A Parallel Algorithm for Global States Enumeration in Concurrent Systems

Yen-Jung Chang
Department of Electrical and Computer Engineering
University of Texas at Austin, USA
cyenjung@utexas.edu

Vijay K. Garg
Department of Electrical and Computer Engineering
University of Texas at Austin, USA
garg@ece.utexas.edu

Abstract

Verifying the correctness of the executions of a concurrent program is difficult because of its nondeterministic behavior. One of the verification methods is predicate detection, which predicts whether the user specified condition (predicate) could become true in any global states of the program. The method is predictive because it generates inferred execution paths from the observed execution path and then checks the predicate on the global states of inferred paths. One important part of predicate detection is *global states enumeration*, which generates the global states on inferred paths. Cooper and Marzullo gave the first enumeration algorithm based on a breadth first strategy (BFS). Later, many algorithms have been proposed to improve space and time complexity. None of them, however, takes parallelism into consideration. In this paper, we present the first *parallel* and *online* algorithm, named ParaMount, for global state enumeration. Our experimental results show that ParaMount speeds up the existing sequential algorithms by a factor of 6 with 8 threads. We have implemented an online predicate detector using ParaMount. For predicate detection, our detector based on ParaMount is 10 to 50 times faster than RV runtime (a verification tool that uses Cooper and Marzullo's BFS enumeration algorithm).

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Validation

General Terms Algorithms, Verification

Keywords Global states; predicate detection; parallel algorithm

1. Introduction

One of the fundamental problems in concurrent and distributed systems is to determine whether a given condition may become true in one of the global states of the program. The condition could correspond to a bug or the negation of an invariant. The problem is challenging because the nondeterministic thread or process scheduling of the system may induce a different execution path in each run of the program; even for the same user input. Moreover, each execution path covers a different set of global states, and the number of global states may be exponential in the number of threads or processes in the system.

Predicate detection is a method that detects whether the user specified condition (predicate) could become true in any of the global states in an execution path of the program with a different thread schedule. The approach is predictive because it does not actually re-run the program in order to explore different execution paths. Instead, it generates inferred execution paths from the observed execution path. Then, it checks if the predicate can become true in any of the global states on the observed or inferred paths. The notion of predicate detection is first introduced by Cooper and Marzullo [6] for distributed debugging. Later, jPredictor [5] applies the notion for concurrent programs. In both works, the enumeration algorithm is general-purpose, i.e., it has no assumption on the predicate to be detected. Hence, the algorithm has to ensure that every global state is enumerated at least once.

We use the concurrent program in Figure 1 to explain the notion of predicate detection in concurrent systems. Note that the notion can also be applied in distributed systems. In the snippet of code, the synchronization between threads is accomplished via a Java monitor: thread 1 executes event $e1$, sets flag x, and then notifies thread 2 to execute $e2$. When the program is executed, the observed execution path is captured and converted to a partially ordered set (poset) of events. Assume that Figure 2(a) is the captured poset, in which the *causal dependency* between $e1$ and $e2$ is established by the Java monitor, because Java memory model guarantees that the updates in $e1$ can be seen when $e2$ is executed by thread 2. Note that causal dependency is a logical order between events rather than real-time order.

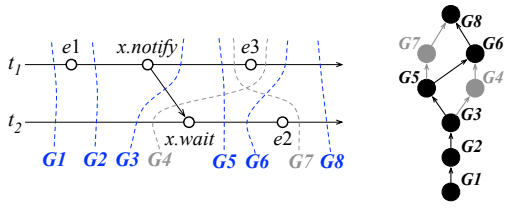
Informally, a global state in the poset is consistent if there exists an execution path to reach the state. In Figure 2(a), the dashed lines show all consistent global states of the poset. In the graphical representation, a global state contains the events on its left. For example, the global state $G4$ contains events $e1$, *x.notify()*, and $e3$. To reach $G4$, the events are executed in this sequence: $e1 \rightarrow$ *x.notify()* $\rightarrow e3$. Some global states may be reached by

```
// x initially is 0
THREAD 1                          THREAD 2

e1.execute();                     synchronized(x) {
synchronized(x) {                   while (x != 1)
  x = 1;                              x.wait();
  x.notify();                     }
}                                 e2.execute();
e3.execute();
```

Figure 1. A concurrent program which contains two threads that use a Java monitor to synchronize with each other.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688520



(a) (b)

Figure 2. (a) The captured logical order between events, which form a poset. The dashed lines are consistent global states of the program. (b) The relationship of the consistent global states.

```
1. predicate(GlobalState G) {
2.   for (int i = 1; i <= n; ++i) { // n is #threads
3.     for (int j = i; j <= n; ++j) {
4.       if (G[i] and G[j] are concurrent and conflict) {
5.           // a data race is found.
6.       }
7.     }
8.   }
9. }
```

Figure 3. A predicate to look for a pair of maximal events, which are conflict and concurrent, in the global state G.

multiple sequences of events. For instance, $G6$ can be reached by two sequences: 1) $e1 \rightarrow$ *x.notify()* $\rightarrow e3 \rightarrow$ *x.wait()*, or 2) $e1 \rightarrow$ *x.notify()* $\rightarrow$ *x.wait()* $\rightarrow e3$. The relationship of the consistent global states of the poset is shown in Figure 2(b). We can see that if the event sequences 1 and 2 reach global states $G4$ and $G5$, respectively, then they can reach $G6$ by executing events *x.wait()* and $e3$, respectively. So, we say that $G6$ is reachable from both $G4$ and $G5$.

When a program executes an event, the program reaches the next global state. Therefore, the execution path of the program can also be represented by a sequence of global states. Assume that $G1 \rightarrow G2 \rightarrow G3 \rightarrow G5 \rightarrow G6 \rightarrow G8$ is the observed execution path. Then the notion of predicate detection is to generate the global states $G4$ and $G7$ and hence the two inferred execution paths, $G1 \rightarrow G2 \rightarrow G3 \rightarrow G4 \rightarrow G6 \rightarrow G8$ and $G1 \rightarrow G2 \rightarrow G3 \rightarrow G5 \rightarrow G7 \rightarrow G8$, can be predictively verified without re-executing the program.

A predicate is defined to determine if the user specified condition could become true in a global state. We use the condition of data races to explain how a predicate is defined. A data race occurs when conflicting operations (e.g., a pair of read-write or write-write operations) are *concurrently* executed on the same memory address by different threads. If two executed events of different threads have no causal dependency in a global state, they can be executed concurrently. For instance, in the global state $G8$, the maximal events of thread t_1 and t_2 (i.e., $e3$ and $e2$, respectively) do not have causal dependency and can be executed concurrently. Figure 3 shows the predicate for detecting data races. The nested *for* loop at lines 2 and 3 gets all pairs of maximal events of the global state G; the symbols $G[i]$ and $G[j]$ at line 4 are the maximal events of thread t_i and t_j. Assume that the events $e2$ and $e3$ are the write operations to the same memory address, then a potential data race error is found in the global state $G8$.

