McPatom: A Predictive Analysis Tool for Atomicity Violation using Model Checking

Reng Zeng, Zhuo Sun, Su Liu, and Xudong He

School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
{rzeng001,zsun003,sliu002,hex}@cis.fiu.edu

Abstract. Multi-thread programs are prone to bugs due to concurrency. Concurrency bugs are hard to find and reproduce because of the large number of interleavings. Most non-deadlock concurrency bugs are atomicity violation bugs due to unprotected accesses of shared variables by multiple threads. This paper presents a dynamic prediction tool named McPatom for predicting atomicity violation bugs involving a pair of threads accessing a shared variable using model checking. McPatom uses model checking to ensure the completeness in predicting any possible atomicity violation captured in the abstract thread model extracted from an interleaved execution. McPatom can predict atomicity violations involving more than three accesses and multiple subroutines, and supports all synchronization primitives. We have applied McPatom in predicting several known bugs in real world systems including one that evades several other existing tools. We provide evaluations of McPatom in terms of atomicity violation predictability and performance with additional improvement strategies.

1 Introduction

Multi-core hardware is a growing industry trend, for both high performance servers and low power mobile devices. Multi-thread programs can exploit multi-core processors at their full potential. In the real world, most servers and high-end critical software are multi-thread. Unfortunately, multi-thread programs are prone to bugs due to the inherent complexity caused by concurrency. It is difficult to detect concurrency bugs due to the huge number of possible interleavings. Many concurrency bugs escape from testing into software releases and cause some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity [1].

Among different types of concurrency bugs, atomicity violation bugs are the most common one. Atomicity violation bugs are caused by violations to the atomicity of certain code regions without proper synchronization. They widely exist in the real world systems and contributed to about 70% of the examined non-deadlock concurrency bugs [2]. Therefore, techniques for detecting atomicity violation bugs are extremely important.

This paper presents a dynamic prediction tool McPatom to predict atomicity violation bugs involving a pair of threads accessing a shared variable using model checking, based on binary executables that use POSIX thread library. McPatom uses memory access patterns instead of subroutine atomicity. The only input needed by McPatom is a binary executable, while source code is optional for locating bugs.

The McPatom framework contains the following major steps: (1) using Pin [3] to instrument an interleaved execution of a multi-thread program and to record an interleaved trace containing only atomicity violation impacting events including all shared variable accesses and all synchronization routines (locks, condition variables, barriers and thread management events); (2) projecting the interleaved trace into a partial order thread model of abstract threads, which maintains the causal relation within actual threads imposed by the synchronization routines; (3) automatically translating the partial order thread model into a Promela program for model checking in Spin [4]; (4) defining a complete set of atomicity violation patterns involving a pair of threads accessing every single shared variable and automatically translating them into temporal logic formulas; (5) using Spin to model check the atomicity violation patterns; and (6) mapping the violation reported in Spin to the execution trace in the original multi-thread program. Figure 1 gives an overview of McPatom framework.

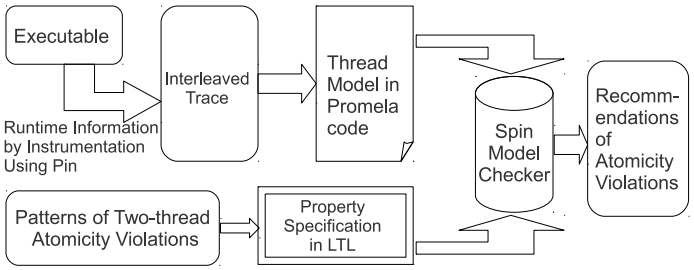

Fig. 1. Overview of McPatom Framework to predict atomicity violation bugs using model checking

Our work makes the following contributions:

1. A method to extract a thread model from an instrumented interleaved trace that only records events related to atomicity violations. Such an interleaved trace is much smaller than the program behavior in a complete execution. Furthermore the extracted thread model enables the checking of all alternative traces with the same causal relationships as the interleaved trace. The completeness of instrumented interleaved traces and the extracted thread models is proved.

2. A complete set of the patterns of unserializable interleavings involving two threads (most concurrency bugs involve only two threads [5]) containing any number of accesses to a shared variable (either user defined or every word sized dynamically allocated memory accessed by multiple threads). These patterns generalize and cover the three accesses proposed in [2][6]. These atomicity violation patterns become property specifications to be checked.
3. A unique prediction tool - McPatom, for detecting atomicity violation bugs through model checking. McPatom instruments interleaved executions, extracts thread models from interleaved traces, automatically converts (1) thread models into Promela programs and (2) atomicity violation patterns into property specifications. By constraining the checking within a pair of threads involving one shared variable at a time, the interleaving space to be checked is vastly reduced. As a result, McPatom is applicable to large software systems. McPatom can predict atomicity violations that do not manifest during testing or runtime.

