Analysis of Synchronization Errors for Multithreaded Programs

A Dissertation Presented

by

Liqiang Wang

to

The Graduate School

in Partial fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2006

Abstract of the Dissertation

Analysis of Synchronization Errors for Multithreaded Programs

by
Liqiang Wang

Doctor of Philosophy
in
Computer Science
Stony Brook University
2006

This dissertation describes runtime, static, and hybrid analyses to detect synchronization errors in multi-threaded programs. Three kinds of synchronization errors are considered: deadlocks, data races, and atomicity violations. Deadlocks and data races are well-known and have been studied for a long time. Atomicity violation is not as well-known and deeply studied. This dissertation focuses on detecting atomicity violations.

Atomicity is a correctness condition for concurrent systems. Informally, atomicity is the property that every concurrent execution of a set of transactions is equivalent to some serial execution of the same transactions. In multi-threaded programs, executions of procedures (or methods) can be regarded as transactions. Correctness in the presence of concurrency typically requires atomicity of these transactions. Tools that automatically detect atomicity violations can uncover subtle errors that are hard to find with traditional debugging and testing techniques. Furthermore, an atomic code block can be treated as a single transition during subsequent analysis of the program; this can greatly improve the efficiency of the subsequent analysis.

This dissertation describes three algorithms for runtime detection of atomicity violations and compares their cost and effectiveness. The reduction-based algorithm checks atomicity based on commutativity properties of events in a trace. The block-based algorithm efficiently represents the relevant information about a trace as a set of blocks (*i.e.*,

pairs of events plus associated synchronization), and checks atomicity by comparing each block with other blocks. The commit-node algorithm organizes the events in each transaction into a tree, then detects atomicity violations by analyzing relationships between nodes in different trees.

We evaluated the algorithms on several benchmarks totaling 36 KLOC. Many synchronization errors were revealed, including some previously unknown errors in Sun's implementation of the Java standard library. The block-based algorithm is most accurate; the reduction-based algorithm is the least accurate; the commit-node algorithm is less accurate than the block-based algorithm in theory, but they have the same accuracy in practice. In practice, the commit-node algorithm is as fast as the reduction-based algorithm, and significantly faster than the block-based algorithm.

This dissertation also presents an automated static analysis of atomicity for programs that use non-blocking synchronization. The analysis determines commutativity of operations based primarily on how synchronization primitives (including locks, load-linked, store-conditional, and compare-and-swap) are used. A reduction theorem states that certain patterns of commutativity imply atomicity. We successfully applied the analysis to several well-known non-blocking algorithms that cannot be automatically analyzed using previous approaches.

We also developed more accurate and efficient runtime algorithms for detecting deadlocks and data races in multi-threaded programs.

We explored the use of static analysis to significantly decrease the overhead of runtime checking for synchronization errors. The analysis results of the type systems are used to identify parts of the program from which runtime checking can safely be omitted.

To
My wife, Li Liu,
My parents, and
My son, Daniel Wang

Contents

List of Tables

List of Figures

Acknowledgments

My most earnest acknowledgment must go to my advisor, Prof. Scott D. Stoller, for his extraordinary guidance, caring, and patience. This dissertation would not be completed without his support and tireless help. As an excellent supervisor and researcher, he will be a great example throughout my professional life.

I would like to express my deepest gratitude to the dissertation committee members, Dr. Christoph von Praun, Prof. C. R. Ramakrishnan, and Prof. Radu Grosu, for their invaluable suggestions to improve this dissertation.

I would like to thank all my friends and colleagues in the Computer Science Department at Stony Brook University. This exceptional group of people created a collaborative, friendly, well-organized, and stimulating environment.

My wife, Li Liu, has consistently supported me, and shared all happiness and difficulties with me. My son, Daniel, brings tremendous joy to us. I would also like to thank my parents and my parents-in-law. They were always encouraging me with their best wishes.

