 = 0$. We explain how to check the equivalence of two prefixes computed from the same trace in next section.

Initially x := y := 0			
p: *x := 1*	*q:* *lock(l)* *r1 := x* *unlock(l)*	*r:* *y := 1*	*s:* *lock(l)* *r2 := y* *unlock(l)*

Figure 5.5: An example with locks.

Challenges 2 The second challenge to identify a redundant interleaving is to guarantee the execution is feasible. Consider an example in Figure 5.5, a variation of the program in Figure 3.2. The

only difference is that the two reads shown in Figure 5.5 is protected by a lock. For the program in Figure 5.5, suppose initially the program is executed in the order *p.q.q.q.r.s.s.s* and reaches the initial state S_0, where $r1 = 1$ and $r2 = 1$,

$$S_0 : p.q.r.s \quad \{r1 = 1, r2 = 1\}.$$

To make *x* and *y* read from the initial writes, the first two prefixes computed from the initial execution become $q.q$ and $s.s$ because the *lock(l)* happens before the read. If we directly combine the two prefixes, it could be $q.q.s.s$. However, this execution sequence is not feasible because $s : lock(l)$ should happen after $q : unlock(l)$. As a result, we need to add events to the prefix until it contains the unlock event, *e.g.*, $q.q$ is extended to $q.q.q$ and $s.s$ is extended to $s.s.s$ In our implementation, we model a `wait` event as a `wait-lock` pair and handle it in the same way as we handle the unpaired `lock` event here.

Challenges 3 Moreover, it requires a theoretical proof that the alternative execution (generated by swapping the order of two seed interleavings) is always valid and removing the redundant interleavings does not harm the completeness of the exploration.

We present how we address these challenges in next section.

5.3 Maximal Causality Reduction across Equivalent Interleavings

In this section, we present a new algorithm, *SE-MCR*, to address the redundant issues in MCR and a proof that shows our algorithm will not miss any interleavings. *SE-MCR*extends the original MCR to check the redundancy across all seed interleavings generated by MCR with switch equivalence. Given a random trace τ_0, the algorithm enumerates all the reads on it. For each read, the algorithm checks all the different writes that this read can read from, and generates a causal prefix $\mathcal{P}$ to enforce the order of the events in the program to make a specific read to read from that write. Among the prefixes that are computed from τ_0, the algorithm checks if any two of them make a redundant execution by altering their order. Section 5.3.1 introduces the new algorithm, Section 5.3.2 shows the advantages of the new algorithm over MCR and Section 5.3.3 formally proves the

correctness of this algorithm.

5.3.1 MCR with Switch Equivalence

Algorithm 7: *SE-MCR* algorithm

```
1  𝒫 = {𝒫_0} // 𝒫: a set of unexplored prefixes
2  ℰ[𝒫_0] = ∅ // ℰ: a map from a prefix to its equivalent prefixes
3  foreach 𝒫 ∈ 𝒫 do
4  |  Explore(𝒫)
5  end
6  Function Explore(𝒫):
7  |  τ ← Execute(𝒫)
8  |  Φ ← ConstraintsModel(τ)
9  |  ℙ ← SearchNewPrefixes(τ,Φ)
10 |  CheckEquivalence(ℙ, ℰ)
11 |  foreach 𝒫_new ∈ ℙ do
12 |  |  if 𝒫_new ∉ ℰ[𝒫] then
13 |  |  |  add 𝒫_new to 𝒫
14 |  |  end
15 |  end
```

The*SE-MCR* is presented in Algorithm 7. We use $\mathscr{P}$ to maintain all the prefixes that have not been explored by the algorithm. A map denoted as $\mathcal{E}$ records the equivalent prefixes of a prefix $\mathcal{P}$. Initially, $\mathscr{P}$ only contains $\mathcal{P}_0$ that executes the program in the program order thread by thread. The algorithm terminates when all the prefixes in $\mathscr{P}$ are explored (lines $3-5$).

Whenever the program is executed along the prefix, a trace denoted as τ is collected to record the new generate events (line 7). Then a maximal causal model is constructed over the trace (line 8), encoding all the possible executing sequences that can be derived from this trace. The model is constructed by $\Phi^{mcm} = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{validity}$ (3.1.2). In line 9, the algorithm searches new prefixes that drive the program to reach different states by calling **SearchNewPrefixes**. For all the new computed prefixes, the algorithm checks if either two of them lead to an equivalent execution by calling **CheckEquivalence** (line 10). Then for each new prefix $\mathcal{P}_{new}$, if it is not in the equivalent

prefixes of $\mathcal{P}$, the algorithm adds it to $\mathscr{P}$ (lines $11-15$).

5.3.1.1 Search New Prefixes

Algorithm 8: SearchNewPrefixes(τ,Φ)

```
   Return: a set of prefixes ℙ
1  Function SearchNewPrefixes(τ,Φ):
2      ℙ = ∅
3      foreach r = read(t, x, v) ∈ τ do
4          foreach w = write(−, x, v′) ∈ τ ∧ v′ ≠ v do
5              Φ_P ← NewPrefixesConstraints(r, w, Φ)
6              P ← SolveConstraints(Φ_P)
7              add P to ℙ
8          end
9      end
10     return ℙ
```

The new prefix searching algorithm is presented in Algorithm 8. The key insight behind this algorithm is taking the values of reads and writes into consideration. It enforces a read event to read a different value from that in the given trace τ. For each read $r = read(t, x, v) \in \tau$, it enumerates each write $w = write(-, x, v') \in \tau$ (from any thread) that writes a different value from that returned by r (lines $3-4$). Function **NewPrefixesConstraints** constructs constraints to identify if r can read from w (line 5). Basically, it enforces other writes (*e.g.*, $w' = write(-, x, v''), v'' \neq v'$) to either happen before w or after r, recalling that $\Phi_{value}(r, v)$ makes the read r to read a value v. The following constraint defines how to compute $\Phi_{\mathcal{P}}$.

$$\Phi_{\mathcal{P}} \equiv \Phi_{lock} \wedge \Phi_{mhb} \wedge \Phi_{validity}(r) \wedge \Phi_{validity}(w) \wedge \Phi_{value}(r, v')$$

For each $\Phi_{\mathcal{P}}$, we invoke a constraint solver. If the solver returns a solution, the solution represents a new interleaving that is feasible and in which the read will read that new value. We construct a new prefix from the solution and add it to $\mathbb{P}$ (lines $6-7$). 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 6.1.

5.3.1.2 Check Equivalence across New Prefixes

Algorithm 9: Switch Equivalence($\mathbb{P}$, $\mathcal{E}$)

```
1  Function CheckEquivalence(ℙ, ℰ):
2      n = ℙ.size()
3      for i = 0 to n − 2 do
4          for j = i + 1 to n − 1 do
5              P_i = ℙ[i], P_j = ℙ[j]
6              P = Equivalent (P_i, P_j)
7              if P ≠ ε then
8                  add P to ℰ[P_i]
9              end
10         end
11     end
12 Function Equivalent(P_i, P_j):
13     P = P_ij = P_ji = ε
14     r_i = last(P_i), r_j = last(P_j) // must be a read
15     if r_i ↛ r_j and r_j ↛ r_i then
16         P_ij = P_i◇P_j, P_ji = P_j◇P_i
           // P_ij and P_ji lead to the same state
17         if P_ij ≃ P_ji then
18             P = P_ij.remove(P_i)
19         end
20     end
21     return P;
```

Algorithm 9 checks if two prefixes in $\mathbb{P}$ generated by Algorithm 8 can lead to equivalent executions. Given a scheduling prefix $\mathcal{P}$, we use the following notations to help illustrate the algorithm:

- $last(\mathcal{P})$ refers to the last schedule choice of $\mathcal{P}$;
- $\mathcal{P}_i.remove(\mathcal{P}_j)$ removes from $\mathcal{P}_i$ the common elements of $\mathcal{P}_i$ and $\mathcal{P}_j$;

- $\mathcal{P}_i.\mathcal{P}_j$ appends $\mathcal{P}_j$ to $\mathcal{P}_i$;

- $\rightarrow$ refers to the happens before relation.

For any two prefixes $\mathcal{P}_i$ and $\mathcal{P}_j$, the algorithm invokes **Equivalent** to search the equivalent prefix (lines $3-6$). However, before checking if two prefixes are equivalent or not, it first needs to extend the prefix if it contains an unpaired *lock* event as stated in Section 5.2. To detect if the a prefix contains an unpaired *lock*, we traverse the events in the order specified by the prefix with a stack. If we encounter a *lock* event, we push it to the stack; if an *unlock* event is met, we pop the unpaired *lock* from the stack. At last, we check if the stack still contains any events. If so, it means the prefix contains an unpaired *lock* event. To extend the prefix, we first identify the thread of that unpaired *lock*, then adding events from that thread to the prefix until it meets the *unlock* event.

Switch Equivalence: To judge if two prefixes $\mathcal{P}_i$ and $\mathcal{P}_j$ will yield reduandant executions, we introduce *switch equivalent*, which is depicted as the following two conditions: ❶ the last events r_i in $\mathcal{P}_i$ and r_j in $\mathcal{P}_j$ do not happen before each other(line 15); ❷ $\mathcal{P}_{ij}$ and $\mathcal{P}_{ji}$ make r_i and r_j see the same value as that in $\mathcal{P}_i$ and $\mathcal{P}_j$, respectively. The computation of $\mathcal{P}_{ij}$ is defined as follows.

Definition 5.3.1 ($\mathcal{P}_{ij} = \mathcal{P}_i \diamondsuit \mathcal{P}_j$)**.** $\mathcal{P}_i \diamondsuit \mathcal{P}_j = \mathcal{P}_i.(\mathcal{P}_j.remove(\mathcal{P}_i))$.

We use the following definitions to describe two equivalent execution sequences.

Definition 5.3.2 ($\mathcal{P}_x^v$)**.** $\mathcal{P}_x^v$ denotes an execution sequence in which a read access to a shared memory location x returns the value v.

Definition 5.3.3 ($\simeq$)**.** Given $\mathcal{P}_x^{v_1}$ and $\mathcal{P}_y^{v_2}$, let $\mathcal{P} = \mathcal{P}_x^{v_1} \diamondsuit \mathcal{P}_y^{v_2}$ and $\mathcal{P}' = \mathcal{P}_y^{v_2} \diamondsuit \mathcal{P}_x^{v_1}$. $\mathcal{P}$ is equivalent to another $\mathcal{P}'$ (denoted as $\mathcal{P} \simeq \mathcal{P}'$) iff, for the read accesses to x and y, denoted as r_x and r_y, in $\mathcal{P}$ and $\mathcal{P}'$, the values returned by them, $value(r_x)$ and $value(r_y)$, should equal to $v1$ and $v2$, respectively;

Definition 5.3.3 means that the combination of two prefixes should make the reads return the same value as they do in the prefix they belong.

Initially x := y := z := 0					
p: *r1 := x*	*q:* *r2 := y*	*r:* *r3 := z*	*s:* *x := 1*	*t:* *y := 1*	*u:* *z := 1*

Figure 5.6: An example shows the efficiency of *SE-MCR*.

