Concurrency Verification with Maximal Path Causality

Qiuping Yi
Texas A&M University
College Station, Texas, USA
qiuping@tamu.edu

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

ABSTRACT

We present a technique that systematically explores the state spaces of concurrent programs across both the schedule space and the input space. The cornerstone is a new model called *Maximal Path Causality* (MPC), which captures all combinations of thread schedules and program inputs that reach the same path as one equivalency class, and generates a unique *schedule+input* combination to explore each path. Moreover, the exploration for different paths can be easily parallelized. Our extensive evaluation on both popular concurrency benchmarks and real-world C/C++ applications shows that MPC significantly improves the performance of existing techniques.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; **Software testing and debugging**;

KEYWORDS

Concurrency, Verification, Dynamic Symbolic Execution

1 INTRODUCTION

The challenge of interleaving explosion has inspired a number of significant work [2, 10, 18, 19, 24] in testing and verification of concurrent programs. An essential idea is to identify redundant interleavings, which can be ignored because they produce equivalent program states. For example, in partial order reduction [2, 10, 18], an interleaving is identified as redundant if it can be generated from another interleaving by swapping non-conflicting events of different threads.

Maximal causality reduction (MCR) [19] is a more recent technique that minimizes redundant interleavings by exploiting the maximal causality between events in an execution trace with a constraint solver. A key idea of MCR is to capture the value of reads and writes in a trace and use the value to drive new executions, such that every new execution reaches a distinct program state. MCR is shown effective for finding extremely subtle bugs in both sequential and weak consistency models, such as TSO and PSO [21].

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236048

T1	T2	T3
1. i = input();	4. j = input();	7. **if**(x >= 100);
2. x = i;	5. x = j;	8. *error*;
3. x = 2;	6. x = 4;	9. **else** y = 2;

Figure 1: An example with inputs i and j.

However, all these techniques suffer from a serious limitation: they only explore program state spaces under a *fixed* program input, but leave the whole *input-space* except the fixed input unverified. In other words, they may fail to verify states that can only be reached by a certain combination of schedule and input.

Consider a simple example in Figure 1 with three threads and two inputs `i` and `j`. There is an error state at line 8. When input `(i,j)` is fixed to `(0,1)`, there are `15` interleavings totally. POR explores all the `15` interleavings; MCR explores only `4` because the program has only four different values for the read of `x` at line `7` (written by the four writes to `x` at lines `2-3` and `5-6`). However, neither POR nor MCR can find the error at line `8`. In fact, the error can never be exposed with the input `(0,1)`. One way to find the error is to check all possible inputs and for each input use MCR to explore the interleavings. Unfortunately, the input-space is huge: even if both `i` and `j` are restricted to [0, 100], to verify this program there are 150,000 executions that must be explored. It poses a significant challenge to verify both the input-space (M) and schedule-space (N). In theory, the whole search space is $M \times N$, where M is often infinite and N is exponential in the program size.

In this paper, we present a new technique, *Maximal Path Causality (MPC)*, that systematically explores the state spaces of concurrent programs while reducing redundant explorations across *both* the schedule space and the input space. The key idea is to combine MCR with dynamic symbolic execution (DSE), a well-known systematic path exploration technique. In DSE, each execution is symbolically analyzed to find the next input that triggers the execution of an unexplored path. Similar to that, MCR analyzes each execution trace symbolically and generates thread schedules that are not explored before. By combining these two techniques that have conceptual similarities, MPC can systematically explore both the input space and the schedule space at the same time.

More specifically, MPC captures both schedules and inputs as a set of constraints, which encode all *schedule+input* (*SI*) combinations that drive the program to the same path as one equivalency class. For each equivalency class, it generates a unique *SI* (through solving the MPC constraints) to explore each path once only. Back to the example in Figure 1. For input `(i=0,j=0)`, all the `15` schedules reach the same

path which takes the `false`-branch at line `7`. MPC only explores the path `p1={1-6,7,9}` once and ignores the other `14` schedules. After `p1` is explored, MPC generates a new *SI* to explore an unexplored path with path condition $R_x^7 \geq 100$, where R_x^7 denotes the value of x at line `7`. Thus, the schedule {`T1-T1-T1-T2-T2-T3-T3-T2`} with input `(i=0,j=100)` is generated to cover a new path `p2={1-5,7,8,6}`, which triggers the error. MPC terminates after two executions, because no other reachable (but unexplored) paths exist.

Moreover, MPC can be easily parallelized with two types of parallelism. First, dynamic executions for different *SI*s can be parallelized, because each execution is independent with previously explored paths. Second, the offline analysis for generating *SI*s for different paths can be parallelized, because the offline analysis only depends on the observed execution.

For the last decade, DSE has enabled addressing diverse software engineering problems – not only software testing [7] but also automated debugging [26, 33] and automated program repair [23, 25]. We expect that MPC can be used to extend many of those work done for sequential programs to concurrent programs, given the similarity of MPC to DSE.

We highlight our contributions as follows:

- To our knowledge, MPC is the first technique that systematically explores state spaces of concurrent programs while reducing redundant explorations across both input and schedule spaces.
- To ensure the verification soundness, a key challenge in MPC is how to systematically explore all reachable program paths. We present a sound path exploration algorithm based on the unsatisfiable cores of the MPC formulas.
- We evaluate MPC extensively on both popular benchmarks and real-world applications in C/C++. We show that MPC is significantly more effective and efficient than MCR and Con2Colic [17].
- Our MPC tool is open source at https://github.com/parasol-aser/MPC.

2 OVERVIEW

We start with a motivating example in Figure 2 (slightly more complicated than the example in Figure 1) to illustrate the challenges of verifying concurrent programs. We then use this example to illustrate how MCR works and identify its advantages and limitations. Finally, we show how MPC addresses these limitations and draw its overview.

2.1 Motivating Example

As shown in Figure 2(a), our motivating example contains three threads (`T1`-`T3`) which access two shared variables (`x` and `y`) and use a lock `l` to synchronize some (but not all) accesses to `y`. The program has an error at line `13`, which crashes the program when both the two branch conditions at lines `11` and `12` are satisfied. The error, however, is difficult to manifest because it requires a specific combination of thread schedule and program input (e.g., `i=3` and `j=2`).

Specifically, to satisfy the branch condition at line `11`, line `14` should be executed as the latest write to `x` before line `4`; meanwhile line `11` should be executed after line `4`. Note that, the condition at line `3` must be `true` to execute line `4`, which requires the input `i` to be 3. To satisfy the branch condition at line `12`, line `10` must be followed by line `18`, and at the same time no other writes to `y` should happen before line `12`. In addition, the input `j` must be larger or equal to `2` to ensure that the *while* loop at line `16` is executed at least twice. One such error-triggering schedule is `T1-T1-T1-T1-T2-T3-T3-T1-T1-T1-T1-T2-T3-T3-T3-T3-T3-T3-T3-T3-T3-T2-T2-T2`, corresponding to a path denoted by lines {`0-3,9,14,15,4-7,10,16-19,16-19,16,11-13`}.

To detect this error, a technique must find both a correct schedule and a correct input. For example, the error can never be revealed if the program input is fixed to `(i=0,j=0)` or `(i=3,j=1)`, no matter what thread schedule the program executes. This example shows the importance of cross-input verification of concurrent programs. However, existing state-space exploration techniques [2, 10, 18, 19, 24] all assume a fixed input.

