The Complexity of Predicting Atomicity Violations[⋆]

Azadeh Farzan[1] and P. Madhusudan[2]

[1] University of Toronto
[2] Univ. of Illinois at Urbana-Champaign

Abstract. We study the prediction of runs that violate atomicity from a single run, or from a regular or pushdown model of a concurrent program. When prediction ignores all synchronization, we show predicting from a single run (or from a regular model) is solvable in time $O(n+k.c^k)$ where n is the length of the run (or the size of the regular model), k is the number of threads, and c is a constant. This is a significant improvement from the simple $O(n^k \cdot 2^{k^2})$ algorithm that results from building a global automaton and monitoring it. We also show that, surprisingly, the problem is decidable for model-checking recursive concurrent programs without synchronizations. Our results use a novel notion of a *profile*: we extract profiles from each thread locally and compositionally combine their effects to predict atomicity violations.

For threads synchronizing using a set of locks $\mathcal{L}$, we show that prediction from runs and regular models can be done in time $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k + k^2})$. Notice that we are unable to remove the factor k from the exponent on n in this case. However, we show that a faster algorithm is *unlikely*: more precisely, we show that prediction for regular programs is unlikely to be fixed-parameter tractable in the parameters $(k, |\mathcal{L}|)$ by proving it is $W[1]$-hard. We also show, not surprisingly, that prediction of atomicity violations on recursive models communicating using locks is undecidable.

1 Introduction

The new disruptive trend in microprocessor technology that bodes a future where there will be no significant speed-up of individual processors but only a multitude of processor cores, poses a tremendous challenge to computer science. Parallel computers will become ubiquitous and all software will have to exploit parallelism to gain performance. One of the most challenging aspects of this overhauling of technology is that concurrent programs are very hard to write and debug, making reliability and programmer productivity a huge concern.

Despite various efforts in computer science that strive to enable simple models for concurrency such as *transactional memory*[24], stream-programming, actors and MPI (message passing interface) paradigms [1,14,2], that escape the dread of a wild shared-memory program, it is fairly clear that concurrent reactive

[⋆] For a full version of this paper refer to [8].

programs will exhibit significant non-determinism in terms of interleaved executions. A serious consequence of this is that software will become very hard even to *test against one particular input*: given a concurrent program and an input, there will be a myriad of interleaved executions, making testing extremely challenging. The CHESS project at Microsoft research and IBM's ConTest tool are efforts that try to address this problem.

An extremely common *generic* concurrency bug is the violation of *atomicity.* Intuitively, a programmer writing a procedure often wants non-interfered access to certain data, enabling local reasoning of the procedure in terms of how it affects the state. A programmer often puts together concurrency control mechanisms to ensure atomicity, often by taking locks on the data accessed. This is extremely error-prone: errors occur if not all locks for accessed data are taken, non-uniform ordering of locking can cause deadlocks, and naive ways of locking can inhibit concurrency, which forces programmers to invent intricate ways to achieve concurrency and correctness at the same time. Recent studies of concurrency errors [19] show that a majority of errors (69%) are atomicity violations. This motivates the problems we consider in this paper: to study algorithms that can help search the space of all interleavings for atomicity violations.

First, we assume that we have a mechanism to observe the global run of a concurrent program as an interleaved sequence of events executed by the different threads[1]. Assuming a program's global run is divided into transactions, where a transaction is a block of code like a procedure that we expect the programmer intends to be atomic, we would like to check for runs of the program that violate atomicity with respect to these transaction boundaries[2]. The notion of atomicity we study is a standard notion called *conflict-serializability*— intuitively, a conflict serializable run is a run that may involve interleaving of threads but is semantically equivalent to a serial run where all transactions are executed in a sequential non-interleaved fashion.

Given a run, the first problem of interest is to check whether it is serializable. This problem is a *monitoring* problem and we have recently solved this problem satisfactorily [7], showing that there is a deterministic monitoring algorithm that uses space at most $O(k^2 + kv)$ (for a program with k threads and v global variables). The salient aspect of this algorithm is that the space used is independent of the length of the observed run, making it extremely useful in practice.

In this paper, we study the harder problem of *predicting* atomicity violations. Given a run r, we would like to predict other runs r' which are not serializable. This is an extremely interesting and useful problem to solve; if we execute a program on an input and obtain one run r, and use it to predict non-serializable runs r' *efficiently*, it gives us a very effective mechanism of finding atomicity violations without generating and testing all interleavings in a brute-force manner.

[1] In practice, we can augment the program so that it communicates to a monitoring module, and with extra synchronization, ensure sequential consistency, and correct observation of runs.

[2] Note that we do not assume a transactional memory programming model; the programs we consider run on "wild" shared memory.

Our prediction model is simple and intuitive: given a run r, we project the run r to each of the threads to get local runs $r_1, \ldots, r_k$. We then consider *all* runs that can be obtained by combining the runs r_1 through r_k in any interleaved fashion to be predicted by the run r. Note that our notion of a run does not include conditional checks made by the threads nor the actual data written by the programs: this is intentional, as considering these aspects leads to a very complex prediction model that is unlikely to be tractable. Our prediction model is *optimistic*: we predict a larger class of runs than may be allowed by the actual program, and hence any non-serializable execution that we infer must be subject to testing to check feasibility of execution by the program.