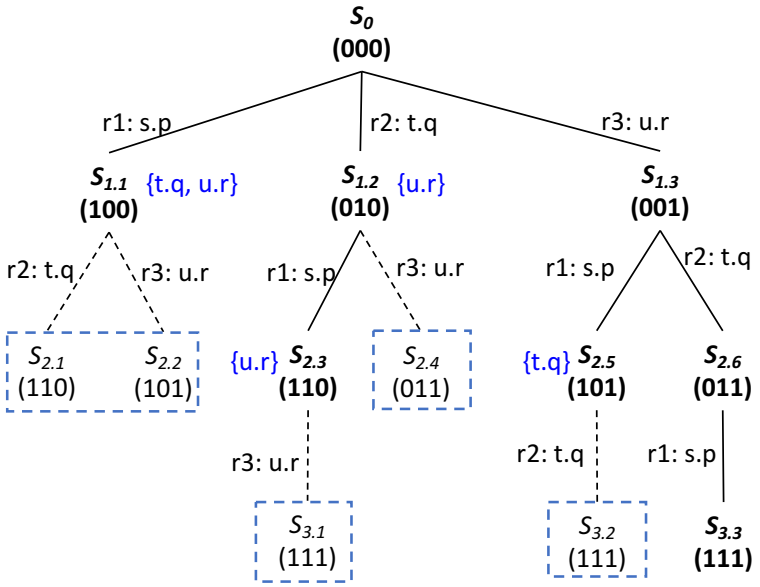


Figure 5.7: State-space exploration for the program in Figure 5.6 by *SE-MCR*.

To illustrate Algorithm 9, consider the exploration in Figure 5.3. Given the trace $S_0 : p.q.r.s.s$, Algorithm 8 generates three causally different prefixes: $\boxed{\text{q}}$, $\boxed{\text{s.q}}$ and $\boxed{\text{s.s}}$. For prefixes $\mathcal{P}_i = \boxed{\text{s.q}}$ and $\mathcal{P}_j = \boxed{\text{s.s}}$, their last events are $r1 := x$ and $r2 := y$, which do not happen before each other. Then Algorithm 9 computes $\mathcal{P}_i \diamondsuit \mathcal{P}_j = s.q.s$ and $\mathcal{P}_j \diamondsuit \mathcal{P}_i = s.s.q$. Since both of them make $r1 = 2$ and $r2 = 0$, the algorithm adds the execution sequence $\mathcal{P}_{ij}.remove(\mathcal{P}_i) = s$ (it will be generated from the execution trace starting with $\mathcal{P}_i$) to $\mathcal{P}_i$'s equivalent prefix set, $\mathcal{E}(\mathcal{P}_i)$. When the algorithm considers the trace $p.r.s$ in $S_{0.2}$, it ignores the new prefix s so that it avoids the execution $S_{2.1} : s.p.r$.

5.3.2 Efficiency

In this section, we use the program in Figure 5.6 to show the efficiency of *SE-MCR* in reducing the size of the state space and that our algorithm is intuitively correct. The exploration of the state space is abstracted as the process of traversing a tree presented in Figure 5.7. Each node represents a program's state. For instance, assume that the program is executed in the order *p.q.r.s.t.u* and reaches the state $S_0(000)$, meaning that the values of $r1, r2$ and $r3$ are all 0. For each state, it is assigned with prefixes in the braces, marking that executions beginning with them from this state will be redundant. Each edge is labeled with the target read and the corresponding prefix to make that read return a different value. Take the out edge *r1: s.p* from S_0 as an example. It enforces $r1 = 1$ by executing the program starting with `s.p`. The dashed edges refers to the redundant execution starting with the prefix on the edge. For instance, the execution starting with *r2: t.q* out from $S_{1.1}$ is equivalent to that along *r1: s.p* out from $S_{1.2}$. As a result, our algorithm only executes one of them. All the states in the dashed box are redundant states that are generated by MCR but not by our algorithm. In total, *SE-MCR* explores all the eight unique states that can be produced by the program in Figure 5.6, from $000, 100, ...$ to $011, 111$. However, MCR explores 16 states in total, half of which are actually redundant.

As is shown in Figure 5.7, our searching starts with an initial state (*e.g.*, S_0). For each state, all possible distinct execution sequences are computed. For instance, our algorithm computes prefixes $s.p$, $t.q$ and $u.r$ from S_0 in this example. By re-executing the program starting with the prefix, the program reaches different states, *e.g.*, $S_{1.1}$ produced by the execution along $s.p$. The key insight for *SE-MCR* to avoid redundancy is to **make each state aware of its sibling's states**. For example, when the program reaches state $S_{1.1}$, it knows that its sibling $S_{1.2}$'s state is 010 and $S_{1.2}$ next will reach 110 by attempting to change the value of $r1$ from 0 to 1. Knowing that *s.p.t.q* is equivalent to *t.q.s.p*, $S_{1.1}$ will ignore the prefix *r2: t.q* generated from the execution starting with *r1: s.p*.

When two states at the same level do not influence each other (*i.e.*, no *happens-before* relation between the target reads), they are interchangeable in the trace, and a redundant execution is generated by swapping the exploration order of the two states. Algorithm 9 illustrates when redundant

execution sequences can happen. Since each state on the same level of the tree comes from the same prefix, exploring only one of the two sequences would not influence the completeness. So our algorithm does not miss any causally different execution sequences. In the next two subsections, we formally prove the feasibility of interchanging two states at the same level in the tree, and optimality of Algorithm 9.

5.3.3 Correctness

To prove the correctness of Algorithm 9, we first show why $\mathcal{P}_i \diamondsuit \mathcal{P}_j.remove(\mathcal{P}_i)$ must be explored after the program is re-executed along $\mathcal{P}_i$. Then we prove that our algorithm achieves completeness.

5.3.3.1 Feasibility

We prove that at an exploration point, the program may reach state A or B, and if the program reaches either of the states, it will next explore the other. Given a prefix $\mathcal{P}$, we first give the following notations:

- $dom(\mathcal{P})$ means $\{1, 2, ..., \mathcal{P}.len()\}$;
- $\mathcal{P}^k$ refers to the kth event in $\mathcal{P}$.

Given a trace τ, we compute two prefixes: $\mathcal{P}_A$ represents the execution that makes program reach state A and $\mathcal{P}_B$ for state B. We use $r_A = last(\mathcal{P}_A)$ and $r_B = last(\mathcal{P}_B)$. For a prefix $\mathcal{P}$ computed from τ, it has two properties:

Property 1. $\forall i \in dom(\mathcal{P})$, $\mathcal{P}^i \rightarrow last(\mathcal{P})$; $\forall e \in \tau$, if $e \rightarrow last(\mathcal{P})$, $e \in \mathcal{P}$.

Property 2. $\forall r \in \mathcal{P}, r \neq last(\mathcal{P})$, r returns the same value as that returned by the same read in τ.

Lemma 4. If $r_B \nrightarrow r_A$, r_B must be explored after enforcing r_A to see a different value, and vice versa.

Proof. Suppose $\mathcal{P}_A$ enforces r_A to see a different value. Since $r_B \nrightarrow r_A$, $r_B \notin \mathcal{P}_A$ (*Property 1*). Therefore, r_B must occur in the trace after executing the program along $\mathcal{P}_A$. Our algorithm then will explore a different value that r_B can see. □

Lemma 5. If $e \in \mathcal{P}_B$, and $e \notin \mathcal{P}_A$, e must happen after r_A and before r_B, and vice versa.

Proof. $e \rightarrow r_B$ due to *Property 1*. If $e \rightarrow r_A$, then $e \in \mathcal{P}_A$, which contradicts with the condition $e \notin \mathcal{P}_A$. □

Theorem 6. Given a state S_0, and S_0 contains two reads r_A and r_B. If there exist

1. two different prefixes $\mathcal{P}_A$ and $\mathcal{P}_B$, making that $S_0 \overset{\mathcal{P}_A}{\rightsquigarrow} S_A$, r_A reads from a write w_A that writes a different value; and $S_0 \overset{\mathcal{P}_B}{\rightsquigarrow} S_B$, r_B reads from a write w_B that writes a different value;

2. $r_A \notin \mathcal{P}_B$, and $r_B \notin \mathcal{P}_A$,

then $S_0 \overset{\mathcal{P}_A}{\rightsquigarrow} S_A \overset{\mathcal{P}'_B}{\rightsquigarrow} S$ is feasible, and there exists another execution sequence $S_0 \overset{\mathcal{P}_B}{\rightsquigarrow} S_B \overset{\mathcal{P}'_A}{\rightsquigarrow} S'$, where $\mathcal{P}'_B = (\mathcal{P}_A \diamondsuit \mathcal{P}_B).remove(\mathcal{P}_A)$ and $\mathcal{P}'_A = (\mathcal{P}_B \diamondsuit \mathcal{P}_A).remove(\mathcal{P}_B)$.

Proof. According to Lemma 4, if the program is executed along $\mathcal{P}_A$ first, then our algorithm must explore the read r_B in the new trace. If $\mathcal{P}_A$ is reached first, r_B will be considered next. Assume an execution $S_0 \overset{\mathcal{P}_A}{\rightsquigarrow} S_A \overset{\mathcal{P}'_B}{\rightsquigarrow} S_B$. After the state S_A is reached by $S_0 \overset{\mathcal{P}_A}{\rightsquigarrow} S_A$, if $\mathcal{P}_A \cap \mathcal{P}_B = \emptyset$, it implies that for any event e, if $e \in \mathcal{P}_B$, then $e \notin \mathcal{P}_A$. According to Lemma 5, for each $e \in \mathcal{P}_B$, $e \rightarrow r_B$, therefore $e \in \mathcal{P}'_B$ (*Property 1*). So $\mathcal{P}'_B = \mathcal{P}_B$, which also equals to $\mathcal{P}_B.remove(\mathcal{P}_A)$, *i.e.*, the shortest sequence to make r_B see a different value. If $\mathcal{P}_A \cap \mathcal{P}_B \neq \emptyset$, then for each $e \in (\mathcal{P}_B - \mathcal{P}_A \cap \mathcal{P}_B)$, $e \rightarrow r_B$, and thus $e \in \mathcal{P}'_B$. Because events that have already been contained in $\mathcal{P}_A$ do not appear in the trace any more, $\mathcal{P}'_B$ will not contain any event $e | e \in \mathcal{P}_A$. So $\mathcal{P}'_B = \mathcal{P}_B - \mathcal{P}_A \cap \mathcal{P}_B = \mathcal{P}_B.remove(\mathcal{P}_A)$. After the program reaches state S_A along $\mathcal{P}_A$, $\mathcal{P}_B.remove(\mathcal{P}_A)$ is the shortest execution sequence that reaches r_B from S_A. Because of symmetry, if the program reaches S_B first along $\mathcal{P}_B$, it next explores r_A along $\mathcal{P}_A.remove(\mathcal{P}_B)$. □

5.3.3.2 *Completeness*

To show that our new searching algorithm is complete, we prove that SE-MCR never misses any states that can be generated by MCR.

Theorem 7. SE-MCR explores every unique execution sequence that MCR explores.

Suppose two execution sequences $\mathcal{P}_A.\mathcal{P}'_B \simeq \mathcal{P}_B.\mathcal{P}'_A$. According to Algorithm 8, we will skip the exploration of the sub-space constructed from the trace by re-executing the program starting with $\mathcal{P}_A.\mathcal{P}'_B$. Next, we prove that this does not cause our exploration to miss any behaviors that can be covered by MCR.

Proof. We proceed by contradiction. Suppose that the exploration of the trace by re-executing the program beginning with $\mathcal{P}_B.\mathcal{P}'_A$ misses a state but covered by the exploration along $\mathcal{P}_A.\mathcal{P}'_B$.