We applied McPatom to predict several known atomicity violations in real world systems as well as an atomicity violation that cannot be detected by several existing tools. We obtained favorable experimental results with regard to atomicity violation predictability, accuracy and performance of using McPatom.

2 Extracting Partial Order Thread Models from Multi-thread Program Executions

2.1 Description of the Partial Order Thread Model

A multi-thread program has a set of threads and a set of shared variables. Shared variables are addresses of global variables and every word sized dynamically allocated memory accessed by multiple threads. The same memory address is considered as another shared variable if it is released and reallocated through the invocations of memory functions. An execution $\sigma = s_1, ..., s_n$ of a multi-thread program P is a sequence of executed statements. A trace is the projection of an execution to a sequence of annotated shared variable accesses and synchronization events. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle tid_i, timestamp_i, action_i \rangle$ in which tid_i is a thread handle, $timestamp_i$ is a time stamp based on real time and $action_i$ is one of the following: (read/write, a shared variable), (a synchronization routine, a synchronization variable) or (a thread management operation, a thread handle). McPatom uses POSIX Threads in which a synchronization routine is a routine related to semaphores, mutex locks, condition variables and barriers, does not handle user-defined synchronization primitives. McPatom also assumes a shared variable as a synchronization variable if it is accessed by synchronization routines, thus does not treat its accesses as shared variable accesses.

Lemma 1. *A trace $\tau = e_1, ..., e_m$ extracted from an execution sequence $\sigma = s_1, ..., s_n$ is sound and complete with respect to σ in terms of atomicity violation predictability.*

Proof. (1) Soundness: An atomicity violation revealed in τ must exist in σ. This is obvious since τ is a projection of σ. An atomicity violation pattern appearing in τ exists in σ.

(2) Completeness: Any existing atomicity violation in σ remains in τ. Since atomicity violations do not depend on general program states, and only depend on the execution orders of shared variable accesses and synchronization events, that are completely captured in τ.

Definition 1. *Given a trace $\tau = e_1, ..., e_m$ containing shared variable accesses and synchronization events, a partial order thread model $(E_\tau, \prec)$ is defined as follows:*

1. $E_\tau = \{e_i \mid e_i \; in \; \tau\}$
2. $\prec$ *is a partial order relation such that, for any* $e_i, e_j \in E \; (i \neq j)$, $e_i \prec e_j$ *iff*
   (a) $tid_i = tid_j$ *and* $i < j$, *or*
   (b) $tid_i \neq tid_j$, $action_i = (Signal, cvar)$, $action_j = (Wait, cvar)$ *and* $\forall k \,.\, ((j < k < i) \wedge (action_k \neq (Signal, cvar))$ *in which cvar is a condition variable, or*
   (c) $tid_i \neq tid_j$, $action_i = (Wait, bvar)$ *and* $(i < j) \wedge \exists k . ((tid_k = tid_j) \wedge (k < j) \wedge action_k = (Wait, bvar) \wedge \forall h \,.\, ((tid_h = tid_k) \Rightarrow \neg(k < h < j)))$ *in which bvar is a barrier variable, or*
   (d) $tid_i \neq tid_j$, $action_i = (Create, tid_j)$, *or*
   (e) $tid_i \neq tid_j$, $action_j = (Join, tid_i)$.
3. *Mutual exclusion: for any* $e_i, e_j, e_m, e_n \in E \; (i \neq j \neq m \neq n)$, $e_j \prec e_m$ *or* $e_n \prec e_i$ *iff*
   (a) $tid_i = tid_j$, $action_i = (Lock, lvar)$, $action_j = (Unlock, lvar)$, *and*
   (b) $tid_m = tid_n$, $action_m = (Lock, lvar)$, $action_n = (Unlock, lvar)$.

The above partial order relation (or simply causal relation) is similar to the happened-before relation given in [7]. From the above definition, we have (1) shared variable accesses within the same thread are ordered, and (2) a pair of shared variable accesses from two different threads are only ordered if and only if they are constrained by some intermediate synchronization events such as one thread creating the other.

While the partial order thread model $(E_\tau, \prec)$ respects the causal relation in trace τ, it captures an equivalent class of alternative traces that obey the same causal relation as τ, in which each alternative trace τ' is a result of rearranging some shared variable accesses not constrained by $\prec$. The partial order thread model allows us to explore all possible alternative traces that correspond to a set of feasible interleavings in a multi-thread program, however, the model provides an over-approximation without considering data-flow, thus cannot guarantee each permissible trace in the model is covered by some feasible interleaved execution in the multi-thread program P.