In summary, predicate detection contains three parts. First, an observer captures the poset of events from the observed execution path of the program. Second, an enumeration algorithm takes as input the poset and generates all consistent global states of that poset, including the ones on inferred paths. Third, a predicate determines whether the user specified condition could become true in one of the consistent global states. *In this paper, we study the parallelism of the enumeration algorithm, which is the second part.*

The enumeration algorithm for global states can be applied in both concurrent and distributed systems. In this paper, the term *threads* would mean threads in concurrent systems or processes in distributed systems. Moreover, *global states* would mean consistent global states unless specified otherwise.

For certain classes of predicates, the computation time of enumeration algorithm can be reduced to polynomial time because only a partial set of global states needs to be enumerated [13, 16, 20, 24]. If no assumption is made on the predicate, enumerating all global states in the poset is unavoidable. Cooper and Marzullo [6] gave the first general-purpose enumeration algorithm, which guarantees to enumerate every global state at least once. The algorithm is based on a breadth first strategy (BFS) and requires exponential memory space and $O(n^3 \times i(P))$ time, where n is the number of threads in the poset P and $i(P)$ is the number of global states in P. Later, many general-purpose algorithms are proposed to optimize time and space complexity [2, 11, 12, 14, 17, 29, 30]. These algorithms, however, are single threaded and can only be used in an offline fashion for terminating programs. In this paper, we present the first *parallel-and-online* algorithm, named ParaMount, for global states enumeration. The source code of ParaMount is available on the website: http://pdsl.ece.utexas.edu/yenjung/.

ParaMount partitions the set of global states of a poset into multiple intervals (subsets) of global states; every global state belongs to exactly one interval. For each interval, ParaMount can use existing sequential enumeration algorithms as its subroutine without increasing the asymptotic work complexity. In this paper, we use the BFS [6] or the lexical [11, 12] enumeration algorithm for the subroutine. From the experimental results, ParaMount is 6 to 11 times faster than the original sequential algorithms when using 8 threads. The reason that ParaMount sometimes shows superlinear speedup is that partitioning the set of global states transforms the original problem into multiple sub-problems that are much easier to solve. Moreover, partitioning also reduces the memory space consumed by intermediate data which eliminates the running time wasted by Java garbage collector.

ParaMount is also an *online* enumeration algorithm, which can incrementally enumerate the global states during the construction of the poset. Because of this property, ParaMount can run along with the execution of concurrent or distributed programs and is applicable even to non-terminating programs such as web-server applications. Note that the online and parallel property of ParaMount can be applied together. Thus, it is possible to use ParaMount to perform an *online-and-parallel* predicate detection.

We evaluate the online property of ParaMount by conducting an online-and-parallel predicate detection of data races in concurrent programs. We compare our predicate detector with another general-purpose predicate detector, RV runtime [22] (the successor of jPredictor), and an online data race detector, FastTrack [10], using several benchmarks, e.g., *sor, tsp*, and *hedc* [5, 10, 33]. On average, our detector is 10 to 50 times faster than RV runtime. On the benchmark *raytracer*, RV runtime runs out of memory whereas our detector uses only 25% of the system memory. The performance of our detector is also comparable to that of FastTrack for most benchmarks even though the enumeration algorithm of ParaMount is not designed specifically for detecting data races.

The rest of the paper is organized as follows. Section 2 defines the poset of events and consistent global states. Section 3 presents the parallel algorithm – ParaMount – for global states enumeration. Section 4 gives the implementation of our online-and-parallel predicate detector. Section 5 shows the experimental results. Section 6 discusses the related work. Section 7 concludes the paper.

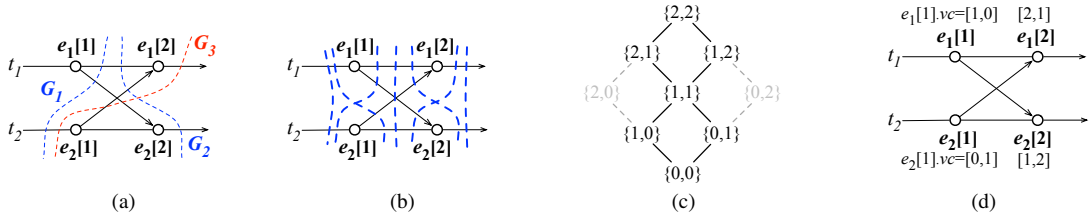


Figure 4. (a) A poset of events. The global states G_1 and G_2 are consistent but G_3 is not. (b) All consistent global states of the poset. (c) The set of global states of the poset. The grayed out global states are inconsistent. (d) The vector clocks of the events in the poset.

2. Definitions

ParaMount takes as input a poset of events and enumerates all consistent global states of that poset as discussed next.

2.1 Poset of Events and Consistent Global States

A concurrent execution is modeled as a poset $P = (E, \rightarrow)$ of events, which contains a set E of events together with Lamport's happened-before (HB) relation [18]. The HB relation is used to represent the causal dependency between events. In the rest of the paper, the HB relation is denoted by the symbol $\rightarrow$. The events that are executed on thread i are denoted as the sequence $E_i = e_i[1], e_i[2], \cdots$ of events, and the symbol e_i denotes an arbitrary event in E_i. Figure 4(a) shows a poset with two threads. The horizontal arrows represent the total order relation among the sequence of events that occur on the same thread.

A *consistent global state*, G, is a subset of E, such that if G includes any event f, then it also includes all events that happened before f [3]. Formally, G is a consistent global state if

$$\forall e, f : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G).$$

In Figure 4(a), the global state G_2 contains three events: $e_1[1], e_2[1]$, and $e_2[2]$. G_1 and G_2 are consistent but G_3 is not, because $e_2[1] \rightarrow e_1[2]$ but $e_2[1] \notin G_3$.

A global state can be identified by the maximal events of each thread, called *frontier*. The frontier can be simply written as an array of integers, in which the i-th integer is the index of the event in E_i. For instance, the global state G_1 in Figure 4(a) is represented by the frontier $\{e_1[1], e_2[0]\}$ or simply $\{1, 0\}$; the event $e_2[0]$ means no event is executed by thread t_2. The symbol $G[i]$ denotes the event of thread t_i in the frontier of G. For example, $G_1[1]$ refers to the event $e_1[1]$. The symbol $G(e)$ denotes a global state containing the event e in its frontier. Given a set of global states such that each of them contains e in its frontier, let $G_{min}(e)$ denote the *least global state*, and $G_{max}(e)$ denote the *greatest global state*. In Figure 4(a), G_1 is the least (smallest) and G_2 is the greatest (largest) global state of all global states containing $e_1[1]$ in the frontier.

All consistent global states of the poset in Figure 4(a) are shown by the dashed lines in Figure 4(b). Moreover, the relationship of these global states is illustrated in Figure 4(c), in which the inconsistent global states are grayed out. *The objective of ParaMount is to enumerate the set of consistent global states in parallel.*

2.2 Vector Clocks and Happened-Before Relation

In the poset P, each event has a vector clock (vc), which is an array of integers, and the information of the HB relation is stored in vector clocks. If a thread t_i executes an event e_i, then $e_i.vc[i]$ is the index of e_i in E_i. Moreover, for $j \neq i$, $e_i.vc[j]$ is the index of event e_j in E_j such that $e_j \rightarrow e_i$. For instance, in Figure 4(d), the vector clock $e_1[2].vc = [2, 1]$ contains the information: the index of the current event $e_1[2]$ is 2 and the event $e_2[1] \rightarrow e_1[2]$.