Chapter 1

Introduction

Multi-threading is an increasingly common programming technique. At the hardware level, multi-core processors (also called chip-level multi-threading) are increasingly used to improve performance. At the system level, most operating systems and middlewares are multi-threaded and support multi-threaded applications for efficient and convenient sharing of resources. At the application level, more and more programs (*e.g.*, web servers and browsers) are multi-threaded for the same reasons. However, developing multi-threaded programs is difficult. Concurrency introduces the possibility of errors that do not exist in sequential programs. Furthermore, multi-threaded programs may behave differently from one run to another, because threads are scheduled indeterminately. For most systems, the number of possible schedules is enormous, and testing the system's behavior for each possible schedule is infeasible. Specialized techniques are needed to ensure that multi-threaded programs do not have concurrency-related errors.

1.1 Synchronization Errors

In concurrent or parallel programming, *synchronization* means the coordination of simultaneous threads (or processes) in order to get correct runtime order of events and avoid unexpected concurrency-related errors.

This dissertation mainly considers Java synchronization mechanism [22]. Java synchronization is implemented using monitors. Each object in Java is associated with a monitor. A synchronized method or synchronized block automatically performs a lock action before its body executes and an unlock action after its body has completed. Java offers two procedures `wait` and `notify` that may be called in synchronized methods or blocks. Since this

dissertation focuses on analyzing atomicity violations, `wait` and `notify` are not considered, because we believe that `wait` and `notify` are rarely used to achieve atomicity. This dissertation assumes sequential consistency as the memory model.

We consider three common synchronization errors: deadlocks, data races and atomicity violations. Deadlocks and data races are well-known and have been studied for a long time. Numerous static and runtime (dynamic) analysis techniques are designed to ensure that concurrent programs are free of deadlocks and data races [14, 7, 6, 44, 9]. Atomicity violation is not as well-known and deeply studied. This dissertation focuses on detecting atomicity violations, but also considers the other two.

1.1.1 Deadlock

A *deadlock* occurs when all threads are blocked, each waiting for some action by one of the other threads. Dependences among threads and resources can be modeled by a resource allocation graph, where nodes denote threads and exclusive resources, and edges denote allocation or wait-for relations between threads and resources. A common approach to detect deadlock is to check whether the resource allocation graph contains a cycle [46].

A program has *potential for deadlock* if some executions of the program ends in deadlock, but deadlock does not necessarily occur in every execution. We describe and evaluate a dynamic technique and a static technique to detect potential for deadlocks in Chapter 9.

1.1.2 Data Race

Two accesses to shared variables *conflict* if they access the same variable and at least one of them is a write. Following [44], a *data race* occurs when multiple concurrent threads perform conflicting accesses and the threads use no explicit mechanism to prevent the accesses from being simultaneous. Similar to potential for deadlock, a program has *potential for data race* if data race occurs in some executions of the program.

There are two common approaches to detect potential for data races. One approach is lockset-based detection, where a potential data race is reported when two or more threads access the same shared variable without holding a common lock [44]. The other approach is based on happen-before analysis, where a potential for data race is deemed to have occurred when two or more threads access the same shared variable, and the accesses are causally unordered [45]. We describe a new lock-based algorithm that also considers some happen-before relations in Section 4.5.

1.1.3 Violation of Atomicity

Even if a program does not have any potential for deadlock or data races, it may still contain synchronization errors. Consider the implementation of `Vector` in Sun JDK 1.4.2,

```
public class Vector extends ... implements ... {
  public Vector(Collection c) {
  // c is v1, elementCount is the field of v2.
 1   elementCount = c.size();
 2   elementData = new Object[(int)Math.min((elementCount*110L)/100
                              Integer.MAX_VALUE)];
 3   c.toArray(elementData);
  }
  public synchronized int size() { return elementCount; }
  public synchronized Object[] toArray(Object a[]) {
    if (a.length < elementCount)
      /* i.e. v2.length < v1.elementCount,
         this branch will be taken if v1.add is executed. */
      a=(Object[])java.lang.reflect.Array.newInstance(
          a.getClass().getComponentType(), elementCount);
    System.arraycopy(elementData,0,a,0,elementCount);
    if (a.length > elementCount)
      a[elementCount] = null;
    return a;
  }
  public synchronized void removeAllElements() {...}
  public synchronized boolean add(Object o) {...}
}

 thread_1  Vector v2 = new Vector(v1);
 thread_2  v1.removeAllElements(); or v1.add(o);
```