It can only be one of the two cases: (1) there exists a read r, $r \in \mathcal{P}_B.\mathcal{P}'_A$, but $r \notin \mathcal{P}_A.\mathcal{P}'_B$ such that r only appears in the trace guided by $\mathcal{P}_A.\mathcal{P}'_B$. If our algorithm selects to analyze the trace by re-executing the program starting with $\mathcal{P}_B.\mathcal{P}'_A$, it misses r. (2) there exists a write w, $w \in \mathcal{P}_B.\mathcal{P}'_A$, but $w \notin \mathcal{P}_A.\mathcal{P}'_B$. Then a read r which occurs later in the trace by the execution starting with $\mathcal{P}_B.\mathcal{P}'_A$ has no chance to read from w because this w has already been contained in the prefix. But the two cases above are impossible. Because both $\mathcal{P}_B.\mathcal{P}'_A$ and $\mathcal{P}_A.\mathcal{P}'_B$ derived from the same state, and the execution to reach that state is the same. Based on Theorem 6, for any event $e \in \mathcal{P}_A.\mathcal{P}'_B$, $e \in \mathcal{P}_B.\mathcal{P}'_A$. □

5.4 Parallel SE-MCR

DPOR uses a DFS strategy to search the state space and maintains a backtrack set for each scheduling point to record the next executable steps. The backtrack set is dynamically updated when the execution encounters events that are conflict with the previous steps. Unlike ICB and DPOR which are completely online and are hard to parallelize, *SE-MCR* opens the door for massive parallelism. As shown in Figure 5.7, *SE-MCR* adopts a BFS strategy to search the state space. For each state it reaches, it directly computes the possible seed interleavings from the current

Algorithm 10: Parallel-*SE-MCR* algorithm

```
1  𝒫 = {𝒫_0} // 𝒫: a set of unexplored prefixes
2  ℰ[𝒫_0] = ∅ // ℰ: a map from a prefix to its equivalent prefixes
   // each exploration can be in parallel
3  parallel foreach 𝒫 ∈ 𝒫 do
4      ParallelExplore(𝒫)
5  end
6  Function ParallelExplore(𝒫):
7      τ ← Execute(𝒫)
8      Φ ← ConstraintsModel(τ)
       // match each read to a different write in parallel
9      parallel foreach r = read(t, x, v) ∈ τ do
10         parallel foreach w = write(−, x, v′) ∈ τ ∧ v′ ≠ v do
11             Φ_𝒫 ← NewPrefixesConstraints(r, w, Φ)
12             𝒫 ← SolveConstraints(Φ_𝒫)
13             add 𝒫 to ℙ
14         end
15     end
       // call Algorithm 9 to check equivalence
16     CheckEquivalence(ℙ, ℰ)
17     foreach 𝒫_new ∈ ℙ do
18         if 𝒫_new ∉ ℰ[𝒫] then
19             add 𝒫_new to 𝒫
20         end
21     end
```

state (corresponding to an executed trace). 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, which can be further parallelized.

Algorithm 10 shows our parallel *SE-MCR* algorithm. To maximize the degree of parallelism, the input to the procedure *Explore* is a single seed interleaving (initially empty). At any time of our algorithm's execution, there can be many parallel executing *Parallel-MaxCausalExplore* procedures each working on a different seed interleaving. **parallel** means executing the **for** loop in parallel. 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. Lines 9-15 describe the parallel seed interleaving generation. 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 $\Phi_{\mathcal{P}}$ is constructed and solved with a SMT solver. If the constraint is satisfiable, a new seed interleaving will be returned and added to a set $\mathbb{P}$.

5.5 Implementation

We implement our tool using ASM [67] and Z3 [68]. The tool consists of three parts: an instrumentor, an offline constraint analyzer, and a dynamic scheduler. Although the tool so far is implemented in Java, the algorithm can be applied to any other programming languages. The tool is publicly available at: `https://github.com/parasol-aser/JMCR`.

Instrumentor To collect the trace information, we leverage the framework ASM to dynamically instrument the critical events using Java bytecode rewriting. The critical events include all shared data accesses and thread synchronizations. If an event is a *read* or *write*, we use a tuple $\langle tid, op, addr, value\rangle$ to record the thread ID, operation type, memory address of the shared variable and the value information of the event. If it is a synchronization, we use $\langle tid, op, object\rangle$ to describe the what *object* is protected by this synchronization. For *fork* and *join*, we record which thread creates/waits the child thread by $\langle op, tid, tid\rangle$. We also have considered *wait-notify* and *re-entrant locking* statements in our implementation:

- **wait-notify** - Java's *wait()* and *notify()/notifyAll()* are usually not discussed in previous studies [76, 77]. In our implementation, we treat *wait()* as two consecutive *release-acquire* events, *notifyAll()* as multiple *notify()* where the number is equal to the number of currently waiting threads on the same signal, and keep a mapping from *wait()* to its corresponding *notify()* in the original execution. In the constraint, we ensure the order of the *notify()* is between that of the two consecutive release-acquire events of the corresponding *wait()*, but not between that of any other wait() on the same signal (to ensure that the *notify()* is matched

with the same *wait()* as that in the original execution). Currently, we do not model spurious wakeups and lost notifications in our implementation. However, since they happen rarely in practice, this does not limit the usability of our tool.

- **re-entrant locking** - To simplify the constraint, re-entrant lock acquire/release events are filtered out dynamically in the execution, *i.e.*, discarding all but the outermost pair of *acquire/release* events on the same lock.

Offline constraint analyzer 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 constraints encoding is presented in Section 3.1.2. If the solver finds a solution to the constraints, it returns the solution which contains the integer variables and the values assigned to them by Z3. We first map the variables to their corresponding events in the trace and then order the events based on the value of each integer variable. Then we build the schedule by mapping the events to a sequence of thread IDs in the same order, in the form of $\langle tid, tid, tid \ldots \rangle$. As we only want to make the target read to return a different value, the schedule ends with that read event.

Dynamic scheduler To re-execute the class, we implement a re-execute engine based on the *BlockJUnit4ClassRunner* framework. The re-execution of the program is under the control of the dynamic scheduler. We block a thread before it accesses a shared memory. It queries the schedule choices from the prefix computed from the constraints and picks a thread to execute. After all the choices in the schedule are consumed, the scheduler can take a random order to execute the program.

To compare with the state-of-the-art, we have also implemented the ICB algorithm in the original CHESS model checker [20, 57] and its integration with DPOR [26]. In addition, for these algorithms we have implemented the detection of two safety violation properties: null pointer

dereference (NPE) and data race.

ICB and DPOR Our implementation of ICB follows the original algorithm [20]. 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 [26] is unsound. We follow the bounded partial order reduction [78] 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, our implementation maintains vector clocks for tracking happens-before following the optimization in [26]. When a dependence is detected, following [78], we create new schedules to explore by adding backtracking points at both the earlier event and a previous event in the schedule where the backtracked event does not require a preemption.

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 3.1.3. 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.

5.6 Evaluation

In this section, we present the evaluation results on SE-MCR and existing techniques. The evaluation aims to answer two research questions: (1) how effective is SE-MCR for exploring the state space of concurrent programs? (2) how effective is SE-MCR for finding concurrency errors? To answer the first question, we have made three comparisons: ❶ We compare effectiveness of MCR with DPOR and ICB+DPOR on state-space exploration. ❷ We then compare the the performance of SE-MCR and MCR. For this comparison, we add four mutual exclusion algorithms, which contain intensive shared write and read operations. We report the reduction made by SE-MCR in the number of interleavings executed and and the total exploration time used to search the state space of the benchmarks. As SE-MCR relies on extra data structure to store the equivalent execution prefixes, we evaluate the memory space overhead caused by SE-MCR. ❸ Last, we compare SE-MCR with the tool, *Nidhugg* [36], which implements the optimal DPOR. We compare the number of executions by each approach to show the effectiveness in reducing the size of the state space.

For the second question, we compared the number of executions required by different approaches to expose the injected or known errors in each benchmark. We also report the results of SE-MCR on two real-world applications, `Weblech` and `Jigsaw` and some new errors detected by our technique.

We evaluate the three approaches on a variety of popular multithreaded benchmarks including two real-world large applications. All experiments were conducted on a MacBook with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory and JDK 1.8. All results were averaged over three runs. We evaluated different approaches on a wide variety of benchmarks from prior work [19, 2, 79].

5.6.1 Comparison between SE-MCR and MCR

Table 5.1 summarizes the main results that compare our approach with MCR. The second column shows the number of lines of code for each benchmark, and the third shows the number of

threads that we use to evaluate each benchmark. Column 4 shows the comparison of the number of the executions used to explore the state space of each benchmark. Column 5 shows the comparison of the time used to explore the state space of each benchmark. We highlight the results of SE-MCR in gray. For the evaluation of the four mutual exclusion algorithms, we set the loop iteration bound of `Dekker` as 4 and the others as 10. The reason is that `Dekker` is implemented using two nested `while` loops, of which the size of the state space grows exponentially with the number of iterations. In our evaluation, when the loop iteration of *Dekker* is greater than 5, the model checker cannot finish the exploration in one hour. For `BubbleSort`, we tested the program on sorting an array with five integers. For `Account`, we allow four threads to make operations on the account. From the experimental results, we can find that the size of the state space does not depend on the number of lines of code, but the number of read/write operations to the shared memory location. For example, although the *StringBuf* contains about $1.3K$ *LoC*, it only took 8 executions for MCR to check this program. However, *Dekker* is a small program but generates more than four thousands of executions.

Reduction of #executions and exploration time. From the results in Table 5.1, we can see that except `Peterson`, SE-MCR improves the performance of MCR not only in the number of the executions and but also the total exploration time. The number of the executions reduced by SE-MCR ranges from 6.8% to 91.4%. Although it takes time for SE-MCR to check the redundancy across during runtime, the total exploration time is still reduced by 11.1% to 80.6%. The reason why SE-MCR can achieve a significant reduction on `Lamport`, `Dekker` and `BubbleSort` is that each thread in the three programs contains many reads. This implies that each layer of the exploration tree of these programs has more nodes, which increases the probability of redundant executions. If SE-MCR detects that some nodes at this layer can lead to redundant executions, all the explorations from that node starting with the identified execution sequence can be removed. Therefore, SE-MCR can significantly reduce the size of the state space. In fact, the earlier SE-MCR detects the potential redundancy, the more executions it can reduce.

Memory space overhead. The last column of Table 5.1 reports the extra memory space used by SE-MCR. On average, SE-MCR only consumes extra memory from 1.9KB to 937KB. To compute the average memory consumption, we first compute the total memory usage during the whole exploration and count the total number of executions. Then we get the average value by dividing the total memory usage with the number of the executions.

Table 5.1: Results on state-space exploration for benchmarks.

Program	LoC	#Thrd.	#Exec. Explored			Time			Space
			MCR	SE-MCR	reduction	MCR	SE-MCR	reduction	OH.
RVExample	79	3	107	77	$\downarrow 28.0\%$	$2s$	$2s$	–	2.8KB
Account	373	5	27830	14124	$\downarrow 49.2\%$	$45m\ 23s$	$17m\ 42s$	$\downarrow 61.0\%$	353KB
Airline	136	4	2222	1354	$\downarrow 39.1\%$	$1m\ 17s$	$43s$	$\downarrow 44.2\%$	60.2KB
Allocation	348	3	169	113	$\downarrow 33.1\%$	11.1s	10.1s	$\downarrow 9\%$	6.6K
BubbleSort	175	4	29148	8084	$\downarrow 72.3\%$	$20m\ 13s$	$5m\ 56s$	$\downarrow 70.7\%$	181KB
PingPong	388	6	411	282	$\downarrow 31.4\%$	13s	10.3s	$\downarrow 20.8\%$	26.9K
Pool	10K	3	3	3	–	0.9s	0.9s	–	370B
StringBuf	1.3K	3	8	8	–	$0.7s$	$0.7s$	–	249B
Peterson	94	3	402	402	–	$17s$	$17s$	–	1.9KB
Bakery	119	3	822	766	$\downarrow 6.8\%$	$45s$	$40s$	$\downarrow 11.1\%$	2.9KB
Lamport	162	3	1192	103	$\downarrow 91.4\%$	$51s$	$7s$	$\downarrow 86.3\%$	3.8KB
Dekker	119	3	4474	1648	$\downarrow 63.2\%$	$2m\ 17s$	$55s$	$\downarrow 59.9\%$	49.5KB

5.6.2 Comparison between SE-MCR and Optimal DPOR

We compared SE-MCR with *Nidhugg*, which implemented optimal DPOR, with respect to the number of explored interleavings for covering the whole state-space. As the tool is implemented in `C/C++`, we transformed five of the benchmarks to `C` for comparison. Table 5.2 reports the results. Comparing to *Nidhugg*, SE-MCR explores much fewer executions to cover the state space of the benchmark, especially for the `Lamport` and `Dekker` benchmarks. SE-MCR reduces the number of executions by more than 95% on these two programs. For `RVExample`, the number of interleavings executed by `Nidhugg` is almost five times that by SE-MCR. The reason why *Nidhugg* explores more executions is that it does not consider the values of the read and write operations. For two writes, if they are from different threads and write the same value to the

Table 5.2: Comparison of the number of the executions explored by *SE-MCR* and Nidhugg.