2.2 Implementation of the Partial Order Thread Model

Capturing runtime traces and related source code McPatom uses Pin binary instrumentation framework [3] to collect runtime trace information, specifically including, every access to every shared variable and every synchronization event using POSIX Thread (locks, condition variables, barriers, thread joining and etc.). For each collected event, McPatom also finds the corresponding source code information including file name and line number. The source code information can be used to help locating the predicted bugs. A sample of a partial trace is shown in Fig. 2.

```
3047143104, 1, thread.c-624, Read, threads
3047143104, 1, thread.c-172, Create, 3020999536
3020999536, 1, thread.c-240, Lock, init_lock
3020999536, 1, thread.c-241, Read, init_count
3020999536, 1, thread.c-241, Write, init_count
3020999536, 1, thread.c-242, Signal, init_cond
3020999536, 1, thread.c-243, Unlock, init_lock
```

Fig. 2. A Sample of a Partial Trace (The format of each line: thread handle, timestamp, file name - line number, action)

Automatically encoding traces to Promela code McPatom uses Spin model checker to detect atomicity violations in a partial order thread model. This section shows how we realize a partial order thread model from a recorded trace in Spin's underlying language Promela.

Defining Shared Variable Accesses McPatom defines every shared variable v as a *short* in Promela, automatically assigns a unique value for all reading accesses and a unique value for all writing accesses in each thread. Formally, let $rw \in \{r, w\}$ and *tid* be thread ID, each access of v is defined as *v=rw+tid*. Since the maximum number of threads per process is limited to 64 in POSIX threads, McPatom sets r to *0*, and w to 64. For example, given two threads: *t1(tid=1)* and *t2(tid=2)*, and a shared variable v, McPatom makes the following assignments :

1. v = 64+*1* for each writing access of v in thread *t1*,
2. v = *1* for each reading access of v in thread *t1*,
3. v = 64+*2* for each writing access of v in thread $t2$,
4. v = *2* for each reading access of v in thread *t2*.

Defining Synchronization Primitives McPatom automatically generates Promela code for all synchronization primitives. Due to space limit, we only present

Promela code for mutex locks. McPatom models synchronization events to capture the causal relationships between threads, to prune infeasible interleavings. The Promela code shown in Fig. 3 models the POSIX Thread routines *pthread_mutex_lock* and *pthread_mutex_unlock*. The atomic construct groups indivisible statements together to ensure no interleaving within an atomic sequence. *Lock* inline function accepts a lock l as its argument. If lock l is not locked, *Lock* function locks it and sets the owner to the thread that is the predefined variable *_pid* for the executing process in Promela. If lock l is in locked status, no guards are executable so that the thread is blocked until lock l is available according to Promela semantics. *Unlock* inline function simply sets lock l to unlocked status. It is exactly what is required to model locking and unlocking of a mutex lock.

```
#define NUM_LOCKS 100
short locked[NUM_LOCKS] = -1;
inline Lock(l) {
 if
 ::atomic{(locked[l] == -1) -> locked[l] = _pid}
 fi;
}
inline Unlock(l) {
 assert(locked[l] == _pid);
 locked[l] = -1;
}
```

Fig. 3. Promela Code Modeling Mutex Locks

Defining Threads All events with regard to a particular thread from the recorded trace are grouped into a Promela process in which each event is represented by its corresponding Promela code defined in previous steps as shown in Fig. 4. Since the maximum number of threads per process in POSIX threads is 64, which is well below the maximum number (256) of processes allowed in Promela, we do not have problem to encode all possible threads occurring in a recorded trace. The interleaved execution of processes in the Promela program generates all alternative permissible traces in the partial order thread model.

3 Defining and Encoding Unserializable Interleaving Patterns between Two Threads

Atomicity is a semantic correctness property for concurrent programs. A thread interleaving is serializable if and only if it is equivalent to a serial execution, which executes a code region without other threads interleaved in between. The code region is typically enforced as atomic explicitly in the code. When proper

```
proctype t1() { ... }
proctype t2()
{
  Lock(init_lock);     /* thread.c - 240 */
  init_count = 0 + 2;     /* thread.c - 241 */
  init_count = 64 + 2;     /* thread.c - 241 */
  Signal(init_cond);     /* thread.c - 242 */
  Unlock(init_lock);     /* thread.c - 243 */
  ...
}
init
{
  run t2();     /* thread.c - 172 */
  ...
}
```

Fig. 4. A Sample of Partial Promela Code

synchronization is missing to enforce atomicity, atomicity violation bugs may occur. [8] proved that a thread interleaving is serializable if and only if its conflict graph is acyclic.