Algorithm 1 `ParaMountWorker`(P)

Input: A poset of events P.

```
1: while true do
2:     Event e ← P.getNextEventInTotalOrder→p()
3:     if e is null then break;
4:     G_min(e) = e.vc    ▷ Get least global state from e's vector clock.
5:     G_bnd(e) ← P.getBoundaryGlobalState()
6:     BoundedEnumeration(P, G_min(e), G_bnd(e), e)    ▷ Enumerate
       ∀G : G_min(e) ≤ G ≤ G_bnd(e) using Algorithm 2.
7: end while
```

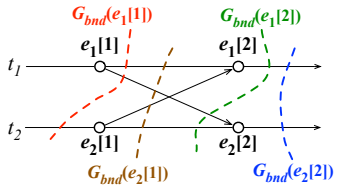


Figure 5. The boundary global states of the events in the poset. Assume that the total order among the events is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$.

During predicate detection, vector clocks are used as follows. First, the poset is constructed by logging the vector clock of every event. Second, the HB relation between events is extracted from the vector clocks and then used by the enumeration algorithm to generate all consistent global states of the poset. It is worth noting that the frontier of $G_{min}(e)$ can be extracted from $e.vc$ directly. For example, $G_{min}(e_1[1]) = \{1, 0\}$ and $e_1[1].vc = [1, 0]$.

3. ParaMount

3.1 Parallel Algorithm for Global States Enumeration

Algorithm 1 shows the worker procedure of ParaMount; each worker is executed by a thread during the enumeration. Before starting the workers, ParaMount determines a total order $\rightarrow_p$ among the events in the poset. Since the poset of events forms a directed acyclic graph (DAG), the order $\rightarrow_p$ can be calculated using any topological sort algorithm for DAG [7]. Because of the topological sort, this property holds:

Property 1. *For all events e and f, $e \rightarrow f \Rightarrow e \rightarrow_p f$.*

Moreover, two concurrent events e and f can be sorted in either $e \rightarrow_p f$ or $f \rightarrow_p e$. In other words, the total order $\rightarrow_p$ among the events is equivalent to the execution order of the events when the program is run on one single thread. For the poset in Figure 5, the four possible total orders among the events are:

1. $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$,
2. $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_2[2] \rightarrow_p e_1[2]$,

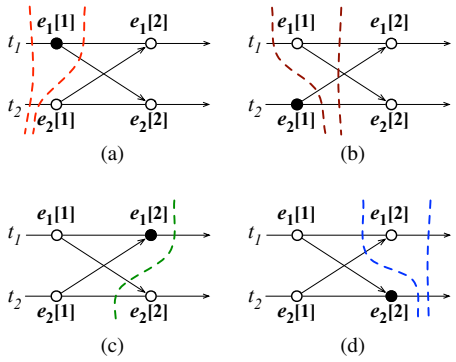


Figure 6. Assume that the total order among the events is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$, then we get (a) the interval $I(e_1[1])$, (b) the interval $I(e_2[1])$, (c) the interval $I(e_1[2])$, and (d) the interval $I(e_2[2])$ of global states. The global state $\{0, 0\}$ is a special case and always belongs to the interval of the first event in the total order $\rightarrow_p$, which is $e_1[1]$.

3. $e_2[1] \rightarrow_p e_1[1] \rightarrow_p e_2[2] \rightarrow_p e_1[2]$,
4. $e_2[1] \rightarrow_p e_1[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$.

Any one of the total orders can be used by ParaMount to partition the set of global states, which is performed from lines 2 to 5. The boundary of an interval of global states is defined by two global states G_{min} and G_{bnd}, which are determined with respect to the events in the poset. Specifically, ParaMount computes $G_{min}(e)$ and $G_{bnd}(e)$ for each event e. Then, any global state G such that $G_{min}(e) \leq G \leq G_{bnd}(e)$ is contained in that interval. Here, the comparison between two global states G and G' is defined as following:

$$G \leq G' \equiv \forall i : 1 \leq i \leq n : G[i] \leq G'[i].$$

At line 2, ParaMount gets the next event in the total order $\rightarrow_p$. If there are no more events, then all intervals are processed. At line 4, $G_{min}(e)$ is simply obtained from the vector clock $e.vc$. And $G_{bnd}(e)$ is defined as follows:

Definition 1. $G_{bnd}(e) = \{f \in E \mid (f = e) \vee (f \rightarrow_p e)\}$.

The examples of G_{bnd} are shown in Figure 5, in which we assume that the total order among the events is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$. For event $e_1[2]$, $G_{bnd}(e_1[2])$ includes all events that are totally ordered before $e_1[2]$. Hence, we get $G_{bnd}(e_1[2]) = \{2, 1\}$. For the same total order, we also get $G_{bnd}(e_1[1]) = \{1, 0\}$, $G_{bnd}(e_2[1]) = \{1, 1\}$, and $G_{bnd}(e_2[2]) = \{2, 2\}$. We next show that $G_{bnd}(e)$ is consistent:

Theorem 1. *$G_{bnd}(e)$ is a consistent global state for all event e.*

Proof. To show that $G_{bnd}(e)$ is consistent, we show that for any event $f \in G_{bnd}(e)$ if there exists any event g such that $g \rightarrow f$, then $g \in G_{bnd}(e)$.

From Property 1, $g \rightarrow f$ implies $g \rightarrow_p f$. Since $f \in G_{bnd}(e)$, we get $(f = e) \vee (f \rightarrow_p e)$ from Definition 1. If $(f = e)$, we get $g \rightarrow_p e$. And if $(f \rightarrow_p e)$, we get $g \rightarrow_p e$ because of the transitivity of $\rightarrow_p$. In both cases, $g \in G_{bnd}(e)$. □

An enumeration interval $I(e)$ of global states corresponding to any event e is formally defined as follows:

Definition 2. $I(e) = \{G \mid G_{min}(e) \leq G \leq G_{bnd}(e)\}$

Figure 6 shows the intervals of global states that are calculated for the events in the poset. In Figure 6(a), the global state $\{0, 0\}$ is a special case and is always enumerated by the first event in the total order $\rightarrow_p$, i.e., $e_1[1]$. At line 6 of Algorithm 1, ParaMount enumerates the interval of global state for the corresponding event.

Algorithm 2 `BoundedEnumeration`$(P, G_{min}(e), G_{bnd}(e), e)$

Input: A poset P, the new event e, and the least $G_{min}(e)$ and boundary $G_{bnd}(e)$ global state of e.