Program	**RVExample**	**Peterson**	**Bakery**	**Lamport**	**Dekker**
SE-MCR	77	402	766	103	1648
Nidhugg	577	780	3137	132034	34931

Table 5.3: Results on bug findings.

Program	**#Race**			**#NPE**		
	ICB	**ICB+DPOR**	***SE-MCR***	**ICB**	**ICB+DPOR**	***SE-MCR***
RVExample	7	10	10	0	0	0
Account	3	3	3	0	0	0
Airline	0	0	0	0	0	0
Allocation	0	0	0	0	0	0
BubbleSort	4	6	7	0	0	0
PingPong	6	7	7	1	1	1
Pool	0	0	0	0	0	0
StringBuffer	0	0	0	0	0	0

same memory location, optimal DPOR will think that there is a dependency between the two writes and the reordering of them could make the program reach a new state. Because the tools are implemented in different programming language, we didn't make comparisons on the time performance of the two tools.

5.6.3 Bugs Finding Report

Table 5.3 shows the races and NPEs detected by SE-MCR and ICB and DPOR. Among all the benchmarks, all tools do not detect any data races and NPEs in `Airline`, `Allocation`, `Pool` and `StringBuffer`. For `RVExample` ICB only finds 7 races while both ICB+DPOR and SE-MCR find 10 races. For `BubbleSort`, SE-MCR detects two more races than ICB and one more race than ICB+DPOR. For `PingPong`, SE-MCR detects one more race than ICB and all the tools find the NPE.

Results on real-world applications To show the effectiveness and scalabitlity of SE-MCR for finding bugs, we also evaluate SE-MCR on two real-world Java applications. Table 5.4 reports our

Table 5.4: Results on real applications. * means OOM.

Program		ICB	ICB+DPOR	*SE-MCR*	*SE-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

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. The last column shows the results of parallelized SE-MCR. 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 *SE-MCR* 32 before they terminated or timed out. For `Weblech`, ICB explored 1229 executions, ICB+DPOR 1072, and *SE-MCR* 185.

Although *SE-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`, *SE-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`, *SE-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 the MCR algorithm on a 32-core machine, *SE-MCR-P* was able to explore many more executions and detect more data races and NPEs within the same time. For `Jigsaw`, *SE-MCR-P* was able to explore 769 executions and detected 38 data races and 10 NPEs, and for `Weblech`, *SE-MCR-P* explored 3311 executions and detected one more data race than *SE-MCR*.

5.7 Summary

We presented the maximal causality reduction (MCR) for reducing the redundant interleavings from the state space of concurrent programs. MCR leverages SMT constraints to model the space of a given trace and takes the value of read and write accesses into consideration to remove re-

dundant states. We have also extended MCR to present a new algorithm *SE-MCR*, which explores a provably minimal set of interleavings. The algorithm regards the exploration of the state space as a process of traversing a tree using breadth first search (BFS). On each *state* node, we make an equivalence checking between the sibling nodes on the same level of the tree. We mark the potential equivalent prefixes and temporarily store them so that the model checker can skip them in the future exploration. We have implemented MCR and *SE-MCR* in a stateless model checker for Java programs. We compared MCR with DPOR and ICB in terms of bug finding and state-space exploration. Our experimental results show that MCR reduces the number of executions by orders of magnitude, comparing to DPOR and ICB. *SE-MCR* significantly reduces the number of executed interleavings and the total exploration time as well. On average, *SE-MCR* reduces the number of explored interleavings by 58.6% and exploration time by 45.6%, compared to the original MCR. We also compared our algorithm with *Nidhugg*, which implements optimal-DPOR. The comparison shows that our algorithm is much more efficient in reducing the size of the state space.

6. H3: PRODUCTION-RUN HEISENBUGS REPRODUCTION WITH INTEL PT *

In addition to verifying concurrent programs, which is sometimes limited to the scalability issue, another effective approach to fixing concurrency bugs is to improve the efficiency of debugging the concurrent program. However, as aforementioned, the bug may disappear when re-executing the concurrent program due to the non-deterministic memory races. Therefore, the ability to reproduce software bugs is crucial for debugging.

Researchers have investigated significant efforts in *record & replay* (RnR) systems aiming to eliminate the non-determinism. CLAP [21] is the most efficient software-based appraoch. It introduces the idea of recording only thread-local information (*i.e.*, thread-local *control flow* paths) and then using offline constraint solving to reconstruct the shared memory dependencies. It is a promising solution for reproducing Heisenbugs because it does not record any cross-thread communication (data or synchronization); hence it requires no synchronizations during recording, which not only reduces the runtime overhead but also minimizes the observer effect.

However, CLAP is still unsatisfactory to enable a production-run RnR solution. In this thesis, I first introduce the basics about CLAP and its limitations, and then present H3, a new *record & replay* (RnR) system to reproduce Heisenbugs using commercial hardware features and offline constraints analysis.

6.1 CLAP

CLAP can not only reproduce Heisenbugs under sequential consistency (SC), but also a wide range of weak consistency memory models, including TSO (total store order) and PSO (partial store order) [62]. It has two key components: I) collecting per-thread control flow information via software path-recording (using an extended Ball-Larus path-recording algorithm [80]), and II) assembling a global schedule by solving symbolic constraints constructed over the thread local

*Reprinted with permission from "Towards Production-Run Heisenbugs Reproduction on Commercial Hardware" by Shiyou Huang, Bowen Cai and Jeff Huang, 2017. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17), 403-415.

paths. To assemble a global schedule, CLAP has three steps:

1. Along the local path of each thread, it collects all the critical accesses (read, write or synchronization) to shared variables.

2. It introduces a fresh symbolic value for each read access, and collects the path constraints following the control flow for each thread via symbolic execution; it introduces an order variable for each critical access, and generates additional constraints according to synchronization, memory-consistency model, and potential inter-thread memory dependencies.

3. It uses an SMT solver to solve the constraints, to which the solutions correspond to global schedules that can reproduce the error. In other words, the SMT solver computes what inter-thread memory dependencies would satisfy the memory-consistency model and enable the recorded local execution path.

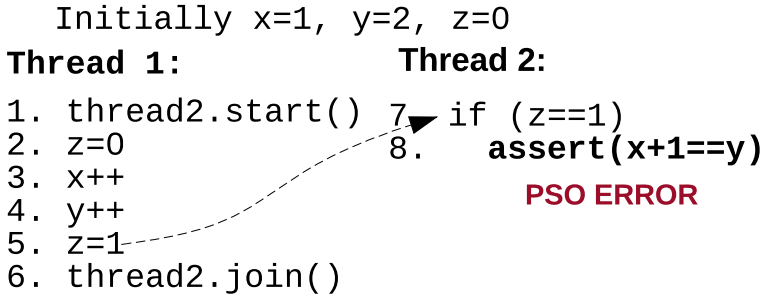


Figure 6.1: A real PSO bug in an electron microscope software [3], which caused a $12 million loss of equipment. Reprinted with permission from [4].

CLAP contains several components to model a failing execution as constraints (*e.g.*, failure, path, synchronization, read-write, and memory model). We next use an example in Figure 6.1 to illustrate these constraints. Section 6.3.3 presents the constraint model in detail.

The program in Figure 6.1 contains a real Heisenbug that only manifests under the PSO memory model, which caused a $12 million financial loss in the real-world [3]. The root cause of the

Read-Write Constraints

$$(R_z^7 = 0 \ \wedge O_7 < O_2) \vee$$
$$(R_z^7 = W_z^2 \ \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)) \vee$$
$$(R_z^7 = W_z^5 \ \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC	PSO
$O_1 < O_2 < O_3^{Rx} < O_3^{Wx} < O_4^{Rx} < O_4^{Wx} < O_5 < O_6$ $O_7 < O_8^{Rx} < O_8^{Ry}$	$O_1 < O_2 \quad O_5 < O_6$ $O_3^{Rx} < O_3^{Wx} \quad O_4^{Rx} < O_4^{Wx}$ $O_7 < O_8^{Rx} < O_8^{Ry}$

Path Constraints	Failure Constraints
$R_z^7 = 1$	$R_x^8 + 1! = R_y^8$

Figure 6.2: The CLAP constraints for reproducing the PSO error in Figure 6.1. To save space, we show the read-write constraints for z only. Those for x and y are similar. Reprinted with permission from [4].

bug is that the write to z (line 5) can be reordered with the writes to x and y (lines 3-4) under PSO. The dashed arrow in the figure shows that the satisfaction of the *if* condition at line 7 depends on the write to z at line 5, which always happens after lines 3 and 4 under SC. However, under PSO, the write to z is allowed to happen before the write to y at line 4. As a result, when the *if* condition is satisfied, the value of $x + 1$ and y may be unequal and hence triggering the error. The error can be triggered by the following PSO schedule: 1-2-3_{R_x}-3_{W_x}-4_{R_y}-5-7-8_{R_x}-8_{R_y} (the subscripts are used to distinguish different operations from the same line).

The CLAP constraints for reproducing the buggy PSO schedule are shown in Figure 6.2. We use the order variable O_i denotes the order of the corresponding access at line i. The symbolic variable R_v^i denotes the value returned by the read access to the variable v at line i, and W_v^i the value written to v by the write at line i. To distinguish different operations at the same line, we add the type of the operation to the order variable. For example, $O_3^{R_x}$ and $O_3^{W_x}$ represent the orders of the read and write to x at line 3, respectively.

To manifest the error, CLAP enforces the assertion to be violated while satisfying the path constraints, i.e., $true \equiv (R_z^7 = 1 \wedge R_x^8 + 1 \neq R_y^8)$. A major part of the CLAP constraints is the read-write constraints, which are used to capture the potential inter-thread memory dependencies.

Because the order of the memory accesses from different threads is unknown, the read-write constraints must encode a schedule for every potential read-write match, in which the read returns the value written by the write. For example, the read of z at line 7, R_z^7, may be matched with either the initial value 0, or the value written by line 2 or 5. If the former, the read R_z^7 should happen before all the writes to z; if the latter, R_z^7 should be matched with the corresponding write. For example, if R_z^7 returns the value by the write at line 2, the constraint $R_z^7 = W_z^2 \wedge O_2 < O_7 \wedge (O_5 < O_2 \vee O_7 < O_5)$ is generated.

CLAP Limitations **1. Exponential complexity of read-write constraints.** The read-write constraints generated by CLAP are very complicated in practice because there may exist many writes that a read can be matched with. In the worst case, the complexity of the read-write constraints (*i.e.*, the space of scheduling choices) is exponential in the number of writes (which typically accounts for a large percentage of the events in the trace). This is a bottleneck in CLAP especially for programs with intensive inter-thread memory dependencies, because the SMT solver may fail to solve the constraints. We will present a detailed complexity analysis in Section 6.3.4.

2. Slowdown of software path-recording. CLAP uses a highly optimized algorithm (i.e., Ball-Larus [80]) to track the control flow information for each thread. Although it greatly reduces the runtime overhead incurred by many other RnR solutions, it still incurs 10%-3X performance slowdown on popular benchmarks [21]. For instance, for the example in Figure 6.1, when the code is executed in a loop for 10 million times, CLAP incurs 2.3X program slowdown.