The problem of inferring whether any interleaved execution of k local runs $r_1, \ldots, r_k$ leads to a violation of serializability is really a *model-checking* problem: for each thread T_i we are given a *straight-line* program executing r_i, and asked whether the concurrent program has a serializability violation. A natural analog of this problem is that we are given a set of k program models (finite-state transition systems or recursive transition systems) and asked whether any interleaving of them results in a serializability violation. Program models can be derived in various ways: for instance we can collect the projections of *multiple* tests and build local transition systems and check whether we can predict a run that violates atomicity. Program models may also be obtained *statically* from programs using abstraction techniques.

This paper is devoted to the theoretical analysis of predicting atomicity violations from *straight-line concurrent programs* (for predictions from tests), *regular concurrent programs* and *recursive concurrent programs.*

Let us briefly consider the problem of inferring runs from straight-line. It is clear that we can construct a global transition system that generates all the interleavings of the program, and by intersecting this with a *monitoring automaton for serializability*, predict atomicity violations. However, this essentially generates all the interleavings, which is precisely the problem we wish to avoid. The goal of this paper is to study when this can be avoided.

Notice that the state space of the global transition system generating all interleavings is $O(n^k)$ in size where n is the size of the program, and k is the number of threads. In practical applications, n is very large (the length of the run) and k, though small, is not a constant, leading to a very large state-space, making prediction almost impossible. Moreover, we clearly cannot expect algorithms to work without an exponential dependence on k (we can show that the problem is NP-complete). However, it would be extremely beneficial if we can build algorithms *where k does not occur in the exponent on n.* An algorithm that works in time $O(n + k.c^k)$ would work much faster in practice. For instance, in the SOR benchmark (see [7]) for $k = 3$ threads, the length of a run is $n = 97 \times 10^6$ nodes, and nothing short of a linear dependency on n can really work in practice.

Secondly, predicting runs gets harder when the synchronization mechanisms have to be respected. In this paper, we consider two models: one where we ignore any synchronization mechanism (which leads to faster but less accurate predictions) and one where we consider synchronization using locks.

Our main contributions in this paper are the following. Assuming the set of variables manipulated is a constant, we show:

- For prediction without considering of synchronization mechanisms, we show:
  - Straight-line programs and regular programs over a fixed set of global variables are solvable in time $O(n+k.c^k)$ for a constant c (which depends quadratically on the number of variables). This result is proved by giving a *compositional* algorithm that extracts relevant results from each thread, using a novel notion called a *profile*, and combines the profiles to check violations.
  - Prediction of atomicity errors for recursive programs is (surprisingly) decidable, and can be done in time $O(n^3 + k.c^k)$.
- For prediction in programs that use lock synchronization over a lock-set $\mathcal{L}$:
  - Straight-line programs and regular programs over a fixed set of global variables are solvable in time $O(n^k \cdot 2^{|\mathcal{L}|\cdot\log k+k^2})$. This is a global algorithm that considers all interleavings, and hence k does occur in the exponent on n. However, we show that removing the k from the exponent is highly unlikely. More precisely, we show that it is unlikely that there is an an algorithm that works in time $O(\text{poly}(n)\cdot f(k,|\mathcal{L}|))$, for *any computable function* f, by showing that the problem is W[1]-hard over the parameters $(k,|\mathcal{L}|)$. W[1]-hard problems are studied in complexity theory, and are believed not to be fixed-parameter tractable.
  - Prediction of atomicity for recursive programs is (not surprisingly) undecidable.

Two aspects of our work are novel. First, the notion of profiles that we use to give the first sound and complete compositional mechanisms to prove atomicity of programs without locks. Second, for programs with locks, our W[1]-hardness lower bound shows that an efficient compositional method is unlikely. Such fixed-parameter intractability results are not common in the verification literature (we know of no such hardness result directly addressing model checking of systems).

The paper is organized as follows. In Section 2 we first define schedules which capture how programs access variables, then define the three classes of programs we study, namely straight-line, regular, and recursive programs. We also define the notion of conflict-serializability and its algorithmic equivalent in terms of conflict-graphs. Section 3 is devoted to the study of finding atomicity violations in programs with no synchronization mechanisms while Section 4 studies the problem for programs with lock synchronization. We end with concluding remarks and future directions in Section 5.

Related Work: Atomicity is a new notion of correctness for concurrent programs. It has been suggested [10,11,26,25,27] that *atomicity violations based on serializability* are effective in finding concurrency bugs. A recent and interesting study of bug databases identifies atomicity violations to be the single major cause for errors in a class of concurrent programs [19] Work in software verification for atomicity errors are often based on the Lipton-transactional framework. Lipton transactions are *sufficient* (but not necessary) thread-local conditions that ensure serializability [18]. Flanagan and Qadeer developed a type system for atomicity [10] based on Lipton transactions (which, being local, is also compositional). Model checking has also been used to check atomicity using Lipton's transactions [11,15]. In [6], we had proposed a slightly different notion of atomicity called *causal atomicity* which can be checked using partial-order methods.

The run-time *monitoring* for atomicity violations is well-studied. Note that here the problem is to simply observe a run and check whether that particular run is atomic (involves no *prediction*). In a recent paper [9], the authors show monitoring algorithms that work with efficient space constraints to monitor atomicity violations during testing. In another recent paper [7], we have established a more sophisticated algorithm that uses *bounded* space to monitor, and results in extremely efficient monitoring algorithms. The existence of a monitor also implies that if the global state-space of a concurrent program can be modeled as a finite-state system, then the *model checking* problem for serializability is decidable.