1: $G \leftarrow G_{min}(e)$ ▷ G: the current global state.
2: **while** $G \leq G_{bnd}(e)$ **do**
3: $predicate(P, G, e)$ ▷ Check the predicate upon G.
4: **if** $G = G_{bnd}(e)$ **then** break; ▷ Reached the boundary of $I(e)$.
5: $k \leftarrow n$
6: **for** $k \leftarrow n$ to $1 : G[k] \leq G_{bnd}(e)[k]$ **do** ▷ Select a new event e_k to add into G.
7: Event e_k = the next event on thread t_k.
8: **if** e_k is enabled **then** break;
9: $G[k] \leftarrow G[k] + 1$ ▷ Add event e_k into G.
10: **for** $i \leftarrow (k + 1)$ to n **do** $G[i] \leftarrow G_{min}(e)[i]$ ▷ Reset events due to lexical order.
11: **for** $i \leftarrow (k + 1)$ to n **do**
12: **for** $j \leftarrow 1$ to k **do**
13: Event e_j = the current maximal event on t_j.
14: $G[i] \leftarrow \max(G[i], e_i.vc[i])$
15: **end while**

3.2 Bounded Enumeration Algorithm

ParaMount can use existing sequential algorithm as its subroutine to enumerate the intervals of global states. However, the sequential algorithm needs to be modified (or bounded) to provide the two properties. First, it takes as input the boundary of an interval of global states and enumerates only the global states in the interval. Second, it enumerates each global state in the given interval exactly once.

We use the lexical enumeration algorithm in [11, 12] as an example to show the modification for the subroutine. Algorithm 2 shows the bounded lexical algorithm. The first modification is located at line 1: the least global state $G_{min}(e)$ of event e is used as the initial global state. The second modification is located at lines 2, 4, and 6: the boundary global state $G_{bnd}(e)$ is used to limit the global states that are enumerated by the algorithm. At line 3, the user specified condition is checked whether it can become true in current global state G. Lines 5 to 14 is a simplified implementation of the original lexical enumeration algorithm in [11, 12].

Lemma 1. *Given an event e in the poset P, Algorithm 2 enumerates every consistent global state G in the interval $I(e)$ exactly once.*

Proof. Suppose there exists a poset Q, which has an initial global state G_{init} and a final global state G_{final}. Lines 5-15 give the least consistent global state in lexical order as shown in [12]. Specifically, the *while* loop at line 2 enumerates every global state G of Q such that $G_{init} \leq G \leq G_{final}$. By the definition of $G_{min}(e)$, $G_{min}(e) \leq G_{bnd}(e)$. Algorithm 2 uses the property by assigning $G_{min}(e)$ to G_{init} and $G_{bnd}(e)$ to G_{final}. Hence, Algorithm 2 enumerates every consistent global state G of P exactly once such that $G_{min}(e) \leq G \leq G_{bnd}(e)$. □

In the evaluation section, the similar modification is applied into the BFS algorithm [6] for comparison.

3.3 Correctness of ParaMount

Now we show that every global state is contained in one of the intervals of global states (Lemma 2) and all intervals are disjoint (Lemma 3).

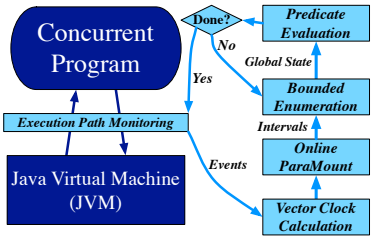


Figure 7. The framework of our online-and-parallel predicate detector.

Lemma 2. *In Algorithm 1, for every consistent global state G of the poset P, there exists an event e such that $G \in I(e)$.*

Proof. We show that for any consistent global state G in the poset P, there exists an event e such that $G_{min}(e) \leq G \leq G_{bnd}(e)$. Let e be the last event (with respect to the total order $\rightarrow_p$) in G. From the definition of $G_{min}(e)$, we get $G_{min}(e) \leq G$. Since e is the last event in G, for any event f in G, either $(f \rightarrow_p e)$ or $(f = e)$. Then from the definition of $G_{bnd}(e)$, we get $G \leq G_{bnd}(e)$. □

Lemma 3. *In Algorithm 1, for every consistent global state G of the poset P, there exists at most one e such that $G \in I(e)$.*

Proof. Suppose event e is the last event (with respect to the total order $\rightarrow_p$) in G. We now show that there does not exist any event $f \neq e$ such that $G_{min}(f) \leq G \leq G_{bnd}(f)$. The proof is by contradiction.

Suppose that $G_{min}(f) \leq G \leq G_{bnd}(f)$. Since $f \in G_{min}(f)$, we get $f \in G$. Because e is the last event in G, we get $(f \rightarrow_p e)$, which implies $(e \neq f) \wedge (e \not\rightarrow_p f)$. From the definition of G_{bnd} (Definition 1), we get $G \not\leq G_{bnd}(f)$, which is a contradiction. □

Theorem 2. *Algorithm 1 enumerates every consistent global state of the poset P exactly once when it uses Algorithm 2 as a subroutine.*

Proof. Follows from Lemma 1, Lemma 2, and Lemma 3. □

3.4 Work and Space Complexity

Now we analyze the work complexity of ParaMount when using Algorithm 2 as its subroutine. Suppose that the poset P consists of n threads, $|E|$ events, $|H|$ pairs of happened-before relation, and $i(P)$ global states. The work complexity of the topological sort is $O(|E| + |H|)$. For each worker, the work complexity is $O(n)$ because it has to store G_{min} and G_{bnd} at lines 4 and 5. Algorithm 2 takes $O(n^2)$ work for each global state because of the nested *for* loop at lines 11 and 12. Due to Theorem 2, Algorithm 2 cumulatively enumerates exactly $i(P)$ global states. As a result, the combination of ParaMount and Algorithm 2 takes $O(n^2 \cdot i(P))$ work, which is as the same as that of the sequential lexical algorithm. In this sense, ParaMount is work optimal.

As for space complexity, ParaMount uses $O(n)$ space for storing G_{min} and G_{bnd}. Hence, the total space complexity of ParaMount is $O(n \cdot |E|)$. Note that the existing general-purpose predicate detector, RV runtime [22], uses the BFS algorithm [6], which consumes memory space exponential in the number of threads in the poset. Thus, it could run out of memory even for a moderately sized benchmark.

4. Implementation of Online Predicate Detector

To evaluate the online property of ParaMount, we use it to build an online-and-parallel predicate detector. Figure 7 shows the framework of our predicate detector, which is described next.

Algorithm 3 `calculateVectorClock`(vc_i, vc_j)

Input: Two vector clocks for the calculation
Output: The vector clock for the new event
1: $vc_i[i] \leftarrow vc_i[i] + 1$
2: **for** $k \leftarrow 1$ to n **do**
3: $\quad vc_i[k] = \max(vc_i[k], vc_j[k])$
4: $vc_j \leftarrow vc_i$
5: return vc_i

4.1 Part I : Construction of Poset P

In the first part, the detector captures the events, which are relevant to the condition to be detected, and their causal dependencies from the observed execution path of the program. When the program is loaded into JVM in the first time, the detector uses bytecode injection technique [1] to inject monitoring instructions into the program during runtime. The injected bytecode are stored in memory, so the original Java program and Java bytecode are unmodified.