3. Difficulty of code instrumentation. It is difficult to apply software path-recording in production runs because it requires code instrumentation. Real-world programs often rely on external libraries, proprietary code, and/or are composed from layers of frameworks and extended by third-party plugins. Tracing the whole program control flow by code instrumentation is difficult or impossible. For example, if a failure is caused by a bug in the uninstrumented external code, the constraints generated by CLAP may be incomplete and hence fail to reproduce the bug.

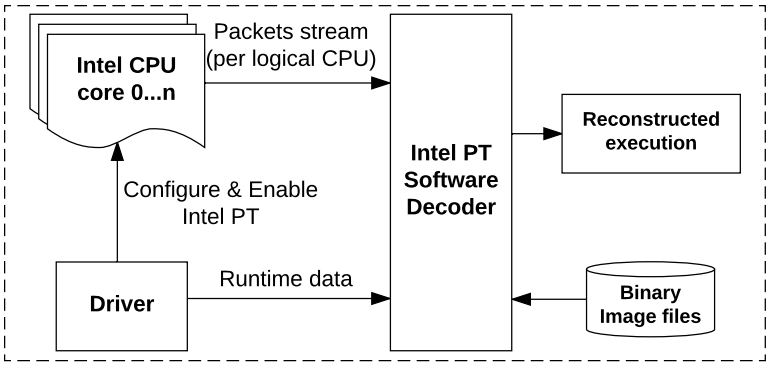


Figure 6.3: Components of Intel Processor Tracing (PT). Reprinted with permission from [4].

6.2 Hardware Control-Flow Tracing

Tracing control flow at the hardware level opens a door to apply CLAP in production runs by addressing the aforementioned limitations in three ways. First, hardware-supported control flow tracing is significantly more efficient than software-level path-recording. Compared to the 10%-3X overhead by software path-recording, PT achieves as low as 5% runtime overhead [81]. Second, hardware can track the full control flow of the code executed on each core. PT can not only trace the application code, but also the whole operating system kernel [81]. Third, tracing the control flow on each core enables a significant reduction of the complexity of the read-write constraints, because reads and writes from the same core are ordered already.

Next, we first review the basics of PT and then show its performance improvement over software path-recording on PARSEC 3.0 benchmarks [82].

6.2.1 PT

As depicted in Figure 6.3, PT consists of two main components: tracing and decoding. For tracing, it only records the instructions that are related to the change of the program control flow and omits everything that can be deduced from the code (*e.g.*, unconditional direct jumps). For each conditional branch executed, PT generates a single bit (1/0) to indicate whether a conditional branch is taken or not taken. As such, PT tracks the control flow information, such as loops, conditional branches and function calls of the program, with minimal perturbation, and outputs a

highly compact trace.

For decoding, PT provides a decoding library [83] to reconstruct the control flow from the recorded raw trace. It first synchronizes the packet streams with the synchronization packets generated during tracing, and then iterates over the instructions from the binary image to identify what instructions have been executed. Only when the decoder cannot decide the next instruction (e.g., when it encounters a branch), the raw trace is queried to guide the decoding process.

PT is configurable via a set of model-specific registers by the kernel driver. It provides a privilege-level filtering function for developers to decide what code to trace (*i.e.* kernel vs. user-space) and a CR3 filtering function to trace only a single application or process. PT on Intel Skylake processors also supports filtering by the instruction pointer (IP) addresses. This feature allows PT to selectively trace code that is only within a certain IP range, which can further reduce the tracing perturbation.

6.2.2 PT Performance.

Table 6.1: Runtime and space overhead of PT on PARSEC. Reprinted with permission from [4].

Program	Native	PT		
	time (s)	time (s)	**OH(%)**	trace
bodytrack	0.557	0.573	**2.9%**	94M
x264	1.086	1.145	**5.4%**	88M
vips	1.431	1.642	**14.7%**	98M
blackscholes	1.51	1.56	**9.9%**	289M
ferret	1.699	1.769	**4.1%**	145M
swaptions	2.81	2.98	**6.0%**	897M
raytrace	3.818	4.036	**5.7%**	102M
facesim	5.048	5.145	**1.9%**	110M
fluidanimate	14.8	15.1	**1.4%**	1240M
freqmine	15.9	17.1	**7.5%**	2468M
Avg.	4.866	5.105	**4.9%**	553M

Table 6.1 reports the runtime and space overhead of PT on the PARSEC 3.0 benchmarks. We report the execution time of the programs without and with PT tracing (and the trace size), marked

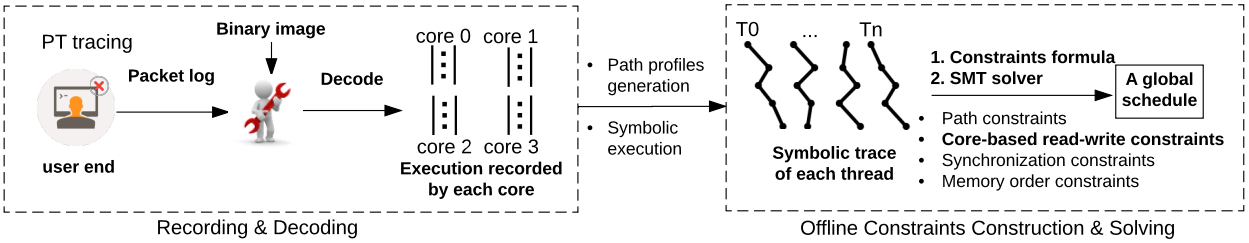


Figure 6.4: H3 Overview. Reprinted with permission from [4].

as native and PT respectively. Among the 10 benchmarks, PT incurs 1.4% to 14.7% runtime overhead (4.9% on average) and 88MB to 2.4GB space overhead (0.5GB on average).

6.3 H3

In this section, we present the technical details of H3. As we have described in Figure 6.4, H3 integrates hardware control-flow tracing with offline symbolic constraint analysis to reproduce Heisenbugs. H3 consists of two phases. First, users run the target program on a COTS (commercial off-the-shelf) hardware with PT enabled. Once a failure occurs, the PT trace together with the thread context switch log are sent to the developer for reproducing the bug. From the PT trace and the binary image of the target program, H3 generates the instructions executed on each core. Second, H3 infers the instructions executed by each thread based on the thread context switch log and generates a symbolic trace for each thread. It then constructs symbolic constraints with the core-based constraint reduction, and computes a global failure reproducing schedule with an SMT solver. Although the overall flow is easy to understand, there are three technical challenges in the integration:

1. **Absence of the thread information**. There is no thread information from the PT traces. It is unknown which instruction is executed by which thread, and hence difficult to construct the inter-thread synchronization and memory dependency constraints.

2. **Gap between low-level hardware traces and high-level symbolic traces**. The decoded execution from PT is in the low-level assembly form. However, to construct constraints and to reproduce bugs, we need a high-level symbolic trace containing shared variable accesses

and branch conditions.

3. **No data values for memory accesses**. PT only traces control flow information but does not record any data values of memory accesses. To reconstruct the shared memory dependencies, we need a way to match reads with writes without using values.

We present our solutions to these challenges in the next three subsections. We also present a constraint reduction algorithm in Section 6.3.4 enabled by the partial order of writes per-core, which significantly reduces the complexity of the generated constraints.

6.3.1 Thread Local Execution Generation

We leverage the context-switch software events (generated by the Linux Perf tool) to distinguish instructions from different threads. Each context-switch event contains three attributes: TID, CPUID, and TIME (*i.e.*, the timestamp of the event). Because PT also generates frequent synchronization packets (including the timestamp information) into the packet stream, we can use the timestamp information to synchronize the context switch events with the PT packets from the same core (*i.e.*, CPUID). Because the timestamp clocks local to each core is precise, the inferred thread identity based on the timestamp information is also precise. Hence, we locate the context switch points in the PT packets on each core by comparing the timestamps, and determine the thread identity of each instruction as the TID attribute of the leading context-switch event.

6.3.2 Symbolic Trace Generation

In CLAP, the symbolic trace of each thread is generated by symbolic execution along the recorded path profile of each thread. The path profile for each thread is decoded (from the Ball-Larus path encoding [80]) as a sequence of basic block transitions at the LLVM IR level in the form of (*Tid, BasicBlockId*). In H3, we also rely on these high-level per-thread path profiles to collect the symbolic traces, and we extract the path profiles from the low-level PT trace as follows. We first instrument all basic blocks of the target program and assign each a unique identifier. Then we compare the generated assembly code from the instrumented program with the decoded instructions from the PT trace to identify which basic blocks are executed by each thread.

Algorithm 11: Path profiles generation

```
   Input  : 𝕃:< line, insn >
   // execute instructions and #line
   Input  : 𝔹: <line, block_id>
   // basic blocks of the paths
   Return: ℚ: <tid, block_id>
   // path profile of each thread
   // traverse each thread
1  for each tid do
       // get the instructions of each thread
2      ℓ = {𝕊 ⊆ 𝕃|∀insn ∈ 𝕊.insn, Tid(insn) = tid} ;
3      for each item ∈ ℓ do
4          if item.line ∈ 𝔹.line then
5              block_id = 𝔹.get(item.line) ;
6              ℚ.add(tid, block_id) ;
7  return ℚ
```

Algorithm 7 shows the process of generating the path profiles for each thread. The algorithm takes as input: (1) the executed instructions and their corresponding line number; and (2) the basic blocks of the control-flow of the program with the *BlockId* and the line number of the first instruction of this block. The algorithm first gets the executed instructions by each thread (line 3) and then matches the line number of the executed instructions with that contained in each basic block (line 4-7). To identify the path profile of a thread, the algorithm iterates over the instructions of each thread to check whether the instruction is the first one of the block by comparing the line number (line 5). If so, we add this block into the path profile as (*Tid, BasicBlockId*).

6.3.3 Matching Reads and Writes

To reconstruct the shared memory dependencies without data values, similar to CLAP, we construct a system of symbolic constraints over the per-thread symbolic traces. The basic idea is to introduce an order variable for each read/write denoting the unknown scheduling order, and a symbolic variable for each read/address denoting the unknown read value and address. We symbolically execute the program following the recorded per-thread control flow, and constructs

constraints over the order and symbolic variables to determine the inter-thread orders and values of reads/addresses.

More specifically, we construct a system of SMT constraints formula, denoted by Φ_g, over the symbolic traces. The computed orders/values from solving Φ_g then correspond to one or more concrete global schedules that can reproduce the Heisenbugs. We note that the computed schedules may be different from that in the failure execution, but any one of them is sufficient to reproduce the Heisenbugs.

Φ_g can be decomposed into five parts:

$$\Phi_g = \Phi_{path} \wedge \Phi_{bug} \wedge \Phi_{sync} \wedge \Phi_{mo} \wedge \Phi_{rw}$$

where Φ_{path} denotes the path conditions by each thread; Φ_{bug} the condition for the bug manifestation; Φ_{sync} the interactions between inter-thread synchronizations; Φ_{rw} the potential inter-thread memory dependencies; and Φ_{mo} the memory model constraints. The formula contains two types of variables: (1) *V - the symbolic value variables* denoting the values returned by reads; and (2) *O - the order variables* the order of each operation in the final global schedule.

Path Constraints (Φ_{path}). The path constraints are constructed by a conjunction of all the path conditions of each thread, with each path condition corresponds to a branch decision by that path. The path conditions are collected by recording the decision of each branch via symbolic execution.

Bug Constraints (Φ_{bug}). The bug constraints enforce the conditions for a bug to happen. A bug can be a crash segfault, an assert violation, a buffer overflow, or any program state-based property. To construct the bug constraints, an expression over the symbol values for satisfying the bug conditions is generated. For example, the violation of an assertion exp can be modeled as $!exp$.