Most concurrency bugs involve two threads, instead of a large number of threads, based on the study in [5], in which 101 out of 105 bugs involved only two threads. Thus atomicity violation bugs in a multi-thread program can be explored through every pair of threads. Our work is inspired by the works in [2][6], which addressed a special case of unserializable interleavings with three accesses of the same shared variable. However, as Fig. 5 shows, there are real world bugs involving four accesses of the same shared variable. Furthermore, there can be more accesses involved, such as reading accesses of a shared variable for logging purpose. The patterns given in this paper cover atomicity violation bugs involving any number of accesses of a shared variable between a pair of threads.

3.1 Three-access and Four-access Atomicity Violation

Many recent works focused on three-access atomicity violations [2][6][5], which involve one shared variable, two threads and three accesses to the variable. For simplicity, two threads are referred as a local thread (Thread 1) and a remote thread (Thread 2), the opposite view is also explored during the detection process. If two consecutive accesses of a shared variable in a local thread are interleaved with an access to the variable from a remote thread, the interleaving is a potential unserializable one. In practice, unserializable interleavings indicate the presence of atomicity violation bugs. The explanation of unserializable interleavings of three accesses and many real world atomicity violation bugs can be found in [2].

Three-access atomicity violations are chosen by tools above because (1) there are many real world atomicity violation bugs involving only three accesses, and (2) checking only two accesses (current access and previous access) in a thread can reduce the complexity of algorithms. However, some atomicity violation bugs involve more than three accesses. A real world example [9] is shown in Fig. 5. The shared variable accesses in Thread 1 must be in an atomic region; otherwise, a possible interleaving may result in HandleEvent function of Thread 2 returning with a missing event. PSet [9] detected this bug (incorrect interleaving 1) since PSet keeps track of either the last writer or the set of last readers for every memory location. However PSet cannot detect the mutant of the bug (incorrect interleaving 2) because in PSet's view the mutant only involves a set of last readers and the current reading access. AVIO [2] cannot detect this bug because it involves more than three accesses.

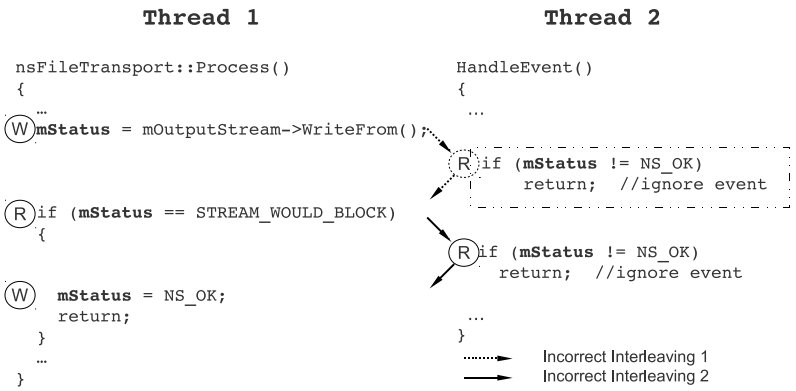

Fig. 5. A four-access atomicity violation bug [9] in Mozilla (Incorrect interleaving 1 was detected by PSet [9] and missed by AVIO [2], while incorrect interleaving 2 cannot be detected by either PSet or AVIO.)

3.2 Patterns of Two-thread Atomicity Violations involving Any Number of Accesses

In the sequel, a two-thread atomicity violation refers to a two-thread atomicity violation involving any number of accesses of a shared variable, and $A \in \{Read, Write\}$, $R = Read$, $W = Write$, A^* denotes zero or more A, A^+ denotes one or more A, R^* denotes zero or more R and R^+ denotes one or more R. This section gives a set of patterns covering all possible two-thread atomicity violations.

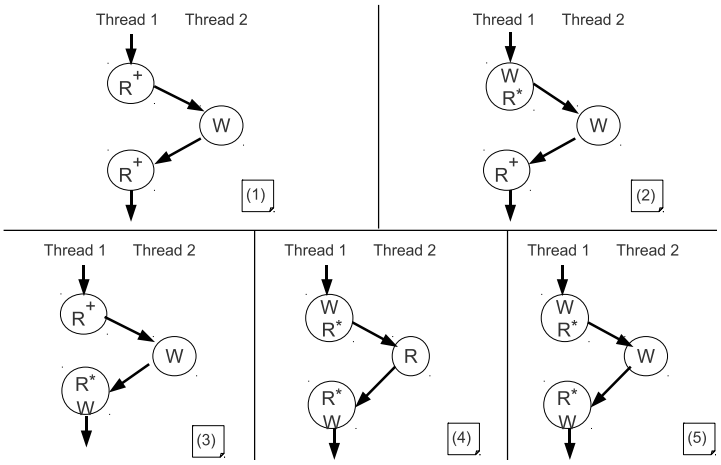

Fig. 6. Unserializable Interleavings with two threads. In (1)(2)(3)(5), W in Thread 2 unexpectedly changes the value; In (4), An intermediate value in Thread 1 is read by Thread 2.