2.2 Maximal Causality Reduction

MCR [19] is an effective stateless model checking approach for concurrent programs under a fixed input. A main advantage of MCR over the other popular approaches (e.g., DPOR [18] and context bounding [24]) is that it uses a maximal thread causality model (MCM [20]) to capture redundant schedules, such that only those unique schedules reaching distinct program states are explored. In other words, MCR never explores the same program state twice given a fixed input, and it ensures a provably *minimal* number of program executions for exploring all program states under the given input.

More specifically, MCR encodes each explored trace as a maximal causality formula Φ_{mcm} with first-order logical constraints. It uses Φ_{mcm} to generate new thread schedules to explore by enforcing a *new-state* condition: each new schedule must contain at least one new event, *i.e.*, a `Read` that returns a new value.

In Φ_{mcm}, each event e from an input trace τ is represented by an order variable O_e, and the order relation between O_e for different events is used to capture all the possible thread schedules that the same program (which generates τ) can execute in alternative runs. Φ_{mcm} is constructed by a conjunction of two subformulas: $\Phi_{mcm} \equiv \Phi_{sync} \wedge \Phi_{rw}$, where Φ_{sync} captures the ordering constraints determined by thread synchronizations, and Φ_{rw} the data-validity constraints over `Read` and `Write` events determined by memory consistency requirements (e.g., sequential consistency or relaxed consistency). Φ_{sync} can be further decomposed as a conjunction of the *must-happen-before* constraints Φ_{mhb} and the *lock-mutual-exclusion* constraints Φ_{lock}. For space reasons, we refer the readers to [19] for the encodings of Φ_{mhb} and Φ_{lock}, and focus on describing Φ_{rw}, which is extended in our new model to handle program inputs.

Data-validity constraints (Φ_{rw}). For an event e to be feasible in an inferred trace, MCM requires every read event r that

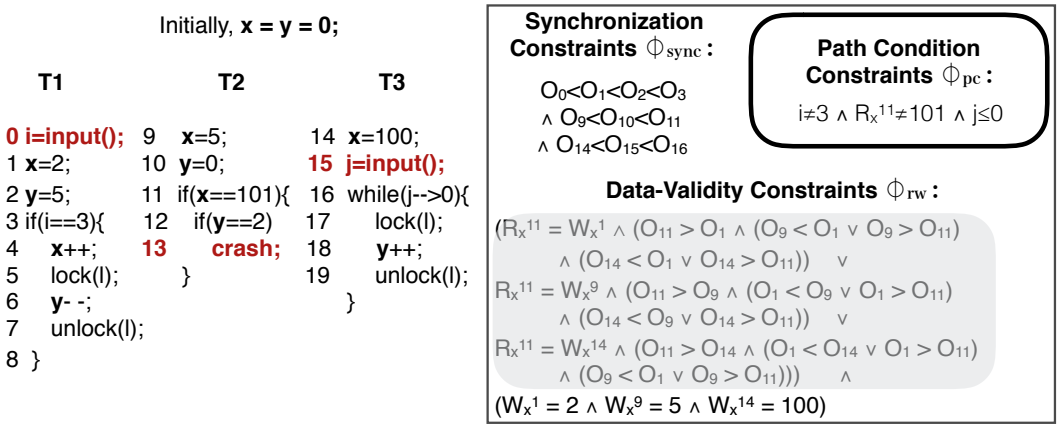

Figure 2: (a) The *crash* at line 13 can only be triggered by a specific combination of thread schedule and program input. (b) The constraints generated by MCR and MPC based on an execution with input (i, j) = (0, 0) and a random schedule.

must-happen-before e to return the *same value* as that returned by r in the input trace. Otherwise, e may become infeasible due to a conditional right after the read event r. For example, if in an observed trace a read r reads value 42, then in the inferred trace it must also read value 42. However, it can read the value 42 written by any write on the same address as long as the written value is 42 (not necessarily the same write as in the observed trace). We refer to this as the *data-validity condition*.

More formally, let $\prec_e$ denote the set of events that must-happen-before an event e. Consider a `Read` r in $\prec_e$ that accesses a memory location x and returns value v, and let W^x denote the set of `Writes` in the trace that write to x, and W^x_v those `Writes` in W^x with value v. The data-validity constraint of an event e, $\Phi_{rw}(e)$, is defined as $\bigwedge_{r\in\prec_e}\Phi_{value}(r)$, where $\Phi_{value}(r) \equiv$

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

The constraint $\Phi_{value}(r)$ enforces the `Read` r to read the value v written by any `Write` w in W^x_v (which writes v to the memory location x), subject to the condition that the order of w is smaller than that of r and there are no other `Writes` in between that write a different value to x.

It is worth noting that Φ_{rw} is recursive. Because in $\Phi_{value}(r)$, to enforce a `Read` r to read from a `Write` w, w must be feasible, which requires $\Phi_{rw}(w)$ to hold.

For this example, suppose the given input is `(i=0, j=0)` and the first schedule explored by MCR is τ_1={`0-3, 9-11, 14-16`}. The formula Φ_{mcm} generated by MCR is shown in Figure 2(b) (ignore the path-condition constraint Φ_{pc} for now). Let W^i_x and R^i_x denote the written and read values of x at line i respectively. R^{11}_x may return any value among W^1_x, W^9_x and W^{14}_x. If R^{11}_x returns W^1_x, then line `11` must be executed after line `1` and no other writes to `x` should happen between lines `1` and `11`.

Finally, MCR explores three executions, each corresponding to a schedule in which the only read at line 11 is matched with one of the three writes at lines 1, 9 and 14. Unfortunately, none of these three executions can trigger the crash at line 13. To trigger the crash, MCR would need a correct input (e.g., `i=3, j=2`). Nevertheless, even with the correct input, to hit the bug MCR still needs to explore 85 executions in our experiment.

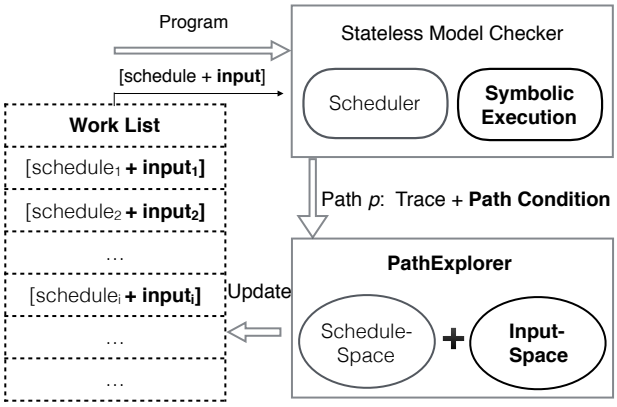

Figure 3: Overview of MPC.

2.3 MPC in a Nutshell

Inspired by MCR, our approach addresses two main problems. First, it extends MCR to handle program inputs, i.e., it not only generates new schedules, but also new inputs. Second, it advances MCR to capture not only redundant schedules, but also redundant inputs and redundant combinations of schedules and inputs. This is a major advance because the space of *schedule+input* (*SI*) combinations explodes much faster than the input-space or the schedule-space alone, yet significant redundancy often exists across the two spaces. We also note that our approach is applicable to a wide range of memory models (such as TSO and PSO), but we focus on sequential consistency (SC) in this paper.