Figure 1.1: An example showing that the constructor of `java.util.Vector` in Sun JDK 1.4.2 violates atomicity.

part of which appears in Figure 1.1. Consider the following execution of the program at the bottom of Figure 1.1: `thread_1` constructs a new vector `v2` from another vector `v1` with k elements by calling the constructor for `Vector`. But before the constructor completes, `thread_1` yields execution to `thread_2` immediately after statement 1 in the `Vector` constructor. `thread_2` removes all elements of `v1`, and then `thread_1` resumes execution at statement 2. The incorrect outcome is that `v2` has k elements, all of which are `null` because the `elementData` array of `v2` is allocated according to the previous size of `v1`. A more subtle error occurs if `thread_2` executes `v1.add(o)` instead of `v1.removeAllElements()`. Then, if $k < 10$, the length of `elementData` is smaller than the new size of `v1`. Although a larger array is allocated in `toArray` to store the elements of `v1`, the array is not returned to the constructor of `v2`, thus `v2` will incorrectly be full of `null` elements. No exception is thrown in these scenarios. Methods `size(), toArray(Object[]), removeAllElements()` and `add(Object)` are synchronized, hence there is no data race in these examples.

The incorrect behavior reflects a higher-level synchronization error, namely, lack of atomicity. Atomicity is well known in the context of transaction processing, where it is

sometimes called *serializability*. The methods of concurrent programs are often intended to be atomic. A set of methods is *atomic* if concurrent invocations of the methods are always equivalent to performing the invocations serially (*i.e.*, without interleaving) in some order. The first scenario of the example in Figure 1.1 contains two invocations, one of `Vector(Collection)` and one of `removeAllElements()`, which obviously do not have an equivalent serial execution. Therefore, these methods violate atomicity. Similarly, the second scenario also shows a violation of atomicity.

We designed, implemented, and evaluated several runtime, static, and hybrid approaches for detecting atomicity violations. They are described in Chapters 4, 5 6, 7, 8, and 9.

1.2 Contributions

The main contributions of this dissertation are thorough analyses for detecting potential atomicity violations, data races and deadlocks. This dissertation focuses on analysis of Java programs, but the techniques can be applied to other languages, too.

- *Conflict-atomicity and view-atomicity.*

  We define two notions of atomicity. They correspond to the classic database notions of conflict-serializability and view-serializability. The idea is to consider all permutations of the observed execution that are consistent with the synchronization events, and to check whether every such permutation is conflict-equivalent or view-equivalent, respectively, to a serial execution.

  We explore the theory of this problem by considering its time complexity. It is well-known that checking conflict-serializability and view-serializability of a history for database transactions are in polynomial time [5] and NP-complete [38] problems, respectively. Surprisingly, we show that the situation is different for atomicity: the problems of checking conflict-atomicity and view-atomicity for the transactions in an execution of a program are polynomially reducible to each other. The proof is presented in Appendix A. We speculate that both problems are NP-complete; proving this is an open problem.

- *Runtime algorithms for detecting potential atomicity violations.*

  Runtime analysis reasons over program executions. It is less powerful than static analysis which reasons over programs without actually executing them, because the runtime analysis cannot ensure correctness of all unexplored behaviors of the system, but may be more precise (*i.e.*, give fewer false alarms) for the explored behaviors. Furthermore, runtime analysis does not require annotations of the code that are often required by static analysis (*e.g.*, type systems); this is a significant practical advantage.