The work in [22] defines *access interleaving invariants* that are certain patterns of access interactions on variables, learns the intended specifications using tests, and monitors runs to find errors. A variant of dynamic two-phase locking algorithm [20] for detection of an serializability violation is used in the atomicity monitoring tool developed in [27].

Turning to predictive analysis, there are two main streams of work that are relevant. In papers [26,25], Wang and Stoller study the prediction of runs that violate serializability from a single run. Under the assumptions of deadlock-freedom and nested locking, they show precise algorithms that can handle serializability violations involving *at most two transactions*. They also give heuristic incomplete algorithms for checking arbitrary runs. In contrast, the algorithms we present here do not make these assumptions, and are precise and complete. Predicting alternate executions from a single run are also studied in a series of papers by Rosu et al [23,4]. While these tools can also predict runs that can violate atomicity, their prediction model is tuned towards *explicitly* generating alternate runs, which can then be subject to atomicity analysis. In sharp contrast, the results we present here search the space of alternate interleavings efficiently, without enumerating them. However, the accuracy and feasibility of prediction in the above papers are better as the algorithm involves looking at the static structure of the programs and analyzing their control dependencies.

2 Modeling Runs of Concurrent Programs

A program consists of a set of threads that run concurrently. Each thread sequentially runs a series of *transactions*. A transaction is a sequence of actions; each action can be a read or write to a (global) variable.

We assume a finite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$. We also assume a finite set of entity names (or just entities) $\mathcal{X} = \{x_1, x_2, \ldots, x_v\}$ that the threads can access. Each thread $T \in \mathcal{T}$ can perform actions from the set $\mathcal{A}_T = \{T\text{:read}(x), T\text{:write}(x) \mid x \in \mathcal{X}\}$. Define $\mathcal{A} = \bigcup_{T \in \mathcal{T}} A_T$.

For most parts of this paper (save for the technical lemmas), we will assume that the number of variables $|\mathcal{X}| = v$ is a fixed constant.

Let us define for each thread $T \in \mathcal{T}$, the extended alphabet $\Sigma_T = A_T \cup \{T{:}\rhd, T{:}\lhd\}$. The events T:read(x) and T:write(x) correspond to thread T reading and writing to entity x, $T{:}\rhd$ and $T{:}\lhd$ correspond to boundaries that begin and end transactional blocks of code in thread T. Let $\Sigma = \bigcup_{T \in \mathcal{T}} \Sigma_T$.

For any alphabet A, $w \in A^*$, let $w[i]$ (where $i \in [0, |w| - 1]$) denote the i'th element of w, and $w[i, j]$ denote the substring from position i to position j (both inclusive) in w. For $w \in A^*$ and $B \subseteq A$, let $w|_B$ denote the word w projected to the letters in B. For a word $w \subseteq \Sigma^*$, $w|_T$ be a shorthand notation for $w|_{\Sigma_T}$, which denotes the actions that thread T partakes in.

The following defines the notion of observable behaviors on the global variables of a concurrent program, which we call a *schedule*.

Definition 1. *A transaction tr of a thread T is a word in* $(T{:}\rhd) \cdot \mathcal{A}_T^* \cdot (T{:}\lhd)$. *Let* $Tran_T$ *denote the set of all transactions of thread T, and let Tran denote the set of all transactions. A schedule is a word* $\sigma \in \Sigma^*$ *such that for each* $T \in \mathcal{T}$, $\sigma|_T$ *is a prefix of* $Tran_T^*$. *Let Sched denote the set of all schedules.*

In other words, the actions of thread T are divided into a sequence of transactions, where each transaction begins with $T{:}\rhd$, is followed by a set of reads and writes, and ends with $T{:}\lhd$. Let *Sched* denote the set of all schedules.

When we refer to two particular events $\sigma[i]$ and $\sigma[j]$ in σ, we say they *belong* to the same transaction if they belong to the same transaction block: i.e. if there is some T such that $\sigma[i], \sigma[j] \in \mathcal{A}_T$, and there is no i', $i < i' < j$ such that $\sigma[i'] = T{:}\lhd$. We will refer to the transaction blocks freely and associate (arbitrary) names to them, using notations such as tr, tr_1, tr', etc.

Concurrent Programs

We now define the three classes of programs we will work with— straight-line, regular, and recursive programs.

For a set of locks $\mathcal{L}$, and thread $T \in \mathcal{T}$, define the set of lock-actions of T as $\Pi_{\mathcal{L},T} = \{T{:}acquire(l), T{:}release(l) \mid l \in \mathcal{L}\}$. Let $\Pi_{\mathcal{L}} = \bigcup_{T \in \mathcal{T}} \Pi_{\mathcal{L},T}$.

A word $\gamma \in \Pi_{\mathcal{L}}^*$ is *lock-valid* if it respects the usual locking pattern imposed by a the locking mechanism, or formally, if for every $l \in \mathcal{L}$, $\gamma|_{\Pi_{\{l\}}}$ is a prefix of $\left[\bigcup_{T \in \mathcal{T}} (T{:}acquire(l)\ T{:}release(l))\right]^*$.

We consider three frameworks based on the structure of code in the threads.