Figure 3 shows an overview of MPC, which systematically explores all reachable paths in the program. At runtime, MPC performs both dynamic scheduling and symbolic execution to explore new program paths. Offline, it formulates constraints from the information observed at runtime, and generates new SI combinations (by solving the constraints) to explore new paths. By leveraging MCR, the advantage of MPC over conventional symbolic execution is that it does not need to reason about symbolic pointers on shared data, because concrete addresses of shared data accesses are all observed at runtime.

MPC maintains a *WorkList* of the to-be-checked SIs. The *WorkList* is initialized with a random SI (i.e., an empty schedule and a random input), and is augmented with new SIs generated from each iteration. In each iteration, the *Scheduler* consumes one SI from the *WorkList* to guide a concrete execution, accompanied by a symbolic execution to collect the path condition. Then, the offline *PathExplorer* finds unexplored paths or path prefixes across the schedule-space and input-space based on previously observed paths. For each such path, it further generates a corresponding SI (which is added to the *WorkList*) for exploring the path. MPC terminates when the *WorkList* is empty and there is no new SI that can be generated, meaning that all reachable paths have been explored.

Figure 4 illustrates MPC on the motivating example. To ease the presentation, we directly add the paths (instead of the corresponding SIs) into the *WorkList*. MPC starts with an input (`i=0, j=0`) and an empty schedule (which means the schedule can be arbitrary). Suppose path `P0` with the path condition $i \neq 3 \wedge R_x^{11} \neq 101 \wedge j \leq 0$ is explored in the first execution, which exhibits three new branches (`b3`, `b11`, $\texttt{b16}_1$[1]) and for all these branches the `false`-branch is taken, denoted by 0 in Figure 4(a).

Then, based on `P0`, MPC identifies 7 new path prefixes `I1`-`I7`, which are different combinations of the branch choices at `b3`, `b11` and $\texttt{b16}_1$, and which have not been explored. For each path prefix, MPC then tries to generate a concrete SI (via a constraint-based approach) to enforce an execution that follows the path prefix. Specifically, MPC uses an SMT solver (e.g., Z3 [15]) to solve the formula $\Phi_{mpc} \equiv \Phi_{sync} \wedge \Phi_{pc} \wedge \Phi_{dc}$, where Φ_{mpc} is the constraint constructed from the maximal path causality (MPC) model, which will be described in Section 3.2. Compared to Φ_{mcm} in MCR, Φ_{mpc} is a relaxation that captures the path condition Φ_{pc} and the data consistency Φ_{dc} over the reads and writes in the path prefix instead of the whole trace.

For example, for `I1` which contains the `false`-branch of `b3` and `b11` and the `true`-branch of $\texttt{b16}_1$, the path condition Φ_{pc} is $i \neq 3 \wedge R_x^{11} \neq 101 \wedge j > 0$, meaning that `i` is not equal to `3`, `j` is larger than `0`, and the value returned by the read of `x` at line 11, R_x^{11}, is not equal to `101`. The MPC constraints are shown in Figure 2(b). The MPC constraint Φ_{mpc} is a conjunction of the path-condition constraint Φ_{pc}, the synchronization constraint Φ_{sync}, and the *data-consistency* constraint Φ_{dc}. The synchronization constraints Φ_{sync} is the same as that in Φ_{mcm}. The data-consistency constraint Φ_{dc} concerns R_x^{11} and three writes to `x`, W_x^1, W_x^9 and W_x^{14}. For this example, Φ_{dc} (shown in the gray area) is similar to the data-validity constraint Φ_{rw}, except that it does not enforce the values of the three writes (i.e., $W_x^1 = 2 \wedge W_x^9 = 5 \wedge W_x^{14} = 100$).

[1]We use $\texttt{b16}_i$ to denote the `i`th instance of the branch at line `16`.

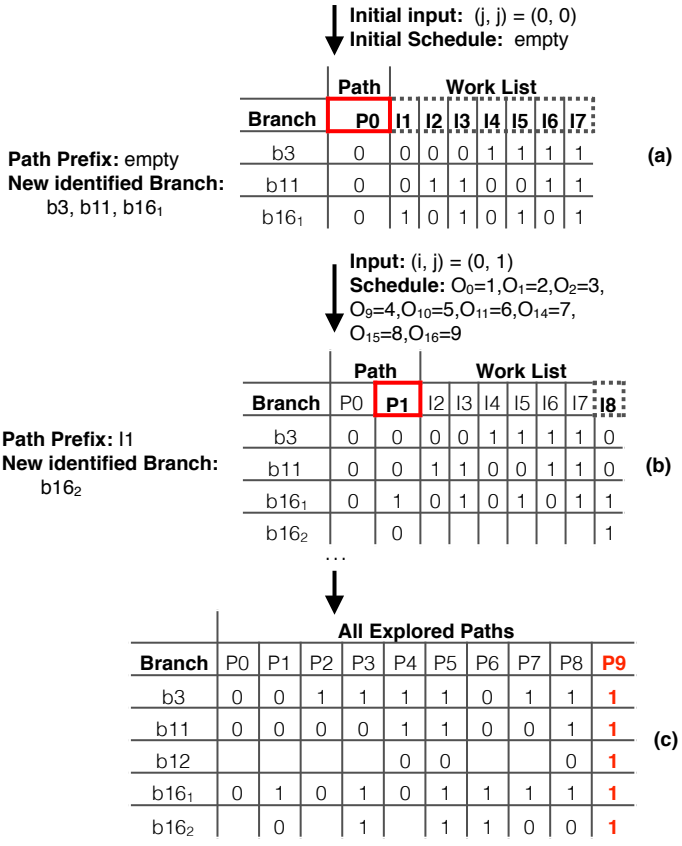

Figure 4: An illustration of MPC on the motivating example in Figure 2. (a) The state of the *WorkList* after exploring an initial input (`i=0, j=0`) and an empty schedule; (b) The state after exploring Path `P1`; (c) All explored paths after MPC terminates, containing the crash-triggering path `P9`.

One solution (i.e., an SI combination) to Φ_{mpc} is shown above Figure 4(b). The solution drives MPC to explore a new path `P1`, which exhibits a new branch $\texttt{b16}_2$ with its `false`-branch taken. Again, based on `P1`, MPC identifies a new path prefix `I8`, which extends `I1` with the `true`-branch of $\texttt{b16}_2$, and generates a new SI for it.

MPC repeats the previous analysis for each SI until the *WorkList* is empty. For our motivating example, MPC terminates after exploring `10` paths, including the crash-triggering path `P9`, as shown in Figure 4(c).

3 THE MPC APPROACH

In this section, we present the MPC approach in detail, including the basic definitions, the MPC model, the overall MPC algorithm, the path exploration algorithms, and the parallel MPC algorithm.

3.1 Basic Definitions

A central notion of our approach is the *path prefix* of a multithreaded program:

Definition 3.1. (Path and Path Prefix). A path p corresponding to a complete execution trace τ is defined as $p \equiv \{p(T_1), \ldots, p(T_N)\}$, where $p(T_i)$ is the executed path of thread T_i. $p' \equiv \{p'(T_1), ..., p'(T_N)\}$ is a path prefix of p, if for all threads $T_i (i = 1, ..., N)$, $p'(T_i)$ is a prefix of $p(T_i)$. Thus p' may be an incomplete path and $|p'(T_i)| \leq |p(T_i)|$, where $|p(T_i)|$ represents the length of $p(T_i)$. Each path p is also a path prefix of itself.