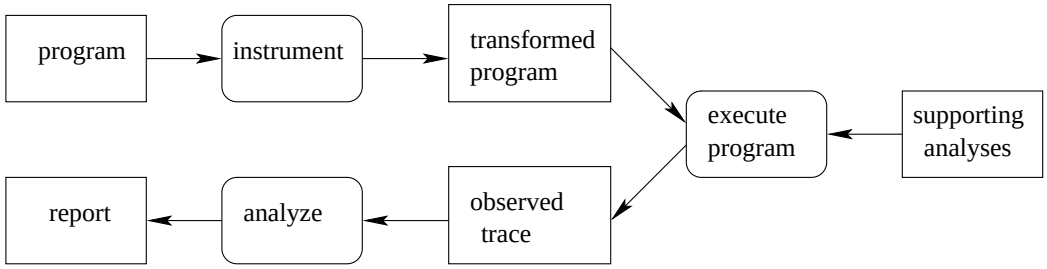

Figure 1.2: The architecture of our runtime analysis.

The architecture of our runtime analysis is shown in Figure 1.2. For a program to be checked, at first, it is instrumented by a modified compiler; then the transformed program is executed and a trace of observed events is preserved, some support analyses, such as dynamic escaped and happen-before analyses discussed in Chapter 3, are used to reduce the number of events to be saved and determine occurrence orders between events, respectively; at last, the observed trace is analyzed and a result is reported.

We designed three main algorithms for runtime detecting potential atomicity violations. They are discussed in Chapters 4, 5, and 6, respectively. The reduction-based algorithm is based on Lipton's reduction theorem [31]. It determines whether there is a data race on each variable and uses this information to determine commutativity of events, then checks whether the pattern of commutativity matches a pattern that implies atomicity. The block-based algorithm constructs blocks (*i.e.*, pairs of events plus associated synchronization) from an observed execution, and then compares all pairs of blocks with atomicity violations patterns. The commit-node algorithm organizes the events in each transaction into a tree, then detects atomicity violations by analyzing relationships between nodes in different trees.

Our algorithms do not merely look for violations of atomicity in the observed execution, but also attempt to determine whether the non-determinism of thread scheduling could allow violations in other executions. We implemented the three algorithms. Experiments of Chapter 7 show that they can successfully find subtle errors.

Our algorithms rely on defaults or information from the user to determine which execution fragments should be considered as transactions. Such user input would typically be provided by annotating some code blocks as expected to be atomic, and considering executions of those code blocks as transactions. In either case, our algorithms can automatically check atomicity of the indicated transactions. In contrast, atomicity type systems may require additional help (in the form of type annotations) from the user to determine whether specified code blocks are atomic. Of course, the defaults, whatever they are, will sometimes not capture the user's intentions accurately, so input from the user is desirable but not always available in practice.

- *Static analysis of atomicity for non-blocking algorithms.*

  Non-blocking (sometimes called lock-free) synchronization is becoming increasingly popular because it offers better performance, immunity to deadlock, and other advantages. Non-blocking synchronization primitives, such as Compare-and-Swap and Load-Linked/Store-Conditional, are supported by common processor architectures, such as Intel, Sun SPARC, PowerPC, and MIPS. Lock-based algorithms rely on mutual exclusion, while non-blocking algorithms instead adopt an optimistic approach that detects conflicts and redoes the computation when conflicts occur. Concurrent programming with locks is already difficult; non-blocking synchronization is significantly more difficult to use correctly.

  Chapter 8 presents a static analysis to determine atomicity of code blocks in programs that use non-blocking synchronization. Static analysis reasons over programs' source code without actually executing programs. Static analysis is often considered as a fundamental tool for analyzing and verifying programs and systems. There are many techniques for static analysis, such as data flow analysis, type and effect systems [36], and abstract interpretation [10]. Our static analysis is based on a type system for atomicity. We successfully applied the static analysis to several well-known non-blocking algorithms. This is the first automated and effective atomicity analysis for those algorithms.

- *Runtime approaches for detecting potential data races and potential deadlocks.*

  Runtime detection of data races is a classic problem. It is needed for some of our atomicity checking algorithms. Chapter 4 presents a new race detection algorithm, called the *multi-lockset algorithm*, that imposes slightly more overhead than the well-known lockset algorithm [44] and is more accurate (fewer false alarms and fewer missed races).

  Chapter 9 describes an algorithm that analyzes the synchronization in a monitored execution to detect *potential* deadlocks, even if deadlocks do not occur in the observed executions.