Figure 6 shows all possible scenarios of unserializable interleavings with only one access from Thread 2. If any of the unserializable interleaving patterns is matched, it indicates a potential atomicity violation.

Theorem 1. *The set of patterns in Fig. 6 is complete, i.e. they cover all possible unserializable interleavings between two threads.*

Proof. Let $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be a sequence of atomic accesses in an interleaved execution of two threads, in which $A_i^{t_i}$ ($t_i \in \{1,2\}$, $A_i^{t_i} \in \{Read, Write\}$, $1 \leq i \leq n$) denotes an atomic access from thread t_i to the same shared variable. Let every subsequence of $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be of the form B_1^1, B_2^2, B_3^1 where B_1^1 and B_3^1 of Thread 1 are sequences of $A_i^{t_i}(t_i = 1)$, B_2^2 of Thread 2 is a sequence of $A_i^{t_i}(t_i = 2)$. Let P_i be pattern *i*. B_2^2 is assumed to be or can be reduced without losing writing operations to a single access A_2^2. If B_1^1, A_2^2, B_3^1 does not match with any of the patterns in Fig. 6, B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4 \wedge \neg P_5$. Since operator $\wedge$ is commutative, we can select a specific order and carry out an incremental analysis of possible B_1^1, A_2^2, B_3^1 based on each of $P_i(1 \leq i \leq 5)$.

1. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1$. B_1^1, A_2^2, B_3^1 can only be one of the following:
   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^+$
   (b) $B_1^1 = A^+$, $A_2^2 = W$, $B_3^1 = A^*WA^*$
   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$
2. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2$. B_1^1, A_2^2, B_3^1 can only be one of the following:
   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$
   (b) $B_1^1 = A^+$, $A_2^2 = W$, $B_3^1 = A^*WA^*$
   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$

3. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3$. B_1^1, A_2^2, B_3^1 can only be one of the following:
   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$
   (b) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B3 = A^*WA^*$ which is equivalent to above one.
   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = A^+$
4. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$. B_1^1, A_2^2, B_3^1 can only be one of the following:
   (a) $B_1^1 = A^*WA^*$, $A_2^2 = W$, $B_3^1 = A^*WA^*$
   (b) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$
   (c) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$
5. B_1^1, A_2^2, B_3^1 satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4 \wedge \neg P_5$. B_1^1, A_2^2, B_3^1 can only be one of the following:
   (a) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$
   (b) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$

According to the Serializability Theorem [8], an interleaved sequence is serializable if and only if its conflict graph is acyclic. Either 5(a) $B_1^1 = R^+$, $A_2^2 = R$, $B_3^1 = A^+$ or 5(b) $B_1^1 = A^+$, $A_2^2 = R$, $B_3^1 = R^+$ is serializable. Therefore, the completeness of the set of patterns in Fig. 6 is proved.

3.3 Automatically encoding atomicity violation patterns into Linear time Temporal Logic (LTL) Formulas

For every shared variable and every pair of threads $t1$ and $t2$, McPatom automatically defines a LTL formula (3.1) for each pattern in Fig. 6 and another LTL formula (3.2) reversing the view of t_1 and t_2. Let v be a shared variable, $r = 0$ and $w = 64$ as defined in section 2.2, $A_i \in \{r, w\}$, and tid_i , $\overline{tid_i} \in \{1, 2\}$.

$$\begin{aligned}
&[]! <> ((v == A_1 + tid_1)\&\& \\
&\quad X((v == A_2 + tid_2)U((v == A_3 + tid_3)\&\& \\
&\quad X((v == A_4 + tid_4)U(v == A_5 + tid_5))))) \qquad (3.1)\\
&[]! <> ((v == A_1 + \overline{tid_1})\&\& \\
&\quad X((v == A_2 + \overline{tid_2})U((v == A_3 + \overline{tid_3})\&\& \\
&\quad X((v == A_4 + \overline{tid_4})U(v == A_5 + \overline{tid_5}))))) \qquad (3.2)
\end{aligned}$$

where "[]" denotes *Always*, "!" denotes *Logical Negation*, "<>" denotes *Eventually*, "X" denotes *Next* and "U" denotes *Until*. These formulas specify that the atomicity violation patterns do not occur.