Definition 3.2. (Path Subsumption). Let $PS(p)$ denote the set of all paths with the same path prefix p. For two path prefixes $p1$ and $p2$, we call $p1$ *subsumes* $p2$ (or $p2$ is subsumed by $p1$) if $PS(p2) \subset PS(p1)$.

Based on the path prefixes, all paths can be organized into a *path tree*, defined as follows:

Definition 3.3. (Path Tree). The path tree of a multithreaded program is a *Tree(N, E)*, where the node set N represents path prefixes and the edge set E represents the following relations between nodes:

- For non-leaf node n, $\forall m \in children(n)$, n subsumes m, i.e., $PS(m) \subset PS(n)$.
- For non-leaf node n, $\forall a, b \in children(n)$, $PS(a) \cap PS(b) = \emptyset$, where $a \neq b$.
- For non-leaf node n, $PS(n) = \bigcup_{\forall m \in children(n)} PS(m)$.

In a path tree, the root represents all paths (because all paths share an empty prefix). Each leaf node represents one concrete path or an unreachable path prefix, and each non-leaf node is a reachable path prefix, which contains one or more paths. For example, Figure 5 shows the path three of the program in Figure 1. It contains three nodes: the root node representing path prefix p_0: *True*, and two leaf nodes representing the paths p_1: $R_x^7 \geq 100$ and p_2: $R_x^7 < 100$ respectively.

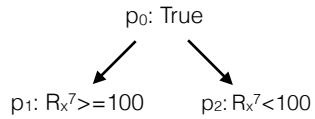

Figure 5: Path tree of the example in Figure 1.

3.2 Maximal Path Causality

In MCM [20], a multithreaded program is modeled as a prefix-closed set of finite traces that can be produced when completely or partially executed. A *trace* is abstracted as a sequence of *events* performed by threads on concurrent objects in a concrete execution, such as *Read/Write*, *Lock/Unlock*, etc. We note that the event *value* is also a part of the model. If the value of an event (e.g., the value returned by a read) is changed, it becomes a different event, such that a conditional branch after the event may produce a different trace.

Compared to MCM, a key difference of *Maximal Path Causality (MPC)* is that the event value is ignored in MPC. Because

```
   T1                T2
      Initially x=0
1. r1=x          3. x=1
2. r2=x
     assert(r1+r2<2);
```

Figure 6: Example of Maximal Path Causality.

the path condition of each thread is also captured in the trace, to ensure the feasibility of an event, it suffices to require that the path condition of the event is satisfied (instead of requiring the data-validity condition as in MCM). This relaxation not only significantly increases the power of MCM, but also reduces the complexity of the generated constraints. Consider an example in Figure 6. Suppose the input trace follows lines `1-2-3`, then the two traces `1-3-2` and `3-1` are also feasible in MCM. However, the trace `3-1-2` is not feasible, because in MCM the feasibility of the event at line 2 requires the read at line 1 to return the same value as that in the input trace (which is 0), but in the trace `3-1-2` it returns the value written by line 3 (which is 1). Therefore, to expose the assertion violation, MCR based on MCM must enforce a new execution following `3-1`, which explores the trace `3-1-2`.

However, the trace `3-1-2` is feasible in MPC. The reason is that the path condition for the event at line 2 is `true` (because there is no branch), so the event is feasible regardless of the value read by the event at line 1. Hence, the assertion violation in this example can be directly exposed in MPC without requiring a new execution.

Given a trace and the corresponding path condition for each thread, the MPC constraint Φ_{mpc} captures the maximal path causality among events in the trace and is defined as $\Phi_{mpc} \equiv \Phi_{sync} \wedge \Phi_{pc} \wedge \Phi_{dc}$. There are two types of symbolic variables in the path condition constraint Φ_{pc}: the value of program inputs and the value of reads on shared data. For example, in the path condition $i \neq 3 \wedge R_x^{11} \neq 101 \wedge j \leq 0$ of our motivating example in Figure 2, i and j are program inputs, and R_x^{11} is the symbolic value of reads. The first type can get arbitrary value allowed by program inputs. The second type can only choose certain values written by writes in the input trace, which is constrained by the data-consistency constraint Φ_{dc}. More specifically, consider a `Read` r, all the `Write` events on the same memory location, denoted by a set W, and the possible value returned by r, denoted by V_r. The data-consistency constraint for r is defined as $\Phi_{dc}(r) \equiv$

$$\bigvee_{\forall w_i \in W} (V_r = w_i \wedge O_{w_i} < O_r \bigwedge_{\forall w_j \neq w_i} (O_{w_j} < O_{w_i} \vee O_{w_j} > O_r)) \tag{1}$$

The constraint $\Phi_{dc}(r)$ states that if the `Read` r returns the value written by a `Write` w, then the `Write`'s order O_w must be smaller than the `Read`'s order O_r and there are no other `Writes` between them. The value of any `Write` w can be either concrete or symbolic, represented by the symbolic inputs or symbolic read values. The size of Φ_{dc}, in the worst case, is cubic in the size of the whole trace (i.e., linear in the number of reads and quadratic in the number of writes).

Algorithm 1: The MPC Algorithm

1: $si \leftarrow$ Random input and empty schedule;
2: $workList.add(si, True)$;
3: **while** ($!workList.empty()$)
4: $(si, prefix) \leftarrow workList.pop()$;
5: $\tau \leftarrow$ **GuidedSE** $(si, prefix)$;
6: $list \leftarrow$**GenerateNewSI(**$\tau, prefix$**);**
7: $workList.add(list)$;
8: **end while**

Compared to Φ_{rw} in MCM, Φ_{dc} in MPC is much simpler and is not recursive. There are two main differences. First, Φ_{dc} only specifies the possible values a read can return, but does not enforce it to return a specific value (as enforced in Φ_{rw}). Second, Φ_{dc} is constructed over events in a given path (instead of the whole trace). Since the path condition constraint Φ_{pc} ensures the feasibility of all events in the path, Φ_{dc} does not need the feasibility constraints for the matched writes (as required in Φ_{rw}).

3.3 The Basic MPC Algorithm

Our goal is to effectively explore the path tree of a concurrent program. Algorithm 1 describes the overall flow of our algorithm. The basic idea is to incrementally construct the path tree based on the MPC model. Each item $\langle si, prefix \rangle$ in the *workList* is used to drive an execution to follow the path prefix $prefix$ with the *schedule+input* (SI) combination si.

To start, the *workList* is initialized with an empty prefix (i.e., the root of the path tree), a random input and an empty schedule. In each iteration (lines 4-7), an SI from the *workList* is consumed, and two steps are performed: dynamic path exploration and static path-prefix identification. The first step carries out a guided symbolic execution along with the concrete execution triggered by the SI to collect the trace τ together with the path condition. In the second step, there are two important tasks: 1) identifying new path prefixes (e.g., unexplored branches) and 2) generating new SIs for each new path prefix. Task 1 is crucial to the verification soundness and it turns out to be highly challenging, which we will elaborate in the next subsection. Task 2 is based on the MPC model, for which we present our algorithm next.

Algorithm 2 describes our algorithm for generating the new SIs from a trace τ and a path prefix *prefix*. It first identifies a set of potential new path prefixes, P, based on τ and *prefix* (recall Section 3.4). For each new path prefix p in P, it constructs a formula $\Phi_{mpc}(p)$ of constraints to generate a corresponding SI (which should exist if p is feasible). $\Phi_{mpc}(p)$ corresponds to the MPC constraints of the path prefix p, which considers a subtrace of τ containing only those events in p. Any solution to $\Phi_{mpc}(p)$ produces a concrete thread schedule and a concrete input that can drive the program to execute the path prefix p.