Synchronization Constraints (Φ_{sync}). The synchronization constraints consist of two parts: partial order constraints and locking constraints. The partial order constraints model the order between different threads caused by synchronizations *fork/join/signal/wait*. For example, The *begin* event of a thread t should happen after the *fork* event that starts t. A *join* event for a thread t should

happen after the last event of t. The locking constraints 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, all the *lock/unlock* pairs of events are extracted, and the following constraints for each two pairs (l_1, u_1) and (l_2, u_2) are constructed: $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

Memory Order Constraints (Φ_{mo}). The memory order constraints enforce orders specified by the underlying memory models. H3 currently supports three memory models: SC, TSO and PSO. For SC, all the events by a single thread should happen in the program order. TSO allows a read to complete before an earlier write to a different memory location, but maintains a total order over writes and operations accessing the same memory location. PSO is similar to TSO, except that it allows re-ordering writes on different memory locations.

Read-Write Constraints (Φ_{rw}). Φ_{rw} matches reads and writes by encoding constraints to enforce the read to return the value written by the write. Consider a read r on a variable v and r is matched to a write w on the same variable; we must construct the following constraints: the order variables of all the other writes that r can be matched to are either less than O_w or greater than O_r.

As discussed in Section 6.1, Φ_{rw} can be complicated because there may exist many potential matches between reads and writes. The size of Φ_{rw} is cubic in the trace size and its complexity is exponential in the trace size. Nevertheless, in next subsection, we show that both the size and complexity of Φ_{rw} can be greatly reduced in H3.

6.3.4 Core-based Constraints Reduction

Besides the low runtime overhead, another key innovation enabled by PT is that the order of executed events on each core (either by the same thread or by different threads) is determined, which can reduce the complexity of Φ_{rw} from exponential in the number of writes to exponential in the core counts.

The key observation of this reduction is that the executed memory accesses on each core decoded from PT trace are already ordered, following the program order. Once the order of a certain write in the global schedule is determined, all the writes that happen before or after this write,

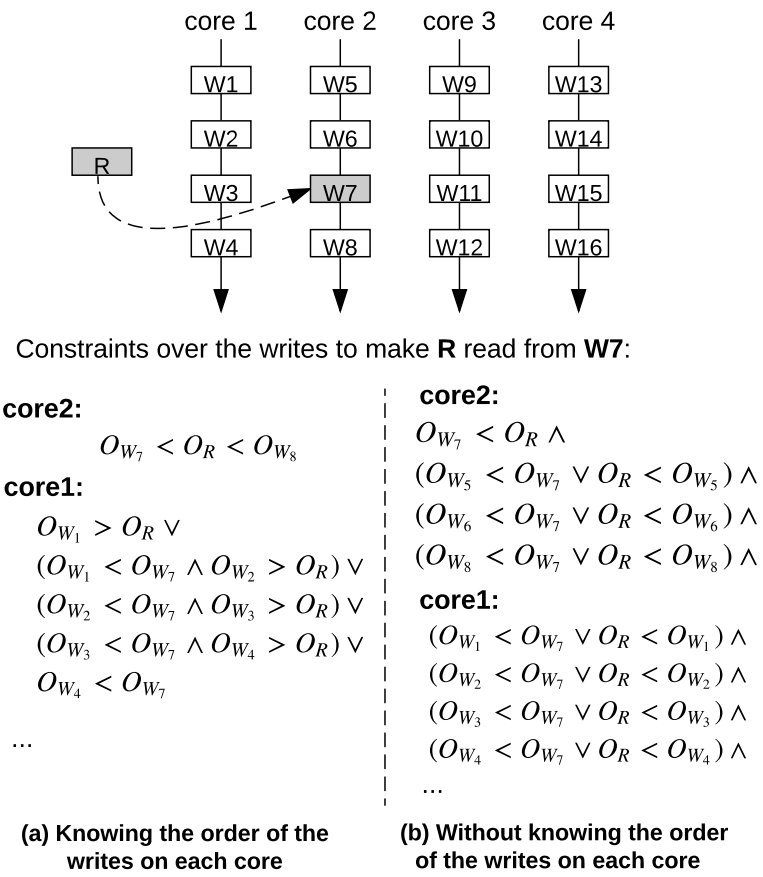


Figure 6.5: Core-based constraint reduction. Reprinted with permission from [4].

on the same core, should occur before or after this write in the schedule correspondingly. This eliminates a large number of otherwise necessary read-write constraints for capturing the potential inter-thread memory dependencies.

Consider an example in Figure 6.5, which has four cores with each executing four different writes. Suppose there is a read R that can be potentially matched with all of these writes, because each of them writes a different value to the same shared variable read by R. Without the partial order information of each core, we must include all writes and their orderings into the constraints. For instance, if R reads the value from the write W_7 on Core 2, then R must happen after W_7 (*i.e.*, $O_R > O_{W_7}$), and all the other writes must either happen before W_7 or after R. Taking W_5 as an example; it must either happen before W_7 or after the read R, resulting in the constraint $(O_R < O_{W_5} \vee O_{W_5} < O_{W_7})$. In general, if there are N writes in the trace, the constraints can

generate 2^N different ordering choices for these writes. As typically most accesses in the trace are reads and writes, this exponential search space can be a bottleneck for the technique to scale.

However, with the per-core partial order information, the execution order of the writes on each core is already determined. To prevent other writes from happening between the considered write and read, we only need to take the read-write as a whole and insert it to those sorted writes. Algorithm 7 presents our constraints reduction algorithm. Following this algorithm, to make R

Algorithm 12: Core-based constraints reduction

Input : a matched read-write $< R, W >$
Return: Φ_{rw} to make *R* read from *W*

1 *Initial*: $\Phi_{rw} = \emptyset$;
2 ***case 1***: *writes executed on the same core as* W ;
3 $\Phi_{rw} = \Phi_{rw} \wedge (O_W < O_R < O_{W'})$ // W' happens right after W on the same core
4 ***case 2***: *writes executed on other cores* ;
5 // for any two writes W_i and W_{i+1} on the same core
6 $\Phi_{rw} = \Phi_{rw} \wedge (O_R < O_{W_i} \vee \ (O_{W_i} < O_W \wedge O_R < O_{W_{i+1}})) \vee \ O_W > O_{W_{i+1}}$;
7 **return** Φ_{rw}

read from W_7, for all the other writes on Core 2, we only require $O_{W_7} < O_R < O_{W_8}$. Moreover, for the writes on the other cores, our new constraints encode fewer ordering choices. For example, for the four writes (W_1-W_4) on Core 1, the constraints are written as $O_R < O_{W_1} \vee (O_{W_1} < O_{W_7} \wedge O_R < O_{W_2}) \vee (O_{W_2} < O_{W_7} \wedge O_R < O_{W_3}) \vee (O_{W_3} < O_{W_7} \wedge O_R < O_{W_4}) \vee O_{W_4} < O_{W_7})$. There are only 5 ordering choices (compared to 16 in CLAP).

We note that the core-based constraints apply to SC and TSO, but may not apply to those weak memory models that allow re-ordering of writes on the same core. The reason is that if writes are re-ordered, the partial order witnessed on each core may not reflect the actual buggy execution order.

Theorem 8 below states the soundness guarantee of the core-based reduction:

Theorem 8. If a concurrent program runs on an SC or TSO platform with C cores and there are N writes executed, the number of the ordering choices of the read-write constraints is reduced from 2^N to $(\frac{N}{C}+1)^C$.

Proof. Consider that a read R returns the value of a write W. When not knowing the partial order of the writes on each core, each write either happens before W or after R. Consequently, there are 2^N ordering choices in total. If the partial order of the writes on each core is known and each core contains $m_i = \frac{N}{C}$ writes, the ordering on each core has only $m_i + 1$ choices. Therefore, the total number of choices is reduced to $\Pi_{i=1}^{C}(m_i+1)$, which equals to $(\frac{N}{C}+1)^C$.

6.4 Implementation

We have implemented H3 for Pthreads-based C/C++ programs based on a number of tools, including CLAP [21], the Linux Perf Tools [84], the PT decoding library [83], and the Z3 SMT solver [22]. We use Perf to control Intel PT to collect the packet streams and the context switch events. We first insert the context switch events to the packet streams by comparing the timestamp information, and then use the PT decoding library to decode the packets information. As in CLAP, we use KLEE [85] as the symbolic execution engine to generate the symbolic traces for each thread, and construct an SMT constraint formula. We modified CLAP to implement the core-based constraint reduction algorithm, and we use Z3 to solve the constraints.

Shared Variable Identification. We first run a static thread sharing analysis based on the Locksmith [86] race detector and then manually mark each shared variable `x` as symbolic by `klee_make_symbolic(&x, sizeof(x), "x")`, like CLAP. One way to automate this step is to conservatively consider all variables in the program as potentially shared and marked them as symbolic. However, this would produce a large amount of unnecessary constraints. For external function calls that are not supported by KLEE, we also mark the input and return variables of the external function calls as symbolic.

Constraint Reduction. For the core-based constraint reduction, we first extract the writes on the same core from the PT trace and store these writes in a map (*core Id:* $w_1[line], w_2[line]...$).

When constructing the read-write constraints, this map is used to determine which write belongs to which core by comparing the associated line number information. Because all writes on the same core occur in the order that they are executed, we construct a happens-before constraint over these writes. When matching a read r to a corresponding write w, we first constrain r to happen after w and happen before the write that occurs right after w on the same core, and we then only need to disjunct the order constraints between w and those writes from a different core.

6.5 Evaluation

Our evaluation of H3 focuses on answering two sets of questions:

- How is the runtime performance of H3? How much runtime improvement is achieved by H3 compared to CLAP?
- How effective is H3 for reproducing real-world Heisenbugs? How effective is the core-based constraint reduction technique?

6.5.1 Methodology

Table 6.2: Benchmarks. Reprinted with permission from [4].

Program	LOC	#Threads	#SV	#insns (executed)	#branches (total)	#branches (app)	Ratio app/total	Symb. time
racey	192	4	3	1,229,632	78,117	77,994	99.8%	107s
pfscan	1026	3	13	1,287	237	43	18.1%	2.5s
aget-0.4.1	942	4	30	3,748	313	5	1.6%	117s
pbzip2-0.9.4	1942	5	18	1,844,445	272,453	5	0.0018%	8.7s
bbuf	371	5	11	1,235	257	3	1.2%	5.5s
sbuf	151	2	5	64,993	11,170	290	2.6%	1.6s
httpd-2.2.9	643K	10	22	366,665	63,653	12,916	20.3%	712s
httpd-2.0.48	643K	10	22	366,379	63,809	13,074	20.5%	698s
httpd-2.0.46	643K	10	22	366,271	63,794	12,874	20.2%	643s

We evaluated H3 with a variety of multithreaded C/C++ programs collected from previous studies [21, 87, 88], including nine popular real-world applications containing known Heisenbugs.

Table 6.2 summarizes these benchmarks. *pfscan* is a parallel file scanner containing a known bug; *aget-0.4.1* is a parallel *ftp/http* downloading tool containing a deadlock; *pbzip2-0.9.4* is a multi-threaded implementation of *bzip* with a known order violation; *bbuf* is shared bounded buffer and *sbuf* is a C++ implementation of the JDK1.4 *StringBuffer* class; *httpd-2.2.9*, *httpd-2.0.48*, *httpd-2.0.46* are from the Apache HTTP Server each containing a known concurrency bug; We also included *racey* [88], a special benchmark with intensive races that are designed for evaluating RnR systems. We use *Apache Bench* (*ab*) to test *httpd*, which is set to handle 100 requests with a maximum of 10 requests running concurrently.

We compared the runtime performance of H3 and CLAP by measuring the time and space overhead caused by PT tracing and software path-recording. We ran each benchmark five times and calculated the average. All experiments were performed on a 4 core 3.5GHz Intel i7 6700HQ Skylake CPU with 16 GB RAM running Ubuntu 14.04.