Using Fig. 6 (2) as a concrete example, one formula in LTL is shown below:

$$\begin{aligned}
&[]! <> ((v == w + 1)\&\& \\
&\quad X((v == r + 1)U((v == w + 2)\&\& \\
&\quad X((v == w + 2)U(v == r + 1))))) \qquad (3.3)
\end{aligned}$$

$(v == w+2)U(v == r+1)$ is true if and only if $v == w+2$ holds until $v == r+1$ is true or simply $v == r+1$ holds without $v == w+2$ holds. This subformula captures $W_2^*R_1^+$ in which W_2^* means zero or more writing accesses from Thread 2, R_1^+ means one or more reading accesses from Thread 1. Furthermore, $(v == w+2)\&\&X((v == w+2)U(v == r+1))$ captures $W_2^+R_1^+$ and $(v == r+1)U((v == w+2)\&\&X((v == w+2)U(v == r+1)))$ reflects $R_1^*W_2^+R_1^+$. Therefore, (3.3) captures $[]! <> W_1R_1^*W_2^+R_1^+$ and ensures that pattern $W_1R_1^*W_2R_1^+$ in Fig. 6 (2) does not occur in the partial order thread model. The reason that the LTL formula contains W_2^+ instead of W_2 is that there can be synchronization events between W_2 and R_1^+, for each of those events, W_2 needs to hold.

4 Predictive Analysis of Atomicity Violation using Model Checking

In this section, we discuss McPatom framework's general merits in terms of its soundness and completeness as well as specific ways in using Spin model checker [4] to show its applicability.

4.1 Soundness and completeness of McPatom

An important feature of a prediction method is its capability to predict as many violations as possible. Since the majority of existing prediction methods uses an abstract model extracted from one interleaved execution at a time from a multi-thread program, a prediction method's capability rests on the quality of the abstract model built and its thoroughness in exploring the permissible traces in the abstract model. McPatom extracts the least constrained partial order thread model respecting the causal relation from the observed interleaved execution and uses model checking to explore all permissible traces in the partial order thread model.

Theorem 2. *McPatom ensures the completeness of its prediction - any possible atomicity violation involving a pair of threads accessing one shared variable in the partial order thread model can be detected.*

Proof. McPatom encodes all possible atomicity violation patterns involving a pair of threads accessing one shared variable (Theorem 1) into linear time temporal logic formulas. McPatom uses model checking to exhaustively check whether any temporal logic formula fails in the partial order thread model. Thus none of possible atomicity violation will be undetected.

In general, McPatom cannot guarantee the soundness of its prediction, i.e., each predicted atomicity violation is covered by a feasible execution, since data-flow is ignored in the partial order thread model.

One major potential problem using model checking is the state explosion problem. Fortunately, the state explosion problem will not occur in atomicity violation prediction due to the following reasons (1) the partial order thread

model (capturing only shared variable accesses and synchronization events) used for model checking is drastically smaller compared to the original multi-thread program, (2) each atomicity violation pattern to be checked involves only one shared variable, and (3) checking each atomicity violation pattern does not depend on the value of the shared variable. Another possible problem with model checking is the potential exponential number of possible interleavings due to the number of threads involved and the number of shared variable accesses. This problem is partially resolved (1) due to our focus on checking atomicity violations involving only two threads, (2) due to the constraints imposed by causal relations that drastically reduce the number of potential interleavings generated by the number of shared variable accesses, and (3) due to our implementation strategies of grouping all reading event sequences in each thread into atomic blocks in Spin to achieve partial order reductions and enforcing the wait/signal order of condition variables in the observed execution while exploring alternative interleavings. Our experiment results show very good performance using model checking.

4.2 Using Spin model checker to find atomicity violation traces

McPatom selects Spin model checker [4] based on its maturity, popularity, and capability. Spin is used to check every atomicity violation freedom property involving every pair of threads accessing every single shared variable one at a time in the partial order thread model extracted from a single interleaved trace recorded through instrumentation using Pin. Based on the partial order thread model encoded in Promela in section 2.2, and the atomicity violation freedom property encoded in LTL formulas in section 3.3, McPatom uses Spin to find atomicity violation traces or report no atomicity violations. Figure 7 gives an example of atomicity violation reported by Spin, which is mapped to real code in the original program.

```
70: proc  2 (t13) spin_av.pml:551 (state 28) [sharedvariable
    = (0+13)]
72: proc  3 (t48) spin_av.pml:591 (state 31) [sharedvariable
    = (64+48)]
76: proc  2 (t13) spin_av.pml:552 (state 29) [sharedvariable
    = (0+13)]
```

Fig. 7. A Sample of Atomicity Violation Trace Reported by Spin

Spin can be configured to search all errors or stop at the first error. McPatom chooses to stop at the first error, thus McPatom reports no atomicity violation if there exists no atomicity violation; when McPatom reports some atomicity violation traces, there may be additional atomicity violations not yet reported, which can be detected by re-running McPatom after grouping the previously

reported violation related accesses into an atomic region so that it will not cause a new violation in the next run. For each shared variable and each pair of threads, an atomicity violation is recorded in a Spin trail file for each pattern if it exists. The Spin trail file can be simulated by Spin to give a clear view of those accesses involved in the atomicity violation, as shown in Fig. 7.