It is important to note that the formula $\Phi_{mpc}(p)$ is sound, i.e., it only captures the space of feasible SIs with respect to path prefix p. Those events that are not on p are excluded from Φ_{dc} and Φ_{sync}. Consider an example in Figure 7. For exploring

Algorithm 2: GenerateNewSI(τ, *prefix*)

1: $SI \leftarrow \emptyset$;
2: $P \leftarrow$ **IdentifyNewPathPrefixes(**$\tau, prefix$**);**
3: **for each** $p \in P$
4: $\tau' \leftarrow$ extractSubTrace(τ,p);
5: Φ_{pc} = PathConditionConstraints(p);
6: Φ_{sync} = SynchronizationConstraints(τ');
7: Φ_{dc} = DataConsistencyConstraints(τ');
8: $\Phi_{mpc}(p) = \Phi_{pc} \wedge \Phi_{sync} \wedge \Phi_{dc}$;
9: $si \leftarrow$ **solve(**$\Phi_{mpc}(p)$**);**
10: **if** $(si \neq null)$ **then**
11: $SI.add(si, p)$;
12: **Return** SI.

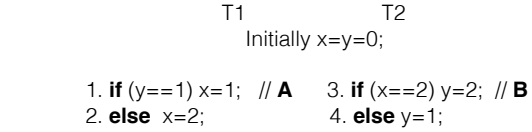

Figure 7: Example for explaining $\Phi_{mpc}(p)$.

path prefix AB with path condition $R_y^1 = 1 \wedge R_x^3 = 2$, the writes at lines 2 and 4 are not considered because they are not executed on this path prefix.

Another advantage of MPC over MCR is that the complexity of generated constraints can be significantly reduced. To generate a new schedule that manifests an event, MCR has to ensure that *all* dependent reads of this event (which are conservatively assumed in MCM as all events that must happen before this event) must return the same value as that in the observed execution. This often leads to a large number of data-validity constraints that are expensive to solve. However, such constraints are avoided in MPC.

3.4 Path Exploration

The key challenge in MPC is how to systematically explore the path tree such that it does not miss any reachable path. We first present an efficient and intuitively sound (but in fact unsound) algorithm. We then present a sound algorithm based on the *Unsat-Core* (UC) [8] of unsatisfiable MPC formulas.

3.4.1 Strategy 1: Combining unexplored path suffixes. Our first strategy is to combine unexplored path suffixes, which are exposed in the newly observed execution trace. This is a natural extension of how existing symbolic execution engine (e.g., KLEE) explores paths for sequential programs. For example, consider a trace τ with two threads driven by a path prefix *pre*. Suppose *pre* is extended with suffixes A and B for each of the two threads, respectively. That is, the newly explored path p is *pre*-A-B. Note that A and B are path conditions with respect to two branch choices. Hence, there are three new possible path prefixes identified by combining different branch choices: *pre*-$\overline{A}$-B, *pre*-A-$\overline{B}$ and *pre*-$\overline{A}$-$\overline{B}$, where $\overline{X}$ means the negation of X, i.e., the path follows the opposite choice of the corresponding branch.

More generally, let $split(pre, p)$ refer to the set of new path prefix combinations based on a newly explored path p and a corresponding path prefix pre, and let $suffix(pre, p, T_i)$ ($suffix(T_i)$ for short) denote the path extension for thread T_i from pre to p. Then $split(pre, p)$ contains new path prefixes formed by all combinations of $suffix(T_i)$ and its negations among all threads. For each individual thread, $suffix(T_i)$ may exhibit more than one new branch. For example, suppose $suffix(T_i)$=$b1$-$b2$-$b3$, then three path prefixes $\overline{b1}$, $b1$-$\overline{b2}$ and $b1$-$b2$-$\overline{b3}$ are identified. Note that other combinations such as $\overline{b1}$-$\overline{b2}$-$\overline{b3}$ are not valid because the execution of a branch may depend on the preceding branch choices. In total, $split(pre, p)$ contains $\prod_{i=1}^{N}(|suffix(T_i)|+1)$ path prefixes, where N is the number of threads and $|suffix(T_i)|$ the number of branches in $suffix(T_i)$.

On the surface, this strategy appears sound as it explores each combination of path prefixes once. However, it is unsound that it may miss certain reachable paths. Consider an example in Figure 8(a), which contains two inputs i and j, and one shared variable x. Let A, B and C denote the branches at lines 1, 2 and 4, respectively. Suppose the first explored path is p_0={T_1:A, T_2:$\overline{C}$}, where the branch at line 1 is *true* ($i > 0$) and the branch at line 4 is *false* ($R_x^4 \neq 2$). Three unexplored path prefixes are identified: p_1={T_1:A, T_2:C}, p_2={T_1:$\overline{A}$, T_2:$\overline{C}$}, and p_3={T_1:$\overline{A}$, T_2:C}. Among them, both p_1 and p_3 are unreachable (based on the constraints generated from the observed trace), and only p_2 is reachable. Hence, p_2 is further explored, which generates p_4={T_1:$\overline{AB}$, T_2:$\overline{C}$}. Based on p_4, a new path prefix p_5={T_1:$\overline{A}B$, T_2:$\overline{C}$} is identified.

The final path tree constructed by Strategy 1 is depicted in Figure 8(b). The error path $\overline{A}BC$ is missed. The reason is that Strategy 1 relies only on the observed trace to determine if a path prefix is reachable or not. However, the first trace τ_1 only observes the *true*-branch at line 1, and thus does not provide any information along the *false*-branch, which affects the branch decision at line 4.

3.4.2 Strategy 2: Unsat Core-based thread-independent exploration. The key idea behind *Strategy 2* is to distinguish unreachable path prefixes from those *overly-subsuming* path prefixes (defined below). For example, the path prefix p_3 in Figure 8(b) subsumes two concrete paths: an unreachable path {T_1:$\overline{AB}$, T_2:C} and a reachable path {T_1:$\overline{A}B$, T_2:C}, which is the error path. Strategy 2 splits such overly-subsuming path prefixes by discovering new behaviors through extending each thread independently. More formally, overly-subsuming path prefixes can be defined as follows:

Definition 3.4. (Overly-subsuming Path Prefix). Given a trace τ with N threads and its corresponding path p, and let $p(T_i)$ denote the path prefix along thread T_i. For path prefix pre, $p' \in split(pre, p)$ is an over-subsuming path prefix, if the following two conditions are satisfied:

- **C1**: $\Phi_{mpc}(p')$ is unsatisfiable.
- **C2**: There exists a thread set $Tset$, based on which a reachable path prefix p_{ext} can be constructed, where $Tset \subseteq Threads(UC(\Phi_{mpc}(p')))$ and for each T_i, $p_{ext}(T_i)$ is defined as follows:

$$p_{ext}(T_i) = \begin{cases} pre(T_i), & \text{if } T_i \in Tset; \\ p'(T_i), & \text{otherwise.} \end{cases}$$

The above conditions state that p' is an overly-subsuming path prefix if it is not reachable based on τ (by satisfying C1), but there exists a reachable path, p_{ext}, in which each thread T_i which follows $pre(T_i)$ or $p'(T_i)$ (by satisfying C2).