When the program starts, the operations of the program are captured as events. Then, the events along with their causal dependencies are converted into the poset P, in which the causal dependency between events is represented by Lamport's happened-before (HB) relation. The HB relation is established by the following rules [10, 19]:

1. **Process order**: If two events e and f are consecutively executed by the same thread, then $e \rightarrow f$.
2. **Lock-atomicity**: If event e corresponds to a thread releasing a lock and f corresponds to subsequent acquisition of that lock (including implicit locks and monitors), then $e \rightarrow f$. Note that any causal dependency that is induced by a Java monitor is also established by this rule.
3. **Fork-join**: One thread creates a new thread or waits for another thread to terminate.
4. **Transitivity**: If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

During bytecode injection, every thread and lock object is automatically attached with a vector clock. When a lock-atomicity or fork-join event is inserted into P, the vector clocks of different threads or locks are updated using Algorithm 3. For example, let event e correspond to the operation of a thread t acquiring a lock l, then the two vector clocks, $t.vc$ and $l.vc$ are passed as arguments to Algorithm 3, i.e., `calculateVectorClock`$(t.vc, l.vc)$. The returned vector clock is copied to the event's vector clock $e.vc$. If the inserted event is a process-ordered event, the vector clock of the thread is simply incremented and copied to the event.

4.2 Part II : Online Global States Enumeration

In the second part, the online detector uses ParaMount to enumerate global states along with the execution of the concurrent program. When an event e is captured, a callback function is triggered to insert e into P and to enumerate $I(e)$. Since multiple events may occur concurrently, the intervals of global states are enumerated in parallel. By default, the bounded lexical algorithm is used as the subroutine of ParaMount.

Algorithm 4 shows the worker of ParaMount which is modified for the online predicate detection. In comparison with Algorithm 1, there are two differences. First, the worker in Algorithm 4 is instantiated for each interval of global states. Second, the poset P is not a complete poset because of the online detection. Hence, the events in P cannot be topologically sorted at this point. Therefore, we use the order of insertion into the data structure of P at line 2 as the total order $\rightarrow_p$. Specifically, the atomic block from line 1 to 5

Algorithm 4 `OnlineParaMountWorker`(P, e)

Input: The new event e to be inserted into the poset P.

```
1: atomic {
2:     Insert e into the data structure of P.
3:     G_min(e) = e.vc
4:     G_bnd(e) ← P.snapshotOfMaximalEventsOfThreads()
5: }
6: BoundedEnumeration(P, G_min(e), G_bnd(e), e)   ▷ Enumerate the
   interval I(e) of global states using Algorithm 2.
```

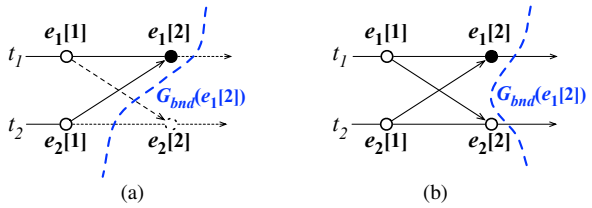


Figure 8. (a) One possible observed execution path when event $e_1[1]$ is inserted into P. Suppose the insertion order is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$. (b) Another possible observed path, in where the insertion order is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_2[2] \rightarrow_p e_1[2]$.

ensures that the events are inserted sequentially. Furthermore, the injected callback function ensures that a thread cannot execute the next event until it has successfully inserted the current event into P. Thus, Property 1 is achieved by the insertion order of the events.

At line 3, $G_{min}(e)$ is obtained from the vector clock of e, which is calculated using Algorithm 3. At line 4, $G_{bnd}(e)$ is determined by taking a snapshot of the maximal events of threads. Figure 8 shows an example of computing $G_{bnd}(e_1[2])$. In Figure 8(a), if the insertion order is $e_1[1] \rightarrow_p e_2[1] \rightarrow_p e_1[2] \rightarrow_p e_2[2]$, then the detector will not see $e_2[2]$ when taking the snapshot for event $e_1[2]$. Hence, our detector gets $G_{bnd}(e_1[2]) = \{2, 1\}$. Figure 8(b) shows another example. If events $e_1[1]$, $e_2[1]$, and $e_2[2]$ are inserted before event $e_1[2]$, then it gets $G_{bnd}(e_1[2]) = \{2, 2\}$. It is easy to see that the snapshot of maximal events satisfies the definition of G_{bnd} in Definition 1.

Since ParaMount allows multiple intervals of global states to be enumerated in parallel, we need to show that the combination of Algorithm 2 and Algorithm 4 can be executed concurrently.

Theorem 3. *Algorithm 2 and Algorithm 4 can be executed concurrently without violating correctness.*

Proof. The freedom from deadlock is obvious since the atomic block of Algorithm 4 can be implemented using one mutex with no wait inside the atomic block.

We now show that the execution of Algorithm 4 does not affect the concurrent executions of Algorithm 2. Suppose that Algorithm 2 is enumerating the global states corresponding to event e and Algorithm 4 is concurrently inserting event f. Since Algorithm 2 stops at $G_{bnd}(e)$, it does not require the information on f. Moreover, the only modification of the poset P happens in the atomic block of Algorithm 4. Hence, there is no interference between the two algorithms. □

4.3 Part III : Predicate Evaluation

We use the predicate for detecting data races as an example, because the condition is easy to understand and requires little knowledge about the concurrent programs. A data race occurs when a pair of conflicting operations (e.g., read-write or write-write operations)

Algorithm 5 `predicate`(P, G, e)

Input: A global state G and the new event e.
Output: the global state that contains data races.

```
1: if e.op = W then                                   ▷ e is a write event.
2:     for i ← 1 to n : e' ← G[i] do
3:         if (e'.op = W ∨ R) ∧ sameMemoryAddress(e, e') then
4:             // a data race detected.
5: else if e.op = R then                              ▷ e is a read event.
6:     for i ← 1 to n : e' ← G[i] do
7:         if (e'.op = W) ∧ sameMemoryAddress(e, e') then
8:             // a data race detected.
```

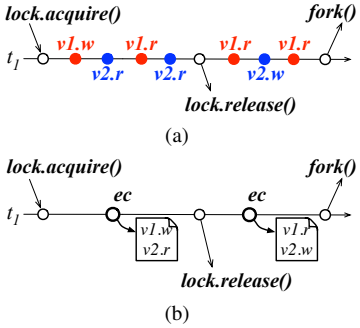


Figure 9. (a) The original process-ordered events. (b) Only the first write or read event of a variable in a sequence of process-ordered events is captured. Moreover, the events are merged into an event collection, ec.

Algorithm 6 `predicate`(P, G, e)

Input: A global state G and the new event e.
Output: the global state that contains data races.

```
1: if e.op = W then                                   ▷ e is a write event.
2:     for i ← 1 to n : ec ← G[i] do
3:         for all e' ∈ ec do
4:             if (e'.op = W ∨ R) ∧ sameMemoryAddress(e, e') then
5:                 // a data race detected.
6: else if e.op = R then                              ▷ e is a read event.
7:     for i ← 1 to n : ec ← G[i] do
8:         for all e' ∈ ec do
9:             if (e'.op = W) ∧ sameMemoryAddress(e, e') then
10:                // a data race detected.
```

is executed concurrently by different threads on the same memory address.