4.3 Mapping the violations reported in Spin to the original program

Atomicity violations reported in Spin, as shown in Fig. 7 as an example, are mapped to real code in original program. McPatom automatically identifies the related lines in Promela files, in which the comments of each line in Promela are file names and line numbers of the corresponding source code. Figure 8 shows the Promela code at the left and the corresponding real code at the right, for the atomicity violation in Fig. 7.

```
sharedvar=0+13; /*mod_log_config.c-1353*/  |if (len+buf->outcnt>LOG_BUFSIZE)
 sharedvar=64+48; /*mod_log_config.c-1373*/|    buf->outcnt += len;
sharedvar=0+13; /*mod_log_config.c-1369*/  |s = &buf->outbuf[buf->outcnt]
```

Fig. 8. Promela code and the corresponding real code in the original program

5 Evaluation

Table 1. Bug List

Bug #	Program	Issue Number
1	Apache	25520
2	Apache	21287
3	Apache	21285
4	MySQL	644
5	MySQL	791
6	Mozilla-extract	Fig. 5

We have used several real-world systems with known bugs listed in Table 1 (the issue numbers are the IDs in corresponding Bugzilla Databases) ([2],[9]) to examine our tool's bug prediction capability, as well as four programs [2] without atomicity violations in SPLASH-2 parallel benchmark suite [10] to test the accuracy of our tool (no false positives are reported).

Table 2. Performance

	Program	Program Input	Trace Size (MB)	Time to Check (mins)	Number of Shared Variables	Number of Properties	Average Time per Property (secs)
1	fft	-p2 -m1024	4.3	304	3656	36560	0.499
2	fmm	Particles : 64 Processors : 2	10.8	183	1248	12480	0.88
3	lu	-p2 -n16	0.3	0.44	5	50	0.53
4	radix	-p2 -n10	3.7	328	3094	30940	0.636
5	Apache	2 concurrent httperf	9.4	15.68	151	3360	0.005

Table 3. Performance (Continue)

	Program	The Shared Variable with Maximum Number of Accesses		
		Number of Accesses	Maximum Number of States	Time to Check (secs)
1	fft	1041	3294	0.04
2	fmm	20064	9996	0.08
3	lu	282	941	0.02
4	radix	81	433	0.01
5	Apache	1415	16	less than 0.01

Bug prediction capability McPatom has successfully predicted all the known bugs listed in Table 1, especially bug number 6 - an extraction of a real world atomicity violation bug reported in [9], which evades PSet [9] because this bug involves a set of last readers and the current reading access, and AVIO [2] because this bug involves more than three accesses.

Accuracy We have chosen four programs (also used in [2]) without atomicity violations in SPLASH-2 parallel benchmark suite [10] to test whether McPatom produces violation predictions, which would certainly be false positives. McPatom passed this test without reporting any violations.

Performance Since McPatom framework uses model checking as the underlying atomicity violation prediction method and relies on a third party tool, Spin, to perform the model checking, it is extremely important to demonstrate the applicability of McPatom. We conducted the experiments[1] on a PC with dual core 2.33GHz CPU and 2GB memory. Performance data are given in Table 2 and Table 3, where time to check included automatically running Spin, compiling generated pan.c and model checking properties for all shared variables. There are ten properties to check for each pair of threads accessing a shared variable based

[1] Data available at http://users.cs.fiu.edu/~rzeng001/spin12/

on five violation patterns and their mutants. Apache program contains more than two threads and results in more properties to be checked. Instrumentation overhead was similar to that given in [2]. Table 3 shows the shared variable with maximum number of accesses in each program. From Table 2 and Table 3, it shows that the number of states does not explode when the number of accesses increases since checking the shared variable with maximum number of accesses took less than 0.01 seconds (not including the time to run Spin and compile generated pan.c) while checking any shared variable on average took 0.005 seconds. These preliminary experimental results are very encouraging and demonstrate the scalability of McPatom. These results also confirm our belief that although the total number of possible interleavings to check can explode quickly as the number of accesses increase; however, the number of actual interleavings are drastically smaller due to the constraints imposed by causal relationships between threads. Other major reasons, which also vastly reduce the possible interleavings, are that McPatom takes advantage of the nature of atomicity violations and considers only a pair of threads and accesses to a single shared variable at one time, groups all reading event sequences in each thread into atomic blocks in Spin to achieve partial order reductions , and enforces the wait/signal order of condition variables in the observed execution while exploring alternative interleavings. Table 2 and Table 3 show that the experiment with Apache has even better performance than others, due to Apache's heavy use of condition variables. Since atomicity violations involving a single shared variable can be checked independently from violations involving other shared variables, we can significantly reduce the duration (not the cumulative time) of model checking by using multiple machines.