- **A Straight-line program over** $\mathcal{L}$ is a set $Pr = \{\alpha_T\}_{T \in \mathcal{T}}$ where $\alpha_T \in (T{:}\rhd(A_T \cup \Pi_{\mathcal{L},T})^* T{:}\lhd)^*$ such that $\alpha_T|_{\Pi_{\mathcal{L},T}}$ is lock-valid.

  The runs defined by the program Pr is given by: $Runs(Pr) = \{w \mid w \in (\Sigma \cup \Pi_{\mathcal{L}})^*$, s.t. $w|_{\Pi_{\mathcal{L}}}$ is lock-valid and $w|_{\Sigma_T}$ is a prefix of α_T, for each $T \in \mathcal{T}\}$.

- **A regular program over $\mathcal{L}$** is a set $Pr = \{A_T\}_{T\in\mathcal{T}}$ where each A_T is a finite transition system. $A_T = (Q_T, q_{in}^T, \to_T)$ where Q_T is a finite set of states, $q_{in}^T \in Q_T$ is the initial state, and $\to_T \subseteq Q_T \times (\Sigma_T \cup \Pi_{\mathcal{L},T}) \times Q_T$ is the transition relation. The language of A_T, $L(A_T)$, is the set of all words $w \in (\mathcal{A}_T \cup \Pi_{\mathcal{L},T})^*$ on which there is a path from q_{in} on w. We require that for any $w \in L(A_T)$, $w|_{\Pi_{\mathcal{L},T}}$ is lock-valid, and $w|_{\Sigma_T}$ is a prefix of $Tran_T^*$.
  The runs defined by Pr is given by:
  $Runs(Pr) = \{w|\ w \in (\Sigma \cup \Pi_{\mathcal{L}})^*, \text{s.t. } w|_{\Pi_{\mathcal{L}}}$ is lock-valid and for each $T \in \mathcal{T}, w|_{\Sigma_T} \in L(A_T)\}$.
- **A Recursive program over $\mathcal{L}$** is a set $Pr = \{P_T\}_{T\in\mathcal{T}}$ where each P_T is a pushdown transition system $P_T = (Q_T, q_{in}^T, \Gamma^T, \to_T)$ where Q_T is a finite set of states, $q_{in}^T \in Q_T$ is the initial state, Γ^T is the stack alphabet, and $\to_T \subseteq Q_T \times (\Sigma_T \cup \Pi_{\mathcal{L},T}) \times \{\text{push}(d), \text{pop}(d), \text{skip}\}_{d\in\Gamma^T} \times Q_T$ is the transition relation. The language of P_T, $L(P_T)$ is the set of all words generated by P_T and is defined as usual. We again require that for any $w \in L(P_T)$, $w|_{\Pi_{\mathcal{L},T}}$ is lock-valid, and $w|_{\Sigma_T}$ is a prefix of $Tran_T^*$.
  The runs defined by Pr is given by: $Runs(Pr) = \{w|\ w \in (\Sigma \cup \Pi_{\mathcal{L}})^*$, s.t. $w|_{\Pi_{\mathcal{L}}}$ is lock-valid and for each $T \in \mathcal{T}, w|_{\Sigma_T} \in L(A_T)\}$.

Finally, for any program Pr as above, the set of schedules defined by Pr is defined as $Sched(Pr) = Runs(Pr)|_{\Sigma}$. *A program without locks is a program Pr over the empty set of locks.*

Defining atomicity
We now define atomicity as the notion of *conflict serializability.* Define the *dependency* relation D as a symmetric relation defined over the events in Σ, which captures the dependency between (a) two events accessing the same entity, where one of them is a write, and (b) any two events of the same thread, i.e.,

$$\begin{aligned} D = \{(T_1{:}a_1, T_2{:}a_2) \mid\ & T_1 = T_2 \textit{ and } a_1, a_2 \in A \cup \{\rhd, \lhd\} \textit{ or} \\ & \exists x \in \mathcal{X} \textit{ such that } (a_1 = \text{read}(x) \textit{ and } a_2 = \text{write}(x)) \textit{ or} \\ & (a_1 = \text{write}(x) \textit{ and } a_2 = \text{read}(x)) \textit{ or } (a_1 = \text{write}(x) \textit{ and } a_2 = \text{write}(x))\}. \end{aligned}$$

Definition 2 (Equivalence of schedules). *The equivalence of schedules is defined as the* smallest *equivalence relation $\sim\ \subseteq Sched \times Sched$ such that: if $\sigma = \rho a b \rho', \sigma' = \rho b a \rho' \in Sched$ with $(a, b) \notin D$, then $\sigma \sim \sigma'$.*

It is easy to see that the above notion is well-defined. Two schedules are considered equivalent if we can derive one schedule from the other by iteratively swapping consecutive independent actions in the schedule.

We call a schedule σ *serial* if all the transactions in it occur sequentially: formally, for every i, if $\sigma[i] = T{:}a$ where $T \in \mathcal{T}$ and $a \in A$, then there is some $j < i$ such that $T[i] = T{:}\rhd$ and every $j < j' < i$ is such that $\sigma[j'] \in A_T$. In other words, the schedule is made up of a sequence of complete transactions from different threads, interleaved at boundaries only.