- *Hybrid analysis.*

  Dynamic analysis often incurs significant overhead. We investigated the use of static analysis to automatically and significantly reduce the overhead of runtime checking. The integration of static and dynamic analyses is called *hybrid analysis*. We successfully applied this strategy to runtime detection of atomicity violations, data races, and deadlocks. For example, for the reduction-based atomicity checking algorithm, static analysis reduced the overhead by a factor of 10-20 in our experiments. Our work on hybrid analysis is described in Chapter 9.

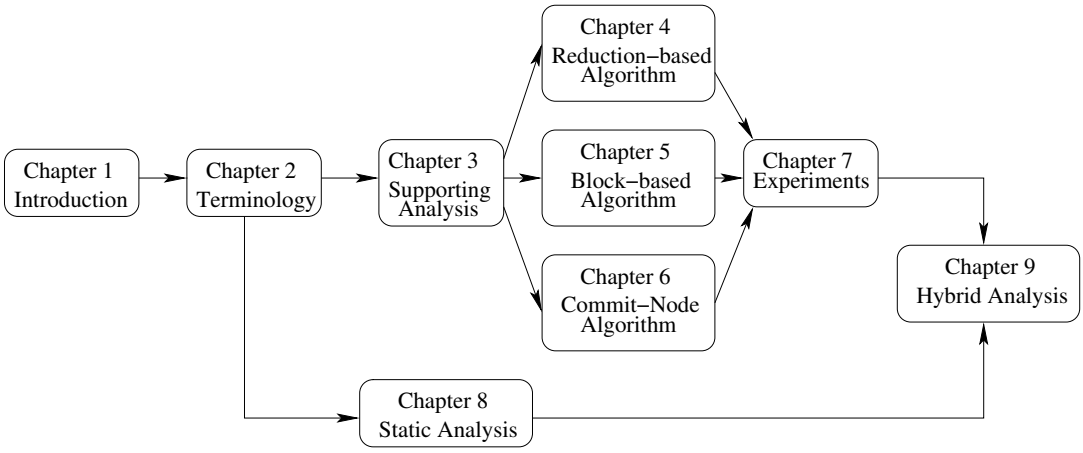

Figure 1.3: Chapter dependences.

1.3 Organization

This dissertation is organized as follows. Chapter 2 defines essential terms. Chapter 3 presents a dynamic escape analysis and a happen-before analysis that support the following runtime analyses for atomicity violations, data races, and deadlocks. Chapters 4, 5, and 6 describe the reduction-based, block-based, and commit-node runtime algorithms, respectively, for detecting atomicity violations. Chapter 7 summarizes the experiments with these three algorithms and related work. Chapter 8 presents a static analysis of atomicity for non-blocking algorithms.

The dependences between chapters are outlined in Figure 1.3.

Chapter 2

Terminology

This chapter defines the essential terms used in this dissertation. Figure 2.1 shows a class for bank account, which is used to illustrate some of the definitions.

Event. Informally, an *event* is one step in an execution of a program. This dissertation considers the following operations as events: read and write escaped (*i.e.*, accessible to multiple threads) variables; acquire and release locks*; start and join threads; start and exit invocations of methods; and the barrier synchronization operation discussed in Section 3.2. For example, `synchronized`(l) $\{$*body*$\}$ in Java indicates two events (in addition to the events performed by the body): acquiring lock l at the entry point and releasing it at the exit point. Two distinct accesses (even using the same operation) to a variable are different events. Let $held(e)$ denote the locks held by the thread executing event e

*This dissertation considers only the structured locking using `synchronized`; the unstructured locking introduced in JDK 5 is ignored.

```
class Account{
  private int bal;

  private synchronized int getBalance() {return bal;}
  private synchronized void setBalance(int value) {bal = value;}
  public void deposit(int value){
    int tmpBal = getBalance();
    tmpBal += value;
    setBalance(tmpBal);
  }
}
```

Figure 2.1: An example of bank account.

when e is executed. Let $var(e)$ denote the variable on which event e operates. Here, a variable means a storage location, *e.g.*, a field of an object. Two read or write operations *conflict* if they act on the same variable and at least one operation is a write. For the example in Figure 2.1, let l denote the lock acquired and released by methods `getBalance` and `setBalance`. The execution of method `deposit` contains the following events: $acq(l); R(bal); rel(l); acq(l); W(bal); rel(l)$. Here, $held(R(bal))$ and $held(W(bal))$ equal to $\{l\}$.