We evaluated the effectiveness of H3 for reproducing bugs by checking if H3 can generate a failure reproducing schedule and by measuring the time taken by offline constraint solving. We set one hour timeout for Z3 to solve the constraints.

For most benchmarks, the failures are difficult to manifest because the erroneous schedule for triggering the Heisenbugs is rare. Similar to CLAP, we inserted timing delays (*sleep* functions) at key places in each benchmark and executed it repeatedly until the failure is produced. We also added the corresponding assertion to denote the bug manifestation.

Benchmark Characteristics. Table 6.2 reports the execution characteristics of the benchmarks. Columns 3 and 4 report the number of threads and shared variables, respectively, contained in the execution. We also profiled the total number of the executed instructions and branches in the assembly code, and the branches from the LLVM IR code, as reported in Columns 5-7. Column 8 reports the ratio of the number of the branches in the instrumented application code versus the total number of branches (in both the application code and all the external libraries). For most benchmarks (except *racey*), the ratio is smaller than or around 20%. Column 9 reports the time for constructing the symbolic trace for the corresponding recorded execution of the benchmark.

Table 6.3: Performance comparison between H3 and CLAP. Reprinted with permission from [4].

Program	Native time (s)	Time (s)			Branch	Space overhead	
		CLAP (**Overhead**)	H3 (**Overhead**)	Speedup	insts%	CLAP	H3
racey	0.268	0.768(186.6%)	0.288(**7.5%**)	**65.2%**	6.4%	96M	2.68M
pfscan	0.094	0.104(11.0%)	0.116(**23.4%**)	-11.5%	18.4%	3.2K	30K
aget-0.4.1	0.139	0.156 (12.1%)	0.152(**9.4%**)	**2.6%**	17.9%	11K	41K
pbzip2-0.9.4	0.102	0.134(31.4%)	0.112(**9.8%**)	**16.4%**	14.8%	5.2K	677K
bbuf	0.232	0.696(200%)	0.264(**13.8%**)	**62.1%**	20.1%	3.9K	2.7M
sbuf	0.216	0.299(38.5%)	0.256(**18.5%**)	**14.4%**	17.2%	6.6K	4.5M
httpd-2.2.9	0.53	0.71(34.0%)	0.57(**7.5%**)	**19.7%**	17.4%	7.8M	10.43M
httpd-2.0.48	0.45	0.59(32.1%)	0.51(**13.3%**)	**13.6%**	17.4%	8.1M	11.79M
httpd-2.0.46	0.42	0.57(36.2%)	0.50(**19.0%**)	**12.3%**	17.4%	7.2M	10.62M
avg.	0.272	0.447(64.3%)	0.307(**12.9%**)	**31.3%**	16.3%	13.2M	4.8M

6.5.2 Runtime Performance

Table 6.3 reports the performance comparison between H3 and CLAP. Column 2 reports the native execution time of the benchmarks. Columns 3-4 report the execution time with H3 and CLAP and their runtime overhead. Column 5 reports the speedup of H3 over CLAP. Column 6 reports the percentage of branch instructions in the execution. This number is proportional to the runtime overhead of PT. Columns 7-8 report the space overhead of H3 and CLAP, respectively.

Overall, the runtime overhead of H3 on these benchmarks ranges between 7.5%-23.4% and 12.9% on average. Compared to CLAP (11.0%-2X overhead), H3 achieves as much as 8X performance improvement and reduces its overhead significantly by 2.6%-65.2% and 31.3% on average. The only exception is *pfscan*. However, this is just because *pfscan* contains significantly more external calls compared to the other benchmarks; while H3 records all external library calls, the implementation of CLAP does not (which sacrifices the correctness). In addition, the short execution time of *pfscan* can suffer from noise.

For space overhead, H3 produces 30KB-2.4GB traces on these benchmarks, whereas CLAP produces 2KB-2.1GB. Some numbers of CLAP are smaller than that of H3, because external library calls are not traced by CLAP.

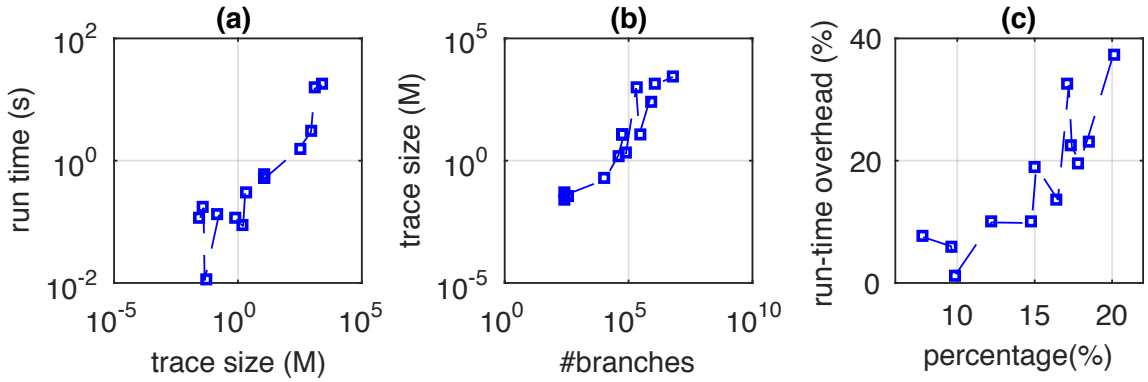


Figure 6.6: H3 performance analysis. Reprinted with permission from [4].

H3 performance analysis. We note that the performance of H3 is dominated by PT for tracking the control flow events. The additional cost for H3 to track context switching events is almost negligible as compared to tracing the control flow. We have also evaluated the runtime performance of H3 on the PARSEC 3.0 benchmarks and found that H3 incurs only 1.4% to 14.7% runtime overhead (4.9% on average) and 0.5GB trace size, the same as that reported in Table 6.1 for PT.

We further conducted a performance study of H3 on PARSEC with respect to three impacting factors: the trace size, the number and percentage of branch instructions, as shown in Figure 6.6. Figure 6.6(a) shows the relation between the size of the recorded trace and the execution time of H3. Figure 6.6(b) shows the relation between the number of executed branches and the size of the recorded trace. Figure 6.6(c) shows that relation between the percentage of executed branch instructions and the runtime overhead of H3. The results indicate that the performance of H3 is proportional to the percentage of executed branch instructions in the execution. Recall Column 8 in Table 6.2 that the number of branches in the application code often accounts for a small percentage of the total number of branches. Hence, in practice, the performance of H3 can be further improved by tracing only the application code and omitting external library calls.

Table 6.4: Results of Heisenbug reproduction. (-) means the solver runs timeout in one hour. Reprinted with permission from [4].

Program	#Var	CLAP #constraints		solve time	success?	H3 #constraints		solve time	success?
		#Total	#RW			#Total	#RW(**Reduction**)		
bbuf	79	14264	13902	98s	Y	10344	9982(**28.2%**)	52s	Y
sbuf	102	438	302	1s	Y	344	208(**31.1%**)	1s	Y
pfscan	25	199	60	1s	Y	179	40(**33.3%**)	1s	Y
pbzip2	113	5890	1270	2s	Y	5460	840(**33.9%**)	1s	Y
racey1	15040	540602	540388	-	N	50602	50388(**90.7%**)	267s	Y
racey2	30108	41612000	41607900	-	N	201202	200788(**99.5%**)	-	N
racey3	67850	1.3×10^8	1.3×10^8	-	N	451802	451188(**99.7%**)	-	N

6.5.3 Effectiveness of Bug Reproduction

Table 6.4 reports the results of Heisenbug reproduction. We successfully evaluated five benchmarks[2] with a total number of seven Heisenbugs. *racey1*, *racey2* and *racey3* correspond to the *racey* benchmark with 500, 1000, and 1500 loop iterations.

Column 2 reports the number of unknown variables in the constraint formula, corresponding to the number of read/write/synchronization operations in the symbolic trace. Columns 3-6 report the results of CLAP, including the total size of the generated constraints (in terms of the number of constraint clauses), the size of read-write constraints, the constraint solving time by Z3 and whether Z3 returns a solution before timeout in one hour. Columns 7-10 report the corresponding results of H3.

Overall, H3 is more efficient and effective than CLAP in reproducing Heisenbugs. The key difference between H3 and CLAP is that with the core-based constraint reduction, H3 generates a much simpler and smaller constraint formula than CLAP. H3 reduces the size of the CLAP constraints by 28%-99%, and is able to reproduce more bugs than CLAP. Both H3 and CLAP reproduce the bugs in the four benchmarks *bbuf*, *sbuf*, *pfscan* and *pbzip2*. H3 additionally reproduces the bug in *racey1*, while CLAP fails because the solver could not solve the constraints in time. In addition, for *bbuf*, although both H3 and CLAP can reproduce the bug, H3 is much faster (52s vs 98s) than CLAP. H3 fails on *racey2* and *racey3* because the constraints in these two cases

[2]We excluded *aget* and the *httpd* benchmarks because the KLEE symbolic execution failed on them.

are still too complex to solve.

6.6 Summary

We have presented H3, a novel technique that reproduces Heisenbugs by integrating hardware control flow tracing and symbolic constraint solving. With the efficient control flow tracing supported by PT, H3 enables for the first time the ability to efficiently reproduce Heisenbugs in production runs on commercial hardware. We have also presented an effective core-based constraint reduction technique that significantly reduces the size of the symbolic constraints and hence scales H3 to larger programs compared to the state-of-the-art solutions. Our evaluation on both popular benchmarks and real-world applications shows that H3 can effectively reproduce Heisenbugs in production runs with very small overhead, 4.9% on average on PARSEC.

7. CONCLUSION AND FUTURE WORK

This thesis makes contributions to verifying and debugging concurrent programs. For verifying concurrent programs, this thesis presents a series of stateless model checking techniques including model checking concurrent programs under relaxed memory models, improving the scalability of the model checker and reducing the redundant exploration of a model checker.

Comparing to the existing POR based model checking techniques, MCR uses SMT constraints to reason about the maximal causality between shared memory operations, which greatly improves the scalability and efficiency. However, MCR is limited to sequential consistency architecture. In this thesis, we present a work to check the correctness of concurrent programs under relaxed memory architecture, TSO and PSO. This work solves two key technical challenges: how to encode operational semantics of TSO and PSO as first-order logical constraints and how to deterministically execute the program following the generated TSO and PSO interleavings. To evaluate the effectiveness, we compare the work with one existing work based on DPOR and the other work SATCheck which also uses constraints analysis. The results show that our approach is much more effective than the other approaches for both state-space exploration and bug finding – on average it explores 5X to 10X fewer executions and finds many bugs that the other tools cannot find.

To further improve the scalability of MCR, I present MCR-S, which significantly reduces the size of the validity constraints built by MCR. This work is motivated by the fact that MCR is completely dynamic and it has to make every read before a target event to return the same value to guarantee the reachability of the target event. MCR-S mitigates the complex constraints by using static dependency analysis to identify a set of reads that the target event depends on. MCR-S considers both control dependency and data dependency from a whole program dependency graph. Compared to MCR, MCR-S reduces the number of the constraints and the solving time by 31.6% and 27.8%, respectively.

SE-MCR optimizes MCR by proposing a new equivalence checking, *switch equivalence* to further reduce the state space explored by MCR. This thesis presents a coarser equivalence,

which we call *switch equivalence*, to check if two seed interleavings are equivalent to each other. We formally prove that *SE-MCR* does not miss any states that MCR can produce and also avoids the redundant executions by MCR. We implemented the new algorithm based on MCR. Compared to the original MCR, SE-MCR reduces the number of explored interleavings by $58.6 and exploration time by 45.6$Compared to Nidhugg, which implements optimal-DPOR, SE-MCR is more efficient in reducing the size of the state space.