We propose to identify the overly-subsuming path prefixes based on the *root causes* of unsatisfiability (i.e., the UCs of the unsatisfiable formula). An Unsat Core (UC) of a formula F is a subset of the clauses in F that contribute to its unsatisfiability. For example, $\{x = 1, y = 1, x > 2, y < 0\}$ is an UC of the formula $(x = 1 \wedge y = 1 \wedge x > 2 \wedge y < 0 \wedge z > 0)$. Condition $z > 0$ is excluded from the UC, because it makes no contribution to the unsatisfiability.

Strategy 2 identifies the UCs of formula $\Phi_{mpc}(p')$ for each unreachable path prefix p', and maps them back to a thread set $Threads(UC(\Phi_{p'}))$, which contains threads that contribute clauses to the UCs. Then, it tries to explore a path prefix p_{ext} which extends the prefixes along all threads except those in $Threads(UC)$.

We note that to ensure soundness, all threads that contribute to the unsatisfiability of $\Phi_{mpc}(p')$ must be considered. Without the UC, one would have to extend the path prefix along all possible thread combinations to check the reachability, which may introduce many redundant explorations. Therefore, the use of the UCs is important to both the soundness and the performance of Strategy 2. In our implementation, we first consider the biggest UC because it may exclude the most threads from being extended, and thus avoiding unnecessary redundant explorations. When failed, to guarantee the soundness, we continue to consider the smaller UCs, until it succeeds or no more UCs can be further identified.

Consider again the example in Figure 8. Although there exists no SI to satisfy p_3={T_1:$\overline{A}$, T_2:C}, Strategy 2 finds that p_3 is an overly-subsuming path prefix. First, the basic condition $C1$ is satisfied because p_3 is unsatisfiable based on p_0. Second, $C2$ is satisfied, because it can generate an SI to extend T_1 along the path prefix $p_3(T_1)$=$\overline{A}$. Specifically, when p_3 is identified as unreachable by checking that $\Phi_{pc} = \{i \leq 0 \wedge R_x^4 = 2\}$ is unsatisfiable, Strategy 2 checks condition $C2$ as follows. First, it computes $UC(\Phi_{pc})$, which is $R_x^4 = 2$. Thus, the corresponding $Threads(UC)$ is $\{T_2\}$, which means that only thread T_2 contributes to the unsatisfiability. Then, Strategy 2 constructs path condition of p_{ext} as $p_3(T_1) \wedge True(T_2)$. Finally, a new SI is generated to drive the program to explore path p_{ext}={T_1:$\overline{A}B$, T_2:C}, which extends prefix $\overline{A}$ along T_1 as expected. Then p_3 is split into two new path prefixes: p_6={T_1:$\overline{A}B$, T_2:C}, and p_7={T_1:$\overline{AB}$, T_2:C}. At this time, p_6 is a reachable path prefix based on p_{ext}. Thus, it successfully generates a new SI combination for triggering an execution along path prefix p_6. Figure 8(c) shows the final path tree constructed by *Strategy 2* for this example. In total, 4 reachable paths are explored.

We next prove the soundness of Strategy 2:

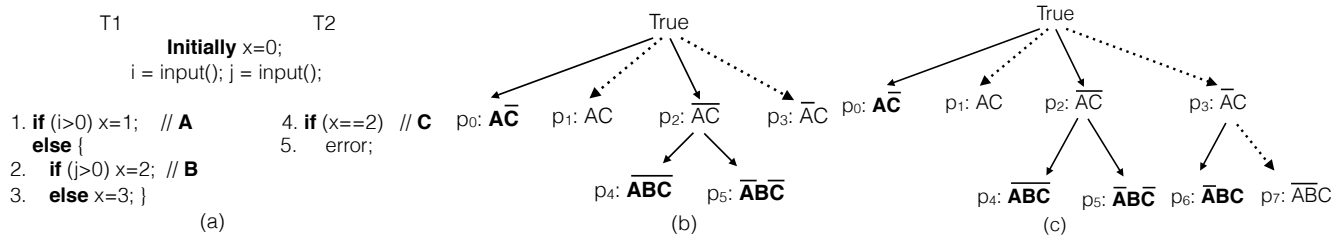

Figure 8: (a) An example. (b) Strategy 1 fails to explore the error path $\overline{A}BC$. (c) Path tree explored by Strategy 2.

Theorem 1 (Soundness). *Strategy 2 will explore all reachable paths if the solver is sound, i.e., for a satisfiable constraint formula the solver will always return a correct solution.*

Proof. By contradiction. Suppose a reachable path p is missed by *Strategy 2* (S2). There are only two possible reasons. First, p is not identified. Second, p is determined to be unreachable, i.e., no SI is generated to explore p. The first reason is impossible because upon observing any new path, all possible path prefix combinations are identified in $split(pre, p)$. For the second reason, since the solver is sound, then p must not be an overly-subsuming path prefix, which means that p violates C1 or C2. If C1 is violated, then an SI can be directly generated. If C2 is violated, then S2 must have missed a certain way to break the identified Unsat Core. However, S2 has enumerated all possible ways to break the root cause of an unreachable path prefix, described by set *Tset* in C2. Thus, p must be unreachable when S2 fails to further extend it, which contradicts with the initial hypothesis that p is a reachable path.

3.5 The Parallel MPC Algorithm

Different from other state-space exploration techniques [18, 24] which are difficult to parallelize, the MPC algorithm can be easily parallelized. Specifically, the online path exploration can be parallelized because it only depends on the provided SI. Similarly, the offline path identification can be parallelized, because it only depends on the trace information collected from the observed execution. Algorithm 3 describes the parallelized MPC algorithm. For each SI and the corresponding path prefix, Parallel-MPC first carries out a guided symbolic execution as in the sequential algorithm, and then invokes Parallel-GenerateNewSI to generate new SIs and path-prefixes in parallel. For each new SI and path prefix, it then starts a parallel instance of Parallel-MPC to continue exploring.

Algorithm 3: Parallel-MPC(si, *prefix*)

```
1: τ ← GuidedSE (si, prefix);
2: SI ← Parallel-GenerateNewSI(τ, prefix):
3: par-forall (si, p) ∈ SI
4:     Parallel-MPC(si, p);
```

4 EVALUATION

We implemented MPC for Pthread-based C++ programs based on KLEE [6] and Z3 [15]. MPC contains three main components: an application-level scheduler, a runtime tracer including a symbolic executor and an offline generator for SI combinations. The scheduler takes an SI as input to guide the thread execution by intercepting the critical events operated on shared variables, including locks and semaphores. The runtime tracer, implemented on KLEE, performs symbolic execution along the controlled execution to collect path conditions and the trace information, such as reads and writes on shared variables, and synchronization operations by Pthread library calls. When a new execution is completed, the offline generator reads the trace information and constructs the constraints. Then it invokes Z3 to identify newly unexplored paths. The output of the offline generator is a set of SIs to trigger the iterative analysis process. The parallelized implementation initializes a new iteration of path exploration whenever a new SI is generated and an idle worker thread is available.

To compare MPC with MCR, we have also implemented MCR for checking C/C++ programs (the original MCR [19] was implemented for Java).

We evaluated MPC with three sets of experiments. We next present our experimental results. All experiments were run on a four-core Linux machine with 2.7GHz Intel i5 CPU and 8GB RAM. The timeout for each benchmark was set to one hour.