Unit and Transaction. A *unit* is a sequence of consecutive events executed by a single thread. A *transaction* is a unit expected to behave atomically. Executions of the following code fragments are considered as transactions by default in this dissertation: non-private methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods; as exceptions, executions of the `main()` method in which the program starts and executions of `run()` methods of classes that implement `Runnable` are not considered as transactions, because these executions represent the entire executions of threads and are often not expected to be atomic. Note that for nested transactions, we check atomicity of only the outermost transactions, since they contain the inner transactions. For the example in Figure 2.1, the executions of methods `getBalance`, `setBalance`, and `deposit` are treated as transactions. But when checking atomicity, only the executions of methods `getBalance` are checked because they contain the executions of the other two methods. Moreover, start, join and barrier operations are treated as unit boundaries, *i.e.*, they separate the preceding events and following events into different units, and are not contained in any unit. We adopt this heuristic because execution fragments containing these operations are typically not atomic and hence are not expected to be transactions. Events (other than start, join, and barrier operations) not in transactions form non-transactional units.

Trace. A *trace* tr is a sequence of events. Given $\langle T, E\rangle$, where T is a set of transactions, and E is a set of non-transactional units, *a trace of* $\langle T, E\rangle$ is an interleaving of events from units in $T \cup E$ that is consistent with the original order of events from each thread and with the synchronization events (*e.g.*, no lock is held by multiple threads at the same time). A trace of $\langle T, E\rangle$ must contain all events from units in $T \cup E$ unless the trace ends in deadlock. We assume that E contains no synchronization; this assumption is satisfied if synchronized blocks are considered to be transactions.

Initial Read and Final Write. Let e_x^r and e_x^w denote a read event and a write event to variable x, respectively. e_x^w is the *write-predecessor* of e_x^r in a trace tr if e_x^w is the last write to x that precedes e_x^r in tr. e_x^r is called a *unit-initial read* if e_x^r does not have any write-predecessor in its own unit in all traces. e_x^r is called a *trace-initial read* in trace tr if e_x^r is not preceded by a write to x in tr. Its write-predecessor is defined to be an imaginary write event e_x^{init} at the beginning of the trace. A write event e_x^w is called a *unit-final write* if it is

the last write to x in its unit; a write event e_x^w is called a *trace-final write* in a trace if it is the last write to x in the trace.

Conflict-Equivalence. Two traces tr_1 and tr_2 for $\langle T, E\rangle$ are *conflict-equivalent* iff (*i*) they contain the same events, and (*ii*) for each pair of conflicting events, the two events appear in the same order in both traces. This corresponds to conflict equivalence in transaction processing in database systems [5]. Suppose T consists of two executions of method `deposit` by two different threads t_1 and t_2, *i.e.*,

$$\begin{aligned} T = \{&\langle acq_{t_1}(l); R_{t_1}(bal); rel_{t_1}(l); acq_{t_1}(l); W_{t_1}(bal); rel_{t_1}(l)\rangle, \\ &\langle acq_{t_2}(l); R_{t_2}(bal); rel_{t_2}(l); acq_{t_2}(l); W_{t_2}(bal); rel_{t_2}(l)\rangle\} \end{aligned} \tag{2.1}$$

tr_1 and tr_2 are shown below. tr_1 and tr_2 are not conflict-equivalent because the order of $W_{t_1}(bal)$ and $R_{t_2}(bal)$ is different in tr_1 and tr_2.