Algorithm 5 detects data races when the current event e is a write or a read event (line 1 and line 5). Assume that e is a write event. Then the *for*-loop at line 2 gets the maximal event e' of other threads. From the construction rules in Part I, two process-ordered events of different threads would not have direct HB relation. Therefore, any two process-ordered events in the frontier of global state can be executed concurrently. Subsequently, if events e and e' at line 3 are conflicting operations on the same memory address, then a data race has detected.

4.4 Other Implementation Details

Our detector captures only the process-ordered events that are relevant to the predicate, which are the read and write operations of variables. Moreover, multiple consecutive process-ordered events, which are executed by the same thread, are merged into one *event collection*. Two process-ordered events are considered consecutive

if there is no fork-join or lock atomicity event between them. The event collection only stores the first write operation of each variable. If there is no write operation for that variable, then its first read operation is stored. In addition, the events in the same event collection share the same vector clock.

Figure 9 shows an example. At the left side of Figure 9(a), thread $t1$ performs a write and then a read operation on variable $v1$. In addition, it performs two read operations on variable $v2$. Then, our detector only inserts the first write event for $v1$ and the first read event for $v2$ into P, as shown at the left side of Figure 9(b). The events in the event collection ec will share the same vector clock and ec is used as an event instance during the enumeration of global states. Algorithm 6 shows the modified predicate for the implementation. The loops at lines 2 and 7 retrieve the event collection on each thread, then the inner loops at lines 4 and 9 check whether the event collection contain any event that conflicts with the current event.

5. Evaluation

5.1 Experimental Results of ParaMount

Now we evaluate the performance of ParaMount, whose subroutine uses bounded BFS algorithm (which is modified from the BFS algorithm [6] and denoted by B-Para) or bounded lexical algorithm (which is modified from the lexical algorithm in [11, 12] and denoted by L-Para). Note that the BFS algorithm in [6] may enumerate the same global state multiple times. In this experiment, we have enhanced it with the technique mentioned in [12], so the BFS algorithm and the subroutine of B-Para enumerates every global state exactly once.

Table 1 shows the benchmarks that are used in the experiment. The benchmarks with the prefix *"d-"* are randomly generated posets for modeling distributed computations. The benchmarks *bank*, *tsp*, *hedc*, and *elevator* are the posets that are generated from real-world concurrent programs. The benchmark *banking* contains a typical error pattern in concurrent programs [8]; *tsp* is a parallel solver for the traveling salesman problem; *hedc* is a crawler for searching Internet archives; and *elevator* is a discrete event simulator for an elevator system. The benchmark programs *tsp*, *hedc*, and *elevator* are also used in [5, 10, 33]. Every program is run once and its execution path is converted to a poset of events using the rules that had been discussed in the implementation section. Then the enumeration algorithm takes as input the poset and outputs the set of global states of that poset. The column "n" shows the number of threads or processes in the poset.

Table 1 also shows the running times of the compared algorithms. The number of threads that are used by B-Para and L-Para are shown in the parentheses. All the experiments are conducted on a Linux machine with an Intel Core i7 1.6 GHz CPU and the heap size of Java virtual machine is limited to 2GB. The running time is wall-clock time measured in seconds.

From Table 1, BFS algorithm has the worst performance because of its expensive time complexity. Moreover, it failed to finish almost half of the benchmarks because it ran out of the available memory (o.o.m.). The reason is that BFS algorithm has to store intermediate global states for future enumerations and the number of the intermediate global states might grow exponentially in the number of threads in the worst case. In B-Para, the benchmarks *bank*, *hedc*, and *elevator* are able to finish because the set of global states are partitioned into multiple small intervals; each of which induces much fewer number of intermediate global states and hence the consumed memory can be less than 2GB.

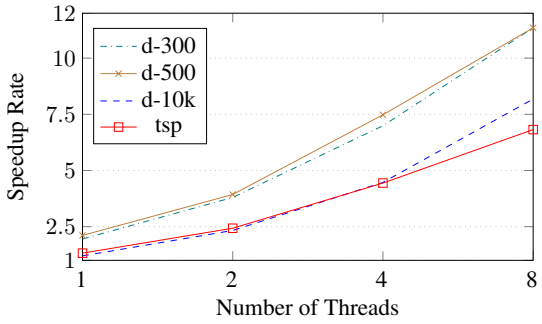


Figure 10. Speedup rate of B-Para with respect to the sequential BFS algorithm.

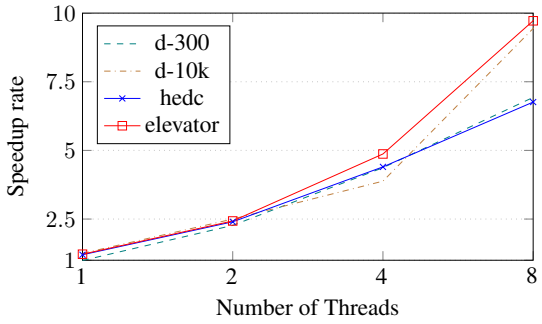


Figure 11. Speedup rate of L-Para with respect to the sequential lexical algorithm.

Partitioning the set of global states helps the performance of the original enumeration algorithm. Figure 10 show the speedup rate of B-Para with respect to the running time of BFS algorithm. The speedup rate on benchmarks *bank, hedc*, and *elevator* are not shown because BFS algorithm cannot finish the enumeration. When B-Para uses one single thread, its performance can be even faster than the original BFS algorithm. The reason is that BFS algorithm continuously triggers Java garbage collector to release the memory, which is used for storing the intermediate global states. In B-Para, the number of intermediate global states is reduced and hence the running time spent by Java garbage collector is significantly

Table 1. The information of benchmarks and running time (seconds) of BFS algorithm, lexical algorithm, and ParaMount.