4.1 Finding Tricky Concurrency Bugs

We first evaluated MPC for finding tricky bugs in a set of micro-benchmarks and a specially designed third-party benchmark, *racey* [31], which contains intensive races for evaluating concurrency bug finding and reproduction techniques. For the micro-benchmarks (code omitted due to page limit), we generated nine variants (M_1-M_9) of the example (M_0) in Figure 2 by varying the number of threads. M_i represents the program that contains i+3 threads, including the original threads (`T1`-`T3`), and i extra threads, each of which executes code {`int k=input(); if(k==10)x++;`}. In addition, the statement at line 11 in Figure 2 is changed to `if(x==i+101)` from `if(x==101)`. With more threads involved, the chance of triggering the crash at line 13 becomes smaller. To ensure program termination, the maximal value of j is set to 2.

For the comparison with MCR, because MCR does not handle program inputs, we conducted three types of experiments. First, providing MCR with a random input in each execution. Second, providing it with a fixed but incorrect input that cannot trigger the error. Third, providing it with a fixed correct input where `i` and `j` are set to 3 and 2 respectively, and other inputs to 10. This guarantees that MCR will trigger the error under certain schedules. For the first two types of experiments, our results confirm that MCR cannot find the error before timeout. In fact, when provided with a random input, MCR often fails at the runtime scheduling phase because the seed

Table 1: Results of finding tricky bugs. *TO*: timeout of one hour. ✓(✗): the error is found (not found).

Benchmark	#Paths			Time			Parallel-Time		SpeedUp	
	MCR	MPC-S1	**MPC-S2**	MCR	MPC-S1	**MPC-S2**	MPC-S1	**MPC-S2**	MPC-S1	**MPC-S2**
M_0	3/✓	6/✗	10/✓	14s.49	0s.31	0s.79	0s.26	0s.48	1.19	1.65
M_1	3/✓	12/✗	16/✓	2m13s.33	0s.71	1s.55	0s.39	0s.77	1.82	2.06
M_3	1/✗	48/✗	52/✓	TO	2s.98	5s.43	1s.64	3s.33	1.82	2.14
M_5	1/✗	192/✗	196/✓	TO	16s.23	38s.11	8s.88	20s.37	1.83	2.05
M_7	1/✗	768/✗	772/✓	TO	1m30s.28	13m50s.01	50s.66	3m17s.02	1.78	2.65
M_9	1/✗	3,072/✗	2,462/✗	TO	8m27s.32	TO	5m21s.43	22m42s.17	1.58	>2.64
Racey	432/✓	432/✓	432/✓	18m52s.51	4m27s.83	7m39s.65	2m25s	3m23s	1.72	2.26

interleavings (which are used to control the schedule) are computed for a different input. We next describe the results for the third type of MCR experiments only.

Table 1 reports the results. Overall, MPC with the sound path exploration strategy (S2) outperforms MCR consistently by finding the error with significantly fewer executions and more identified paths. In most cases (M_3-M_9), MCR runs timeout without finding the error even though the bug-triggering input is provided. For MPC with the unsound path exploration strategy (S1), although it is much faster than S2, due to the unsoundness it fails to find the tricky error. For MPC with S2, it finds all the errors except the timeout case in M_9.

For *racey*, due to its intensive races it typically computes different outputs in different runs. We seed an error by inserting an assertion at the end of the program to check if the program can produce a certain output. To ensure that the program has only finite states, we set the maximal number of checking the state of the barriers in each thread to 10. For this benchmark, all techniques complete with exploring the same number of paths, but MCR and S1 terminate with the most and fewest number of executions, respectively. However, MCR takes the longest time to finish the exploration, which is 4X longer than S1 and 2X longer than S2. The reason is that compared to MPC, MCR generates significantly more constraints (hence requires more time on constraint solving) and also incurs a much larger number of solver invocations (due to more executions).

Performance of Parallel MPC. Table 1 Columns 11-14 report the performance of the parallelized MPC on the same benchmarks. Compared to the sequential version, the parallel MPC achieves as much as 3.3X speedup on the four-core machine. Nevertheless, we note that our current implementation is not fully optimized. To balance parallelism and resource consumption, the verification tasks are not totally parallelized. In addition, the task partition and thread dispatching in our implementation introduce additional overhead.

4.2 Evaluation on concurrency libraries

We next evaluated MPC on a collection of real-world concurrency libraries, including Dekker's mutual exclusion algorithm (*dekker*), CAS-based Spinlock (*spinlock*), Linux reader-write lock (*linuxrwlock*), Michael and Scott lock-free queue (*MSQueue*), Linux reader-write lock (*linuxrwlock*), and Linux sequential lock (*seqlock*). For each concurrency library, we wrote a driver such that their inputs can be configured (i.e., not fixed). For all benchmarks, we also compared with MCR by running it with a fixed correct input.

Table 2: Results on real-world concurrency libraries.

Program	Setting	#Paths		Time	
		MCR	MPC	MCR	MPC
Dekker1	T=2,P=20	729	929	2m13s.34	1m24s.47
Dekker2	T=2,P=30	2,623	2,970	12m52s.34	5m46s.23
Spinlock1	T=2,K=5	363	522	15m49s.16	1m6s.38
Spinlock2	T=3,K=3	375	1,357	20m53s.72	3m28s.47
Linuxrwlock	T=2,K=5	116	1,910	TO	35m8s.39
MSQueue	T=2,K=5	152	539	TO	4m6s.12
Seqlock	T=3,K=3	204	3,877	TO	13m2s.51

Table 2 summarizes the results. T is the number of threads, P is the maximal number of attempts to enter the critical section to ensure termination (for *Dekker*) and K is the number of attempts to acquire and release the lock (for the others).

Overall, MPC significantly outperforms MCR on all these benchmarks in terms of efficiency and effectiveness. For most benchmarks, MCR explores many redundant paths (i.e., repeatedly explores the same path), because it generates schedules for unique states only, but multiple states can be exhibited by the same path. For MPC (with the sound path exploration strategy), it explores orders of magnitude more unique paths than MCR with less time or within the same timeout period.

4.3 Comparing with Con2Colic

We have also compared MPC with Con2Colic [17], a concolic testing technique for concurrent programs. Con2Colic maintains a list of interference scenarios that capture inter-thread data flow, and systematically explores program *branches* by enumerating all interference scenario candidates (ISC) upto a degree K (the number of inter-thread data flow edges in the ISC). Because an unrealizable ISC may become realizable after new states are explored in the future, a challenge faced by Con2Colic is memoizing all the unrealizable ISCs. It stores all ISCs in a global interference forest and upon a new branch is explored it validates the realizability of each ISC. This incurs both a large memory overhead and expensive validations of unrealizable ISCs. Differently, MPC does not memorize any states but tracks the path prefixes through constraint solving, and it models both the path condition constraints and thread interleavings in a uniform constraint formula, thus simultaneously checking all *SI* combinations that cover the same *path*.

Table 3 reports the results on the benchmarks collected from the ConCREST tool [12]. Because Con2Colic is not parallel, we compare it with the sequential MPC. Columns K and #*Runs* respectively report the largest degree of interferences and the number of executions explored by Con2colic. Column #*Paths* reports the number of paths explored by MPC. Note

Table 3: Results of MPC and Con2colic on real applications.