6 Related Works

There are many recent works on tackling atomicity violations. Some works proposed techniques to detect atomicity violations on actual program executions through testing [11] or runtime monitoring ([2], [12], and [13]). Other works developed methods to predict atomicity violations that may evade testing and runtime monitoring. In this section, we mention some recent works most relevant to ours on dynamically predicting atomicity violations. Most of these works share the following fundamental process: (1) instruments a multi-thread program P to record atomicity relevant events, (2) extracts a trace τ of atomicity relevant events from an interleaved execution σ of P, (3) projects trace τ into a partial order model M based on a causal relation defined on P, (4) explores various alternative trace τ' in M to predict potential atomicity violations in a possible corresponding interleaved execution σ' in P. Various methods and their supporting tools differ with regard to the strategies used in the above process.

How to abstract a partial order model M from a trace τ is critical. If the model is too restrictive, many feasible atomicity violations cannot be explored. If the model is too permissible, the prediction may not be sound, i.e. a predicted atomicity violation may not be a feasible interleaved execution of P. Penelope

[14] ignores some causal relationships in building a partial order model and thus requires additional feasibility checking of a predicted atomicity violation. Fusion [6] abstracts a partial order model called concurrent trace program (CTP) that ignores the causal relation between different threads. Linearized atomicity violation traces in CTP are symbolically checked with additional order information from source codes to ensure their feasibility. In [15], a theoretical study was conducted to analyze the complexity of predicting atomicity violations, in which two simplified partial order models are considered. The first one ignores all synchronization and the second one only considers lock-based synchronization. It shows the tradeoffs between efficiency and accuracy. jPredictor [16] defines a partial order model based on a concept of sliced causality and lock-atomicity, which may predict some infeasible violations. Our work abstracts a partial order model respecting the causal relationships imposed by all synchronization constructs, but without considering data-flow, our work also may produce some infeasible violations.

A variety of techniques have been proposed to explore atomicity violation traces from an abstract partial order model. CTrigger [5] and Penelope [14] developed different algorithms to generate potential violation schedules and to prune away many infeasible ones. However these algorithms may report infeasible atomicity violation traces as well as miss feasible ones. jPredictor [16] uses model checking to exhaustively check a property in the partial order model and is capable to predict other concurrency bugs in addition to atomicity violations. Fusion [6] encodes the partial order model, the source program, and three access atomicity violation patterns into a logic formula; and uses a satisfiability modulo theory solver to check the feasible interleavings for atomicity violations. Our work converts the partial order model into a Promela program, defines a complete set of atomicity violation patterns as temporal logic formulas, and then uses Spin model checker to produce atomicity violation traces.

7 Conclusion

Concurrency bugs are extremely hard to detect using testing techniques due to huge interleaving space. This paper presents a tool McPatom using model checking to predict atomicity violation concurrency bugs. McPatom is powerful and can explore a vast interleaving space of a multi-thread program based on a small set of instrumented test runs. McPatom is applicable to large real-world systems.

McPatom focuses on atomicity violations involving each single shared variable, and thus cannot find atomicity violations involving multiple variables. Another limitation is that redundant model checking may be performed if two recorded interleaved traces yield the same partial order thread model.

Acknowledgement

This work was partially supported by the NSF of U.S. under award HRD-0833093.

References

1. K. Poulsen, "Software bug contributed to blackout," URL: http://www.securityfocus.com/news/8016, 2004, [Online; Accessed: 07/16/2011].
2. S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, USA, 2006, pp. 37–48.
3. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *the 2005 ACM Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, IL, USA, 2005, pp. 190–200.
4. G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
5. S. Lu, S. Park, and Y. Zhou, "Finding Atomicity-Violation bugs through unserializable interleaving testing," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, p. 1, 2011.
6. C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'10)*, Paphos, Cyprus, 2010, pp. 328–342.
7. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
8. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 5.
9. J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, USA, 2009, pp. 325–336.
10. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, Madison, WI, USA, 1995, pp. 24–36.
11. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, 2008, pp. 267–280.
12. C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, AZ, USA, 2008, pp. 293–303.
13. L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, pp. 93–110, 2006.

14. F. Sorrentino, A. Farzan, and P. Madhusudan, "Penelope: weaving threads to expose atomicity violations," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, Santa Fe, NM, USA, 2010, pp. 37–46.
15. A. Farzan and P. Madhusudan, "The complexity of predicting atomicity violations," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, York, UK, 2009, pp. 155–169.
16. F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 221–230.