Benchmark	n	#events	#global states	BFS	BPara(1)	BPara(2)	BPara(4)	BPara(8)	Lexical	LPara(1)	LPara(2)	LPara(4)	LPara(8)
d-300	10	300	42million	47.0	35.9	19.4	10.6	6.9	3.4	3.5	1.5	0.8	0.5
d-500	10	500	237million	380.8	195.4	100.5	54.5	33.6	17.8	15.3	7.6	3.9	2.1
d-10K	10	10,000	4,962million	8,599.1	4,089.0	2,190.5	1,150.4	757.7	406.8	327.3	163.4	105.0	43.1
bank	8	96	815million	o.o.m.	635.3	521.4	372.4	302.5	50.8	40.3	20.5	11.0	5.8
tsp	8	10,528	13million	8.6	7.1	3.7	1.9	1.1	1.6	1.5	0.8	0.4	2.3
hedc	12	216	4,486million	o.o.m.	10,850.7	10,182.2	8,032.5	4,646.9	487.4	406.5	203.3	110.8	72.1
elevator	12	38,528	27,643million	o.o.m.	28,655.3	13,903.2	6,985.4	3,696.2	4,233.8	3,491.6	1,742.7	870.2	435.7

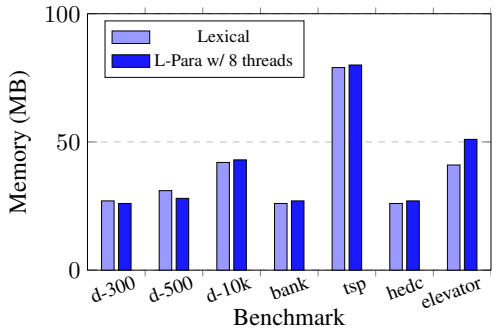


Figure 12. Memory usage of the lexical algorithm and L-Para.

reduced. Moreover, B-Para can be up to 11 times faster than BFS algorithm when using 8 threads.

Figure 11 show the speedup rate of L-Para with respect to the sequential lexical algorithm. We show 4 of the benchmarks because the other benchmarks have the similar trend. For lexical algorithm, partitioning the set of global states still helps the performance for most benchmarks. When using one single thread, L-Para can reduce 20% of the running time in average. When using 8 threads, L-Para can be 6 to 10 times faster than the original lexical algorithm.

Figure 12 shows the memory usage of lexical algorithm and L-Para. Since lexical algorithm is stateless, the memory is mainly used to store the poset, which is the input itself. Although ParaMount requires additional space to store $G_{min}(e)$ and $G_{bnd}(e)$ for each event e, the consumed memory is quite small. For most of the benchmarks, the memory usage of ParaMount is identical to that of the original enumeration algorithm.

5.2 Experimental Results of Online Predicate Detection

To evaluate the online property of ParaMount, we use it to implement an online-and-parallel predicate detector and then use the detector to detect data races in concurrent programs. In this experiment, the bounded lexical algorithm is used as the subroutine of ParaMount.

Table 2 shows the benchmarks that are used in the experiment. "LoC" shows the lines of code. "Thread" shows the number of threads that are used to drive each benchmark and ParaMount; after a thread executes an event, the thread is immediately used to enumerate the interval of global states. Thus, no additional threads are spawn for ParaMount. "#Var" shows the number of variables of the benchmark. Besides the concurrent benchmarks that are used in previous experiment, we also use the following benchmarks. Benchmarks *set (faulty)* and *set (correct)* are incorrect and correct implementations of the concurrent set [15]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *sor* is a scientific computation application; and *raytracer* is a benchmark for measuring the performance of a 3D raytracer. The benchmarks *sor* and *raytracer* are also used in [5, 10, 33].

We compare our online-and-parallel predicate detector (denoted as ParaMount) with another general predicate detector, RV runtime [22], and an online race detector, FastTrack [10]. We chose RV runtime because it is the successor of jPredictor [5] and it uses the notion of predicate detection. The enumeration algorithm that is used in RV runtime is the BFS algorithm [6]. We chose FastTrack because it is the fastest online race detector that uses the technique of vector clocks, even though its algorithm detects only data races. The input of each detector is a concurrent program and the output is a list of variables with data races.

The experimental results are shown in Table 2, in which the column "Base" shows the original execution time of the benchmarks. Each running time of ParaMount, RV runtime, and FastTrack includes the time to inject bytecode for monitoring, to execute the benchmark program, and to perform predicate detection. In RV runtime, bytecode injection and predicate detection are performed in offline; and in both ParaMount and FastTrack, they are performed in online. The running time is wall-time measure in milliseconds. The benchmark *elevator* contains several *sleep()* function calls, which dominate the overall running time, so its running time is almost the same on different detectors; except the one on RV runtime. The numbers of the variables that have data races are also shown in the table.

On average, RV runtime takes 15 seconds to inject the monitor instruments into the benchmark programs. Without considering the running time of bytecode injection, RV runtime still requires 15 seconds or more to finish predicate detection for most of the benchmarks while our predicate detector is able to finish within one second. In the benchmark *raytracer*, RV runtime ran out of the available memory because its BFS enumeration algorithm requires exponential memory space. Furthermore, RV runtime reported a false alarm on the benchmark – *arraylist1*. The reported variable is located in the test driver and its data race is benign; however, both our predicate detector and FastTrack can correctly rule out the variable. In *set (faulty)* and *set (correct)*, RV runtime reported several benign races. Moreover, it failed to detect the data race in *raytracer*. Currently, the results of RV runtime are not completely collected because the tool throws exceptions on some benchmarks.

When compared with FastTrack, the experiments show that ParaMount is as fast as FastTrack for most benchmarks even though its enumeration strategy is not specifically designed for detecting data races. In *set (faulty)* and *set (correct)*, the concurrent set uses

Table 2. The result of the data race detection.

Benchmark	Information			Running Time (ms)				# Detection		
	LoC	Thread	#Var	Base	ParaMount	RV runtime	FastTrack	ParaMount	RV runtime	FastTrack
banking	139	4	7	3	72	32,000	40	1	1	1
set (faulty)	223	4	10	61	152	37,000	428	1	3	1
set (correct)	260	4	10	94	110	39,000	468	0	3	1
arraylist1	1,474	4	6	3	7	exception	29	3	4[a]	3
arraylist2	1,377	4	16	4	5	exception	4	0	–	0
sor	255	4	20	19	81	41,000	179	0	0	0
elevator	547	4	23	16,000	16,000	83,000	16,000	0	0	0
tsp	702	4	36	7	114	exception	146	1	–	1
raytracer	1,885	4	77	32	1240	o.o.m.	998	1	0[b]	1
hedc	25,027	8	345	241	940	exception	1,140	4	–	4

[a] Acquired before the exception is thrown.

[b] The field with data races is not shown in the candidate list of RV runtime.

Table 3. Comparisons of the detectors.

Detector	Type	Poset Construction	Global States Enumeration	Predicate Assumption
ParaMount	Online	1-pass	Parallel	No assumption
RV runtime (jPredictor)	Offline	2-passes optimization	Sequential	No assumption
FastTrack	Online	1-pass	No enumeration involved	Data races

a single linked list to store the data; the linked list is synchronized using a fine-grained hand-over-hand lock-mechanism [15]. Whenever a new data is added to the set, a node object of the linked list is created. In *set (faulty)*, the variable `next` of a node has data races because the variable will be illegally accessed when a thread is adding a new entry and another thread is removing an existing entry.

In *set (correct)*, the access of the variable `next` is always protected by a lock. However, the variable `next` is initialized without the protection of locks; consequently, FastTrack reports the variable even if it is well protected in subsequent accesses. In our implementation, we do not consider initialization events to ever cause the data race since no other thread can have reference to uninstantiated object or variable. In this manner we avoid reporting benign races due to initialization. The source code and the proof of the correctness of the benchmark *set (correct)* are available in [15].