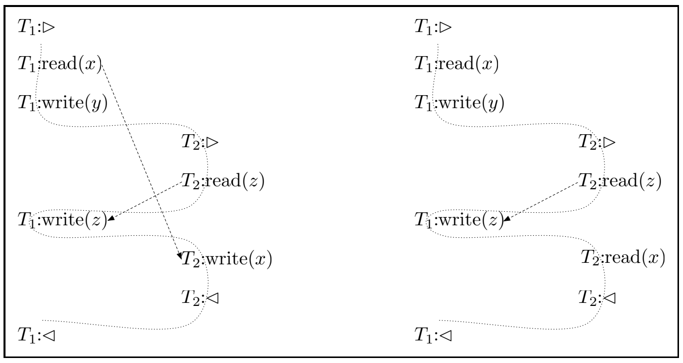


Fig. 1. A non-serializable schedule and a serializable schedule

Definition 3. *A schedule is* serializable *if it has an equivalent serial schedule. That is, σ is a serializable schedule if there a* serial *schedule σ' such that $\sigma \sim \sigma'$.*

Example 1. Figure 1 contains two schedules depicted by the dotted lines. The one on the left is not serializable. The dependent events (T_1:read(x), T_2:write(x)) indicate that T_2 has to be executed after T_1 in a serial run, while the pair of dependent events (T_2:read(z), T_1:write(z)) impose the opposite order. Therefore, no equivalent serial run can exist. The schedule on the right is serializable since in an equivalent serial run exists that runs T_2 followed by T_1.

The Conflict-Graph Characterization: For any schedule σ, let us give names to transactions in σ, say $tr_1, \ldots, tr_n$. The *conflict-graph* of σ is $CG(\sigma) = (V, E)$ where $V = \{tr_1, \ldots, tr_n\}$ and E contains an edge from tr to tr' iff there is some event a in transaction tr and some action a' in transaction tr' such that (1) the a-event occurs before a' in σ, and (2) aDa'.

Lemma 1. *[3,20,12,7] A schedule σ is atomic iff the conflict graph associated with σ is acyclic.*

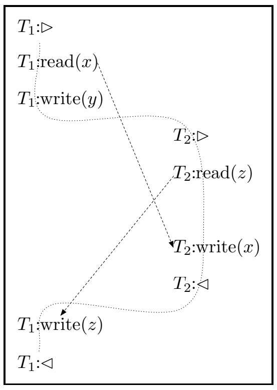
The above characterization yields a simple algorithm for serializability:

Proposition 1. *The problem of checking whether a single schedule σ is serializable is decidable in polynomial time.*

3 Model Checking Atomicity for Concurrent Programs without Synchronizations

In this section, we present model checking algorithms for checking atomicity of finite-state concurrent programs (straight-line, regular, and recursive programs). Let us first show that if the program has a non-serializable run, then it has a non-serializable run of a particular form.

A run σ is said to be *normal* if there is a thread T_i such that $\sigma = u_i \cdot T_i{:}\triangleright \cdot v_i \cdot w_1 \cdot w_2 \cdots w_{i-1} \cdot w_{i+1} \cdots w_k \cdot v_i' \cdot T_i{:}\triangleleft \cdot u_i'$, where $w_j = \sigma|_{\Sigma_{T_j}}$ (for every j), $u_i \cdot T_i{:}\triangleright \cdot v_i \cdot v_i' \cdot T_i{:}\triangleleft \cdot u_i' = \sigma|_{\Sigma_{T_i}}$, and $v_i \cdot v_i' \in A^*_{T_i}$. In other words, a run is normal if it executes a thread from the beginning up to the middle of a transaction in that thread, executes other threads serially and completely, and then finishes the incomplete thread. The Figure on the right demonstrates the normal run which is equivalent to the non-serializable run in Figure 1 (on the left).



The following observation will prove useful throughout this section:

Lemma 2. *If a program with no locks ($\mathcal{L} = \emptyset$) has a non-serializable run, then it has a non-serializable normal run.*

The crucial observation (behind Lemma 2) is that there are really at most two events in each thread that contribute to evidencing the cycle in the conflict graph, and hence witnessing non-serializability. Intuitively, for each thread T in the cycle, we pick can pick two events in_T and out_T, that cause respectively the incoming edge from the previous thread and the outgoing edge to the next thread in the cycle. This observation leads us to the following notion of profiles:

Definition 4 (Profile). *Let $\sigma_T \subseteq \Sigma_T^*$ be a local schedule. A profile for σ_T is a (bounded-length) word π that is of one of the following forms:*

- *$\pi = T{:}\triangleright\ T{:}a\ T{:}\triangleleft$, where $T{:}a$ occurs in σ_T, or*
- *$\pi = T{:}\triangleright\ T{:}a\ T{:}b\ T{:}\triangleleft$, provided there are two indices i and j such that $i < j$, $\sigma_T[i] = T{:}a$, $\sigma_T[j] = T{:}b$, and moreover there is no i' with $i < i' < j$ and $\sigma_T[i'] = T{:}\triangleleft$. In other words, $T{:}a$ and $T{:}b$ occur as events in σ_t in that order, and belong to the same transaction.*
- *$\pi = T{:}\triangleright\ T{:}a\ T{:}\triangleleft\ T{:}\triangleright\ T{:}b\ T{:}\triangleleft$, provided there are two indices i and j such that $i < j$, $\sigma_T[i] = T{:}a$, $\sigma_T[j] = T{:}b$, and moreover there is an i' with $i < i' < j$ and $\sigma_T[i'] = T{:}\triangleleft$. In other words, $T{:}a$ and $T{:}b$ occur as events in σ_t in that order, and belong to different transactions.*

The idea of a profile is that it picks one or two events from a thread's execution, along with the information as to whether the two events occurred in the same transaction or in different transactions. It turns out that profiles are enough to witness non-serializability.