For debugging concurrent programs efficiently, this thesis presents H3, a new *record* and *replay* technique. H3 is motivated by the work CLAP which records thread-local trace and then building SMT constraints to compute the global failure interleaving. As CLAP does not need to record any cross thread communication, it reduces the runtime overhead. However, it is still not unsatisfactory to enable a production RnR system. To address this issue, we developed H3 using the hardware supported control-flow tracing, Intel PT, to record the thread-local trace. To our best knowledge, H3 is the first technique that integrates hardware control flow tracing with offline symbolic analysis for reproducing production-run Heisenbugs on commercial hardware. We develop a new core-based constraint reduction technique that significantly reduces the complexity of generated symbolic constraints from exponential in the trace size to exponential in the core counts. We implement and evaluate H3 on both popular benchmarks and real applications. Experiments show that H3 can reproduce real Heisenbugs in production runs with very small overhead.

Future Work The stateless model checker powered by MCR shows the advantages over the POR based approaches. In the future, I have interests in applying MCR to two important problems. So far MCR has only worked to concurrent programs using shared memories. It would be interesting to see how well MCR can work for message passing concurrency models. With the advent of the new programming language, Go [89] and the success of the open source projects Docker [90] and Kubernetes [91], which uses channels to communicate between threads, it would be interesting to verify the correctness of these softwares. Different from the traditional shared memory models, concurrent modules interact by sending messages to each other via communication channels. To make MCR work for such programs, it needs to encode the message passing events into SMT

constraints to reason about the ordering of these events.

Another direction along model checking I want to explore is how to verify critical systems via model checking. With the rise of blockchain and cryptourrency, a variety of smart contract applications emerge. Due to strategic motives of contract developers (i.e., "time-to-market" requirements) and blockchain system constraints, smart contracts are generally perceived as prone to exploitation and security breaches. Massive attacks have been launched to exploit online smart contracts, causing severe threats to financial stability. As smart contract applications are usually small and it is easy to generate non-deterministic ordering issues, it is good fit for model checkers to systematically checks the states of these applications and secure the program.

H3 achieves a significant performance improvement over CLAP by integrating hardware control-flow tracing with constraint analysis. Nevertheless, there are still several factors that can be leveraged to further improve the performance of H3.

First, On the current platform, the size of the PT trace buffer per core is limited to 4MB. For tracing long running programs, the buffer can get full quickly (e.g., 0.01s for the PARSEC benchmarks). Currently, Perf actively monitors the trace buffer and flushes it to disk once the buffer is full. To avoid overwriting the buffered data, Perf also needs to disable PT when the buffer is full, and wakes it up when the data is copied out. This is a main bottleneck that limits the runtime performance of H3 because the program execution has to be suspended when PT is off, otherwise the control flow data may be lost when the buffer data is being copied out. We also experienced data loss with Perf when using PT to track long traces. This happens because the speed of copying data out is not fast enough, causing certain buffered data overwritten by the new data. We expect that a larger trace buffer or double buffering in the future generations of PT will help alleviate this problem.

Second, another limitation of PT is that it only tracks the control flow of the program but not any data values or memory addresses. This is the main reason why symbolic execution is needed in H3 to construct symbolic traces. Although symbolic execution engines such as KLEE are becoming increasing powerful, scaling symbolic execution to long running programs remains a challenging

problem. In addition, limited by KLEE, H3 currently can only reproduce concurrency failures that occur in the application code, but not external function calls (though it traces the control flow in all external libraries). For future work, we plan to use hardware watchpoints (as also used in Gist [50]) to capture the value and address of variables along with the PT control flow tracing. With the value information, we can then skip the symbolic execution part but construct the constraints by matching the values of reads and writes directly. Moreover, this will further reduce the complexity of the generated constraints.

Third, although our constraint reduction is effective, the complexity of the generated constraints is still exponential in the number of cores. For long traces, the constraint size can still be large and solving them remains challenging. For this problem, we plan to improve H3 in two ways. First, we can perform periodic checkpoints (e.g., using the snapshot mode of Perf) to save the current state of the program, such that when a failure occurs, H3 needs only to generate the constraints from the last checkpoint to the failure. Second, we can reduce the amount of the trace by not tracing the control flow in the external libraries (e.g., using the IP filtering featured supported by Skylake processors). As shown in our experimental results, the branches from the application code account for only a small percentage (7-20%) of the total trace, most of which are from the external libraries. Skipping tracing the external libraries will greatly reduce both the trace size and the runtime overhead.

Last, similar to CLAP, currently H3 does not record the program input but assumes that all program inputs are fixed. If the program input is non-deterministic or certain program inputs are missed, H3 may fail to reproduce the bug. This problem can be addressed by tracking the program input and enforcing the same input value during the symbolic trace construction and the bug reproduction. Mozilla RR [92] is a promising solution to track non-deterministic inputs in real-world systems, by tracing only system call results and signals with `ptrace`. We expect that by integrating H3 with RR, H3 will be able to reproduce failures resulted from both non-deterministic schedules and inputs.

REFERENCES

[1] S. Huang and J. Huang, "Speeding Up Maximal Causality Reduction with Static Dependency Analysis," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pp. 16:1–16:22, 2017.

[2] S. Huang and J. 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.

[3] "A real-world bug caused by relaxed consistency.." `http://stackoverflow.com/questions/16159203/why-does-this-java-program-terminate-despite-that-apparently-it-shouldnt-and-d`.

[4] S. Huang, B. Cai, and J. Huang, "Towards production-run heisenbugs reproduction on commercial hardware," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 403–415, USENIX Association, 2017.

[5] "Understanding the dao attack.." `https://www.coindesk.com/understanding-dao-hack-journalists`.

[6] "An engineering disaster: Therac-25.." `https://en.wikipedia.org/wiki/Therac-25`.

[7] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[8] S. Owens, S. Sarkar, P. Sewell, and A. Better, "x86 memory model: x86-tso," in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

[9] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[10] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 533–536, 2007.

[11] J. Huang, P. Liu, and C. Zhang, "Leap: Lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2010.

[12] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid program analysis for determinism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2012.

[13] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[14] D. R. Hower and M. D. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[15] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA, 2008.

[16] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[17] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu, "Coreracer: A practical memory race recorder for multicore x86 tso processors," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[18] M. Xu, R. Bodik, and M. D. Hill, “A" flight data recorder" for enabling full-system multiprocessor deterministic replay,” in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 122–133, IEEE, 2003.

[19] J. 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.

[20] M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, 2007.

[21] J. Huang, C. Zhang, and J. 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.

[22] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[23] A. Valmari, “Stubborn sets for reduced state space generation,” in *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, 1991.

[24] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032. Springer Heidelberg, 1996.

[25] P. Godefroid, “Model checking for programming languages using verisoft,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1997.

[26] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[27] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[28] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Source sets: A foundation for optimal dynamic partial order reduction," *J. ACM*, vol. 64, Aug. 2017.

[29] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, "Data-centric dynamic partial order reduction," *Proc. ACM Program. Lang.*, vol. 2, Dec. 2017.

[30] S. Burckhardt, R. Alur, and M. M. K. Martin, "Checkfence: Checking consistency of concurrent data types on relaxed memory models," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[31] E. Torlak, M. Vaziri, and J. Dolby, "Memsat: Checking axiomatic specifications of memory models," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[32] B. Demsky and P. Lam, "Satcheck: Sat-directed stateless model checking for sc and tso," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2015.

[33] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[34] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "What's decidable about weak memory models?," in *Programming Languages and Systems*, pp. 26–46, Springer, 2012.

[35] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[36] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas, "Stateless model checking for TSO and PSO," *CoRR*, 2015.

[37] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

[38] B. Norris and B. Demsky, "A practical approach for model checking c/c++11 code," *ACM Trans. Program. Lang. Syst.*, vol. 38, May 2016.

[39] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for c/c++ concurrency," *Proc. ACM Program. Lang.*, vol. 2, Dec. 2017.

[40] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 2017.

[41] G. Altekar and I. Stoica, "Odr: Output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[42] Y. Chen and H. Chen, "Scalable deterministic replay in a parallel full-system emulator," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2013.

[43] D. Lee, M. Said, S. Narayanasamy, and Z. Yang, "Offline symbolic analysis to infer total store order," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 357–358, IEEE, 2011.

[44] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline symbolic analysis for multi-processor execution replay," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2009.

[45] P. Liu, X. Zhang, O. Tripp, and Y. Zheng, "Light: Replay via tightly bounded recording," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2015.

[46] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP, 2009.

[47] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[48] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang, "Order: Object centric deterministic replay for java," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC, 2011.

[49] J. Zhou, X. Xiao, and C. Zhang, "Stride: Search-based deterministic replay in polynomial time via bounded linkage," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE, 2012.

[50] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, 2015.

[51] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu, "Production-run software failure diagnosis via hardware performance counters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[52] X. Yuan, C. Wu, Z. Wang, J. Li, P.-C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, and Y. Guan, "Recbulc: Reproducing concurrency bugs using local clocks," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE, 2015.

[53] N. Machado, D. Quinta, B. Lucia, and L. Rodrigues, "Concurrency debugging with differential schedule projections," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, Apr. 2016.

[54] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, 2010.

[55] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1997.

[56] P. Godefroid, "Software model checking: The verisoft approach," *Formal Methods in System Design*, 2005.

[57] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs.," in *OSDI*, vol. 8, pp. 267–280, 2008.

[58] J. Huang, P. O. Meredith, and G. 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.

[59] T. F. Şerbănuţă, F. Chen, and G. Roşu, "Maximal causal models for sequentially consistent systems," in *Runtime Verification*, pp. 136–150, Springer, 2013.

[60] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, July 1990.

[61] A. Mazurkiewicz, "Trace theory," in *Petri nets: applications and relationships to other models of concurrency*, pp. 278–324, Springer, 1986.

[62] S. International, *The SPARC Architecture Manual: Version 8*. 1992.

[63] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.

[64] A. Roychoudhury, "Formal reasoning about hardware and software memory models," in *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM, 2002.

[65] T. Mitra, A. Roychoudhury, and Q. Shen, "Impact of java memory model on out-of-order multiprocessors," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2004.

[66] N. Zhang, M. Kusano, and C. Wang, "Dynamic partial order reduction for relaxed memory models," *In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[67] "Asm bytecode analysis framework.." `http://asm.ow2.org/`.

[68] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[69] J. Burnim, K. Sen, and C. Stergiou, "Testing concurrent programs on relaxed memory models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 122–132, ACM, 2011.

[70] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 337–348, 2014.

[71] 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.

[72] "Joana: Information flow control framework for java.." `http://pp.ipd.kit.edu/projects/joana/`.

[73] J. 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.

[74] "Wala.." https://github.com/wala/WALA.

[75] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON, 1999.

[76] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[77] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, 2012.

[78] K. E. Coons, M. Musuvathi, and K. S. Mckinley, "Bounded partial-order reduction," in *In Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 833–848, 2013.

[79] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 7–pp, 2003.

[80] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, 1996.

[81] "Intel PT Micro Tutorial." https://sites.google.com/site/intelptmicrotutorial.

[82] "The PARSEC benchmarks." http://parsec.cs.princeton.edu/.

[83] "Intel processor trace decoder library." https://github.com/01org/processor-trace.

[84] "Linux perf documentation." https://github.com/torvalds/linux/tree/master/tools/perf.

[85] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2008.

[86] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Context-sensitive correlation analysis for race detection," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 2006.

[87] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 485–502, 2012.

[88] "Racey:a stress test for deterministicexecution."
`http://pages.cs.wisc.edu/~markhill/racey.html`.

[89] "Golang.."
`https://golang.org`.

[90] "Docker.."
`https://www.docker.com`.

[91] "Kubernetes.."
`https://kubernetes.io`.

[92] "Mozilla rr."
`https://github.com/mozilla/rr`.