Table 3 lists the properties of the detectors that are used in this experiment. RV runtime is an offline predicate detector and hence it can construct the poset of events in 2-passes. It first logs the event on the observed execution path and then uses a pre-processor to optimize the poset of events with respect to the property of the predicate. The construction method [10, 19] used by FastTrack and ParaMount is 1-pass and hence is difficult to optimize; however, it can be used in an online fashion.

For enumeration of global states, RV runtime uses the BFS algorithm [6] to perform offline enumeration. The enumeration algorithm is general-purpose, which makes no assumptions on the nature of the predicate and guarantees that every global state is enumerated at least once. Unfortunately, the algorithm may enumerate the same global state multiple times. ParaMount is also general-purpose but it ensures that every global state is enumerated exactly once. FastTrack does not have any algorithm for global states enumeration, because its detection method is particularly designed for data races.

5.3 Limitations of the Online Predicate Detector

Our online predicate detector uses ParaMount for global states enumeration. Therefore, it guarantees that every global state of the observed execution path (i.e., the poset of events) is enumerated exactly once. However, the method [10, 19] that is used for capturing the HB relation between events does not consider the commuting of mutex. The problem can be solved by incorporating the technique of RichTest [19], which uses a thread scheduler to control the threads and changes the acquisition order of locks. The technique ensures that every re-execution of the program produces a new poset of events. Therefore, RichTest and our online predicate detector are complementary tools.

The second limitation is that even though ParaMount transforms the original lattice of global states enumeration into multiple intervals of global states, the total number of global states is still $i(P)$, which is exponential in the number of threads in the poset of events.

6. Related Work

6.1 Global States Enumeration

Cooper and Marzullo [6] gave the first algorithm for global states enumeration. The algorithm is based on breadth first search (BFS) algorithm and requires $O(n^3 \times i(P))$ time and exponential space in n, which is the number of processes in P. Alagar and Venkatesan [2] presented the notion of global interval which reduces the space complexity to $O(|E|)$. Steiner [30] gave an algorithm that uses $O(|E| \cdot i(P))$ time, and Squire [29] further improved the computation time to $O(log|E| \cdot i(P))$.

Pruesse and Ruskey [26] gave the first algorithm that generates global states in a combinatorial Gray code manner. The algorithm uses $O(|E| \cdot i(P))$ time and can be reduced to $O(\Delta(P) \cdot i(P))$ time, where $\Delta(P)$ is the in-degree of an event; however, the space grows exponentially in $|E|$. Later, Jegou et al. [17] and Habib et al. [14] improved the space complexity to $O(n \cdot |E|)$.

Ganter [11] presented an algorithm, which uses the notion of lexical order, and Garg [12] gave the implementation using vector clocks [9, 21]. The lexical algorithm requires $O(n^2 \cdot i(P))$ time but the algorithm itself is *stateless* and hence requires no additional space besides the poset.

None of the above-mentioned algorithms takes parallelism into consideration. ParaMount is the first parallel enumeration algorithm. Furthermore, ParaMount can enumerate the set of global states in an online fashion. Although Jegou et al. [17] had studied the algorithm for online global states enumeration, their algorithm cannot enumerate global states in parallel. The reason is that their algorithm incrementally builds the subset of global states using the information from previously built subsets.

6.2 Predicate Detection

jPredictor [5] is the first verification tool that applies the notion of predicate detection on concurrent systems. In jPredictor, the global states of poset are enumerated in an offline fashion using the BFS enumeration algorithm [6]. jPredictor focuses on the construction of poset and provides a flexible model for capturing the causal dependency between events. In this work, ParaMount provides an efficient parallel strategy for the global states enumeration. Moreover, ParaMount has been used to build an online-and-parallel predicate detector.

Predicates for detecting different kinds of condition have been extensively studied in the context of distributed systems. These predicates can be roughly categorized into conjunctive predicates [13], linear and semi-linear predicates [4], relational predicates [31], restricted temporal logics [24, 27] etc.

6.3 Testing and Debugging with Scheduler

CHESS [23], Java PathFinder [32], and RichTest [19] execute the program repeatedly and incorporate a scheduler to ensure that each run of the program explores a new execution path. CHESS and Java PathFinder schedule concurrent events into a totally ordered sequence and enable the corresponding threads to execute the events one at a time. The scheduler of RichTest directly changes the partial order among concurrent events. The approach retains the concurrency of events and hence they can be executed concurrently during the testing of the program.

Some tools combine the techniques of scheduler and the notion of predicate detection. These tools have two phases: prediction and replay. In the prediction phase, inferred execution paths are generated from the poset and are checked to determine if the predicate holds. In the replay phase, the program is re-executed

by using any thread schedule to execute the inferred paths and to determine whether the condition can really happen [25, 28, 34]. However, they have assumptions on the condition to be detected and they use heuristic strategies to enumerate global states.

7. Conclusion

We have presented the first parallel and online algorithm, named ParaMount, for global states enumeration. ParaMount can be run along with the execution of concurrent or distributed programs and hence is applicable even to non-terminating programs such as web-server applications. Moreover, ParaMount makes no assumptions to the nature of the predicate to be detected. It guarantees that every global state is enumerated exactly once.

From the experimental results, ParaMount can speedup the existing sequential algorithms up to 11 times with 8 threads. Moreover, we built a simple online-and-parallel predicate detector using ParaMount to detect data races in concurrent programs. We compare the detection results from ParaMount with those from another general-purpose predicate detector, RV runtime [22], and an online data race detector, FastTrack [10] on several benchmarks. Even though ParaMount is not specifically designed for detecting data races, the experiments show that it is still useful even for data races for most benchmarks.

Acknowledgments

This research is supported in part by National Science Foundation awards CNS-1346245 and CNS-1115808 and the Cullen Trust for Higher Education.

References

[1] ASM – a java bytecode engineering library. URL `http://asm.ow2.org/`.

[2] S. Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Trans. on Software Engineering*, 27: 412–417, 2001.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] C. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[5] F. Chen, T. F. Serbanuta, and G. Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

[6] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[9] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, 1988.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[11] B. Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

[12] V. K. Garg. Enumerating global states of a distributed computation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 134–139, 2003.

[13] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[14] M. Habib, R. Medina, L. Nourine, and G. Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN 9780080569581.

[16] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[17] R. Jegou, R. Medina, and L. Nourine. Linear space algorithm for on-line detection of global predicates. In *Proc. of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[19] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

[20] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[21] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 125–226, Chateau de Bonas, France, 1988.

[22] P. Meredith and G. Roşu. Runtime Verification with the RV system. In *the International Conference on Runtime Verification*, pages 136–152, 2010.

[23] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of Conference on Programming language design and implementation*, pages 446–455, 2007.

[24] V. A. Ogale and V. K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of International Symposium in Distributed Computing*, pages 420–434, 2007.

[25] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[26] G. Pruesse and F. Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

[27] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[28] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[29] M. B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

[30] G. Steiner. An algorithm to generate the ideals of a partial order. *Oper. Res. Lett.*, 5(6):317–320, 1986.

[31] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997.

[32] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2): 203–232, 2003.

[33] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[34] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 485–502, 2012.