Lemma 3. *A program P with no locks (straight-line, regular, or recursive) has a non-serializable run if and only if there exists a set $\langle \pi_T \rangle_{T \in \mathcal{T}}$, where each π_T is a profile of $\sigma|_T$, such that the straight line program defined by these profiles has a non-serializable run.*

The above lemma is very important, as it says that no matter how long or complex a thread is, we can summarize it using short profiles and check the profiles for non-serializability. This will form the key technical idea in proving the upper bounds in this section.

3.1 Straight-Line and Regular Programs

We discuss now the problem of checking whether a straight-line or regular program has a non-serializable schedule. We show that, by using profiles, we can solve this problem in $O(n + k.c^k)$ time where n is the maximum size of the program for any thread, k is the number of threads, and c is a constant.

Suppose that a regular program Pr consists of threads $T_1, \ldots, T_k$. The idea is to replace each thread T_i by a set of profiles $\mathbf{P}_i$, and then check whether the collection of profiles $\mathbf{P}_1, \ldots, \mathbf{P}_k$ induces a non-serializable run. By Lemma 3, Pr has a non-serializable run if and only if the collection of profiles $\mathbf{P}_1, \ldots, \mathbf{P}_k$ induces a non-serializable run.

For all threads T_i, the set of profiles $\mathbf{P}_i$ can be computed from T_i in $O(n)$ time. Assuming that T_i is represented by a finite transition system of size n, one can establish in time linear in n whether a profile π is a profile of T_i. Since there are at most v^2 possible profiles (where v is the number of global variables), one can compute all profiles of T_i in time $O(v^2 n)$. In fact, we construct an automaton P_i which accepts $\mathbf{P}_i$, the set of all profiles of T_i. These profile automata are all of size $O(v^2)$. We build a product automaton $\mathcal{P}$ which accepts the set of all possible interleavings of strings accepted by $P_1, \ldots, P_k$. Hence, $\mathcal{P}$ accepts the set of all possible interleavings of profiles of threads $T_1, \ldots, T_k$, and its size is $O\left((v^2)^k\right)$.

We can now intersect $\mathcal{P}$ with a monitoring automaton $\mathcal{S}$ for non-serializability (see [7]). The monitor maintains a graph with k nodes, and a set of labels of size $O(v)$ for each node; therefore $\mathcal{S}$ is of size $O\left(2^{k^2} + k2^{vk}\right)$. However, since it suffices to monitor only *normal* runs, we can restrict ourselves to graphs with only a linear number of edges and vertices, resulting in an automaton of size $O\left(k2^{vk}\right)$. Thus, the product automaton is of size $O\left(k.2^{vk}.v^{2k}\right)$. Hence we have the following result:

Theorem 1. *Given a straight-line or regular program Pr, one can check in time $O(nv^2 + k2^{vk}.v^{2k})$ whether Pr has a non-serializable run, where n is the maximum size of a thread, k is the number of threads, and v is the number of variables. When v is a constant, the complexity reduces to $O(n + kc^k)$ where c is a constant.*

We can show that, in general, an exponential dependence on the input is unlikely to be avoidable:

Theorem 2. *The problem of checking non-serializability of straight-line programs and regular programs, without locks, are both* NP*-complete.*

3.2 Recursive Programs

In this section, we discuss the effect of the presence of recursion in the code on the serializability checking problem. Note that even reachability of a global state is *undecidable* for concurrent recursive programs, and, since serializability is a fairly complex global property, even the decidability of serializability is not obvious.

We show, surprisingly, that checking serializability for recursive programs without locks is indeed *decidable* and in time $O(n^3 + k.c^k)$. Again, the notion of profiles come to the rescue, as they avoid searching the global state-space.

By Lemma 4, the witness for non-serializability need only contain a profile of each thread; Therefore, we can, similar to the regular program case, extract the profiles of each thread, and combine the profiles (which are straight-line programs) to check for non-serializability.

Extracting profiles from non-recursive threads is a rather straightforward task. For recursive programs, this is slightly more involved. Recall that each thread T is modeled as a pushdown automaton (PDA) P_T. We show that for any PDA P , we can *efficiently* construct an NFA (nondeterministic finite automaton) N, such that the set of profiles of P and N are the same. Therefore, we can replace the PDA model (the recursive code) of a thread by regular program, effectively removing recursion, and reduce serializability of recursive programs to that of regular programs.

Lemma 4. *For a PDA P, we can construct, in $O(|P|^3.v^2)$-time, an NFA that is of size $O(|P|.v^2)$ and that accepts the set of all profiles of schedules of P.*

The result below follows from our result on checking serializability of regular programs.