Program	Con2colic				MPC		
	K	#Runs	Success?	Time	#Paths	Success?	Time
aget	7	6	✓(3)	23s.6	4	✓(2)	13s
apache-a	15	411	✓(17/18)	1m32s	80	✓(6/37)	9s.53
apache-b	11	43	✓(16)	3s.24	22	✓(3)	1s.6
bluetooth	1	11	✓(7)	0s.8	36	✓(3)	2s.5
ctrace1	1	5	✓✗(4)	0s.75	10	✓(1/2)	3s.1
ctrace2	2	37	✓(4/5)	TO	393	✓(2/3)	2m23s
pfsan	-	-	✗(**OOM**)	-	112	✓(1/3)	3m39s
rbtree	-	-	-(**crash**)	-	1	-(no bug)	0s.5
sor	4	8	✓(1/2/7)	0s.8	4	✓(1/2/4)	1s.2
splay	17	34	-(no bug)	4s.6	13	-(no bug)	2s.3

that there is no correspondance between #*Runs* and #*Paths*. Because Con2colic is driven by ISCs rather than unique paths, Con2colic tends to explore many executions that exercise the same path. Column *Success?* reports if the technique detects the known bugs in the benchmark, and if yes ✓*(x/y)* shows the first bug was identified in the *x*th execution and the second bug was identified in the *y*th execution, etc. Column *Time* reports the total exploration time for each technique.

Overall, MPC is more effective than Con2colic for verification: MPC finds more bugs than Con2colic and incurs fewer executions to find the same bug. Our results also show that Con2colic is unsound, *e.g.*, it finishes without detecting the second bug in *Ctrace1* and it missed the bugs in *pfscan* due to out of memory. Although in three out of the ten cases, MPC takes longer than Con2colic to finish, MPC always explores more unique paths than Con2colic. For example, for *Apache-a*, MPC explores 1364 unique paths but Con2colic explores at most 411 paths (bounded by the 411 executions explored by Con2colic). Although MPC timeouts, it is able to verify 953 more paths than Con2colic. This also explains why MPC can find the second bug in *Ctrace1* whereas Con2colic cannot.

We also designed a micro-benchmark (Figure 9) to quantify the performance differences between MPC and Con2colic. The benchmark is simpler than the one used in Section 4.1, hence easier to appreciate the technical differences between MPC and Con2colic. It contains two threads each running in a loop for N times to read and write to a shared variable x. The assertion at the end of the program can only be violated if every execution of lines 4 and 5 in thread `T1` is interleaved by the execution of line 9 from the same loop iteration in thread `T2`.

Our results on the micro-benchmark shows that MPC is much more efficient and effective than Con2colic. When N is small (from 1 to 3), both Con2colic and MPC can identify the assertion violation. However, MPC takes significantly fewer executions and much less time to expose the bug than Con2colic. When N becomes larger than 3, Con2colic starts to run timeout in an hour without finding the bug, whereas MPC continues to identify the bug within a few seconds.

5 OTHER RELATED WORK

We would also like to compare with the SV-COMP [1] tools such as Yogar-CMBC [32] and Laze-CSeq [22] in future work. Most existing tools for concurrency verification are based on BMC [9, 28, 32] or lazy sequentialization [22, 27]. BMC uses bounded loop unfolding, and lazy sequentialization translates

```
        T1                                  T2
1.  for(int i=0;i<N;i++){          8.  for(int i=0;i<N;i++)
2.      int tmp = x;               9.      x++;
3.      tmp=tmp+1;
4.      x=tmp;
5.      if(x==tmp)
6.        x=0;
7.  }                      assert(x!=2*N);
```

N	Con2colic				MPC		
	K	#Runs	Success?	Time	#Paths	Success?	Time
1	2	5	✓(5)	0s.2	4	✓(2)	0s.1
2	3	12	✓(12)	1s	6	✓(3)	0s.4
3	5	123	✓(123)	6m47s	10	✓(4)	0s.7
4	4	462	✗	TO	18	✓(6)	1s.6
5	3	697	✗	TO	34	✓(22)	4s.3

Figure 9: Results of MPC and Con2colic on benchmark.

a multithreaded program into a nondeterministic sequential program. Different from MPC, these approaches target the interleaving space but not the input space.

Many improvements on partial order reduction (POR) [10, 11] have been proposed to optimize its performance and effectiveness, such as sleep set [18] and source set [2]. These methods have also been extended to handle weak memory models [3, 34]. However, they cannot handle different program inputs. Several other constraint-based approaches have been proposed for verifying concurrent programs, such as MemSAT [30], CheckFence [5] and SATCheck [16]. These approaches also assume a fixed program input.

Minion [29] proposes to synthesize concurrency tests with heuristics which also generate schedule+input to violate a given assertion. However, it is unsound that it provides no guarantee if the assertion holds or not when it cannot generate a test that falsifies the assertion. In addition, Minion requires sophisticated static analysis (e.g., data-flow analysis, alias analysis and program slicing), whereas MPC is purely dynamic (it uses concrete executions and dynamic symbolic execution).

Several approaches [4, 13, 14] based on schedule memoization have been developed to reduce redundant schedules across inputs. However, they are usually expensive because they require memoizing schedules for different inputs.

6 CONCLUSION

We have presented a new technique, MPC, for verifying concurrent programs by reducing redundant explorations across both the input space and the schedule space. We have also demonstrated with extensive evaluation that MPC significantly improves the performance of existing techniques such as MCR and Con2Colic. Moreover, since MPC shares the same workflow with dynamic symbolic execution, we expect that many testing, debugging and repair work developed based on symbolic execution for sequential programs can be naturally extended to concurrent programs by utilizing MPC.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for many insightful comments and constructive feedback. This work was supported by NSF awards CCF-1552935 and CNS-1617985.

REFERENCES

[1] [n. d.]. SV-COMP. http://sv-comp.sosy-lab.org. ([n. d.]).

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[4] Tom Bergan, Luis Ceze, and Dan Grossman. 2013. Input-covering Schedules for Multithreaded Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*.

[5] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. 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*.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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*.

[7] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[8] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. 2011. Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. *J. Artif. Int. Res.* (2011).

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

[10] E M Clarke, O Grumberg, M Minea, and D Peled. 1999. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer* (1999).

[11] E M Clarke, O Grumberg, M Minea, and D Peled. 1999. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer* (1999).

[12] ConCREST. last accessed May 2017. ConCREST. http://forsyte.at/software/concrest/. (last accessed May 2017).

[13] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient Deterministic Multithreading Through Schedule Relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[14] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. 2010. Stable Deterministic Multithreading Through Schedule Memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[16] Brian Demsky and Patrick Lam. 2015. 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*.

[17] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2Colic Testing. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*.

[18] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[19] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI*.

[20] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[21] Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[22] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. 585–602.

[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 691–701.

[24] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*.

[25] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. 772–781.

[26] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *ACM International Symposium on Software Testing and Analysis*. 199–209.

[27] Shaz Qadeer and Dinghao Wu. 2004. KISS: Keep It Simple and Sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. 14–24.

[28] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *26th International Conference on Computer Aided Verification*. 106–113.

[29] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. 2016. Directed Synthesis of Failing Concurrent Executions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 430–446.

[30] Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[31] Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*.

[32] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2017. Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification. *CoRR* abs/1708.08323 (2017). arXiv:1708.08323

[33] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *European Conference on Computer Systems*. 321–334.

[34] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*.