Theorem 3. *Given a recursive program Pr, the problem of checking whether it generates a non-serializable schedule, is solvable in time $O(n^3v^2 + k.2^{vk}.v^{2k})$, where n is size of the program, k is the number of threads, and v is the number of variables. When v is a constant, the complexity reduces to $O(n^3 + kc^k)$, where c is a constant.*

4 Programs with Lock Synchronization

In this section, we consider programs that synchronize using locks. We establish two simple results: first, we show that the problem of checking straight-line and regular programs with locks is solvable in time $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k})$, and, second, that the problem of checking recursive programs with locks is undecidable. Note that the complexity bounds we prove for straight-line programs and regular programs are *not* of the form $O(\text{poly}(n) \cdot 2^{|\mathcal{L}| \cdot \log k} \cdot f(k))$, i.e., we do not remove k from the exponent on n, as we did for checking atomicity of programs without locks by extracting profiles locally and combining them. However, for programs with locks, a notion of summarizing a thread using a finite amount of information that is independent of n seems hard. In fact, we believe that *no such scheme* exists. More precisely, we show that the problem of checking atomicity in regular programs with locks is unlikely to be *fixed-parameter tractable* (i.e., it is unlikely that there is an algorithm that works in time $O(\text{poly}(n) \cdot f(k, |\mathcal{L}|))$ for *any computable function f*) by showing that the problem is W[1]-hard.

Given a straight-line or regular program with locks, we can construct the product machine that generates all global runs. This machine will be of size $O(n^k \cdot 2^{|\mathcal{L}| \cdot \log k})$, as its state-space will track individual states of each thread, and in addition will keep track for each lock, the thread that holds it. We can now intersect this with a monitoring automaton for non-serializability (see [7]), which is of size $O(2^{k^2+kv})$. It is easy to see that the language of the resulting automaton is empty if and only if the program has a serializability violation. We therefore have proven the following theorem.

Theorem 4. *The problem of checking whether a straight-line program or a regular program with locks has a serializability violation is decidable in time $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k + k^2 + kv})$. When v is a constant, the complexity reduces to $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k + k^2})$.*

Let us now consider recursive programs with locks. It is known that the global reachability problem for two recursive machines communicating via synchronous messages is *undecidable* [21]. Moreover, it is known (see Kahlon et al [16]) that synchronous messages can be simulated using locks, and hence the global reachability problem for two recursive machines synchronizing using locks is undecidable. It is not hard to reduce this problem to checking serializability of a recursive program: intuitively, we augment the machines to execute a non-serializable run when they reach their respective goal states. Hence:

Theorem 5. *The problem of checking whether a recursive program with locks has serializability violations is undecidable.*

4.1 A Lower Bound on Checking Atomicity of Lock Synchronized Regular Programs

In this section, we will assume that the number of variables, v, is a constant.

In the setting of programs where all synchronization was ignored, we showed that predicting atomicity errors can be done in time $O(\mathrm{poly}(n) \cdot k.c^k)$. As we argued, this is a much better algorithm than the naive algorithms that work in time $O(n^k)$ as typically n is much larger than k. In the setting of programs that synchronize using locks, we showed only an algorithm that runs in time $O(n^k \cdot 2^{|\mathcal{L}|\cdot \log k})$. A natural question is to ask whether this problem can also be solved in time $O(\mathrm{poly}(n) \cdot 2^{|\mathcal{L}|\cdot \log k} \cdot f(k))$. We now show that this is unlikely: in fact, we show that the problem is unlikely to be *fixed-parameter tractable* (over the parameter k) by showing it is $W[1]$-hard.

Consider a problem X in which to each instance i we associate in addition to its size n a second a *parameter* $k \in \mathbb{N}$. Then the problem X is said to be *fixed-parameter tractable* with respect to k if there is an algorithm that decides X in time $O(n^c \cdot f(k))$, where f is an *arbitrary* function (we will assume f is computable) and c is a constant.

Fixed-parameter tractability is a mature area of computational complexity theory; we refer the reader to the textbooks [5,13]. For instance, finding a vertex cover of a graph G with k sets is an NP-complete problem, but is fixed-parameter tractable when the parameter is k (in fact, solvable in time $O(2^k \cdot |G|)$). Also, there is a hierarchy of classes of problems, called the W-hierarchy, for which no fixed-parameter tractable algorithms are known, and it is believed that problems complete for these classes are not fixed-parameter tractable. For instance, finding an independent set of size k in a graph G, where k is the parameter, is known to be $W[1]$-hard and hence not believed to be fixed-parameter tractable.

In this section, we will show that the problem of checking whether a regular program with locks has an atomicity violation, where the parameters are the number of threads in the program and the number of locks, is W[1]-hard.

We show hardness by reducing the problem of finite state automata intersection given below, which is known to be W[1]-hard, to our problem:

Finite State Automata Intersection
Instance: A set of k deterministic finite-state automata $A_1, \dots A_k$ over a common alphabet Σ (Σ is *not* fixed).
Parameters: k, m
Question: Is there a string $w \in L(A_1) \cap L(A_2) \cap \dots L(A_k)$ with $|w| \geq m$?

Given an instance of this problem $\langle A_1, \dots, A_k \rangle$, we construct finite-state automata $B_1, \dots B_k$ over a set of locks $\mathcal{L}$ and variables V such that they have a serializability violation if and only if the intersection of $A_1, \dots, A_k$ is nonempty. Furthermore, and most importantly, $|\mathcal{L}| = O(k \cdot |\Sigma|)$, $V = \{x\}$, a single variable, and each B_i will be of size $O(|A_i| \cdot m)$. Note that the parameters never occur in the exponent in the complexity of any of these sizes. Hence, an FPT algorithm for serializability of regular programs with locks will imply that the finite-state intersection problem is fixed-parameter tractable, which is unlikely as it is $W[1]$-hard.

The construction proceeds in two phases. First, we construct automata $C_1, \dots, C_k$ that communicate using pairwise rendezvous, and show that they exhibit a serializability violation if and only if the intersection of $A_1, \dots, A_k$ is nonempty. Then we show that the pairwise rendezvous mechanism can be simulated using locks. Intuitively, the automaton C_1 guesses a letter and communicates it to all other processes by relay messaging. All automata update their state, each C_i simulating automaton A_i. C_1 ensures that at least m letters have been guesses, and then sends a message asking whether all other processes have reached their final states. If they all respond that they have, C_1 and C_2 perform a sequence of accesses to a single variable x that results in a serializability violation. Finally, we show that we can simulate the pairwise rendezvous of communication using only lock-synchronization (using a mechanism in Kahlon et al [16], and build automata $B_1, \dots, B_k$ such that they exhibit a serializability violation if and only if the intersection of the languages of $A_1, \dots, A_k$ has a string longer than m. This leads us to the following theorem:

Theorem 6. *The following problem:*

Serializability of Regular Programs
Instance: *A regular program $B_1, \dots B_k$ with lock synchronization over a set of locks L and over a single global variable x.*
Parameter: $k, |L|$
Question: *Is the program atomic?*

is $W[1]$-hard. □

The above shows that it is unlikely that there is an algorithm that can solve atomicity of regular programs in time $O\left(\text{poly}(n) \cdot f(k, |\mathcal{L}|)\right)$. The question as to whether the problem of checking serializability violations of *straight-line* programs is also $W[1]$-hard is open.

The above reduction from automata intersection to atomicity has the property that the state-space of the machines and the lock-set are only linear in k; this has further implications. In [17], it was shown that the intersection of k finite-state automata, each of size n, is unlikely to be solvable in time $O\left(n^{(k/f(k))+d}\right)$ where $f = o(k)$ and $d > 0$ is a constant (i.e. reducing the exponent from k to a function sublinear in k). The authors show that if this were true, then problems solvable in nondeterministic time t would be solvable in subexponential deterministic time. This unlikelihood combined with our reduction (simplified not to count the number of letters in the word) implies that it is unlikely to find algorithms for atomicity that work in time $O\left(n^{(k/f(k))+d}\right)$ as well. That is, not only is k unavoidable in the exponent on n, a sub-linear exponent is also unlikely.

5 Conclusion and Future Work

We have established fundamental algorithms for predicting atomicity violations from straight-line programs, regular programs, and recursive programs. We have studied two prediction models: one which ignores any synchronization of the threads, and the other that considers lock-based synchronization. Our main results are that the problem is tractable, and solvable without exploring all interleavings, for the case when synchronizations are ignored. We believe that the notion of profiles set forth in this paper, which compositionally solve the serializability model-checking problem, will be very useful in practical tools. For synchronization using locks, we showed that such an efficient compositional scheme is unlikely, by proving a W[1]-hardness lower bound for regular programs.

There are several future directions worthy of pursuit. First, we are implementing prediction tools for atomicity violations in large programs, and preliminary results show that more restrictions (such as limiting violations to involve only two threads) are needed to make algorithms practical. Second, we do not know whether prediction of atomicity violations of straight-line programs with locks is also W[1]-hard; establishing this will give a strong argument to use prediction models that ignore synchronizations. Finally, the recent study of *nested locking* holds promise, as global reachability of concurrent programs synchronizing via nested locks admits a compositional algorithm [16]. We would like to investigate whether atomicity prediction can also benefit if threads use nested locking.

References

1. MPI: A message-passing interface standard, `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`
2. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
3. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. 13(2), 185–221 (1981)
4. Chen, F., Serbanuta, T.F., Rosu, G.: jpredictor: a predictive runtime analysis tool for java. In: ICSE, pp. 221–230 (2008)

5. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1998)
6. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
7. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
8. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. Technical Report CSRG-591, University of Torotno, Department of Computer Science (2009)
9. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI, pp. 293–303 (2008)
10. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349 (2003)
11. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL, pp. 256–267 (2004)
12. Fle, M.P., Roucairol, G.: On serializability of iterated transactions. In: PODC 1982: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pp. 194–200. ACM Press, New York (1982)
13. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
14. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS, pp. 151–162 (2006)
15. Hatcliff, J., Robby, Dwyer, M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004)
16. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
17. Karakostas, G., Lipton, R.J., Viglas, A.: On the complexity of intersecting finite state automata and n l versus n p. Theor. Comput. Sci. 302(1-3), 257–274 (2003)
18. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
19. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339 (2008)
20. Papadimitriou, C.: The theory of database concurrency control. Computer Science Press, Inc, New York (1986)
21. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
22. Lu, S., Tucek, J., Qin, F., Zhou, Y.: Avio: detecting atomicity violations via access interleaving invariants. In: ASPLOS, pp. 37–48 (2006)
23. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. STTT 8(3), 248–260 (2006)
24. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
25. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP, pp. 137–146 (2006)
26. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multi-threaded programs. IEEE Transactions on Software Engineering 32, 93–110 (2006)
27. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. 40(6), 1–14 (2005)