$$\begin{aligned} tr_1 : &acq_{t_1}(l); R_{t_1}(bal); rel_{t_1}(l); \\ &acq_{t_1}(l); W_{t_1}(bal); rel_{t_1}(l); \\ &acq_{t_2}(l); R_{t_2}(bal); rel_{t_2}(l); \\ &acq_{t_2}(l); W_{t_2}(bal); rel_{t_2}(l); \\ tr_2 : &acq_{t_1}(l); R_{t_1}(bal); rel_{t_1}(l); \\ &acq_{t_2}(l); R_{t_2}(bal); rel_{t_2}(l); \\ &acq_{t_2}(l); W_{t_2}(bal); rel_{t_2}(l); \\ &acq_{t_1}(l); W_{t_1}(bal); rel_{t_1}(l) \end{aligned} \tag{2.2}$$

View-Equivalence. Two traces tr_1 and tr_2 for $\langle T, E\rangle$ are *view-equivalent* iff (*i*) they contain the same events, (*ii*) each read event has the same write-predecessor in both traces, and (*iii*) each variable has the same trace-final write event in both traces. This corresponds to view equivalence in transaction processing [5]. It is easy to show that conflict-equivalence implies view-equivalence [5], and that the converse does not hold. For traces tr_1 and tr_2 shown above, tr_1 and tr_2 are not view-equivalent because the write-predecessor for $R_{t_2}(bal)$ is $W_{t_1}(bal)$ and the initial value in tr_1 and tr_2, respectively.

Conflict-Serializability and View-Serializability. A trace of $\langle T, E\rangle$ is *serial* if the events of each transaction in T form a contiguous subsequence of the trace. Note that the events in non-transactional units in E are not required to be contiguous. A trace of $\langle T, E\rangle$ is *conflict-serializable* if it is conflict-equivalent to some serial trace of $\langle T, E\rangle$. A trace of $\langle T, E\rangle$ is *view-serializable* if it is view-equivalent to some serial trace of $\langle T, E\rangle$. Conflict-serializability of a trace tr for $\langle T, E\rangle$ can be decided in polynomial time [5]. Let g be the *serialization graph* for tr, which is a directed graph whose nodes are the units of $T \cup E$, and which contains an edge from node t_i to node t_j if $i \neq j$ and some event of t_i precedes a conflicting event of t_j in tr. tr is conflict-serializable iff g does not contain any

cycle containing two or more transactions. In contrast, checking view serializability of a trace is NP-complete [38].

Conflict-Atomicity and View-Atomicity. $\langle T, E\rangle$ is *conflict-atomic* if every trace of $\langle T, E\rangle$ is conflict-serializable. $\langle T, E\rangle$ is *view-atomic* if every trace of $\langle T, E\rangle$ is view-serializable. It is easy to show that conflict-atomicity implies view-atomicity, but the converse does not hold. As an example, consider $\langle\{t_1, t_2\}, \emptyset\rangle$, where t_1 is $\langle W_1(x); W_2(x)\rangle$, and t_2 is $\langle W(x)\rangle$. When $t_2.W(x)$ happens between $t_1.W_1(x)$ and $t_1.W_2(x)$, the trace does not have any conflict-equivalent serial trace, hence $\langle\{t_1, t_2\}, \emptyset\rangle$ is not conflict-atomic; but the trace is view-equivalent to a serial trace $\langle t_2.W(x); t_1.W_1(x); t_1.W_2(x)\rangle$, and all the other possible traces are serial, hence $\langle\{t_1, t_2\}, \emptyset\rangle$ is view-atomic.

Potential for Deadlock. $\langle T, E\rangle$ has *potential for deadlock* if some trace of $\langle T, E\rangle$ ends in deadlock. A trace that ends in deadlock with some thread in the middle of a transaction is not equivalent to any serial trace. Therefore, our atomicity checking algorithms assume that $\langle T, E\rangle$ has no potential for deadlock. We detect potential for deadlock using an extension of the goodlock algorithm [24]. Our algorithm reports a warning (of potential deadlock) if two threads acquire two locks ℓ_1 and ℓ_2 in different orders in concurrent thread periods (as defined in Section 3.2) without first acquiring some other lock that prevents their attempts to acquire ℓ_1 and ℓ_2 from being interleaved. This algorithm is unsound because it can miss potential for deadlocks involving three or more threads and locks, and it can miss deadlocks due to synchronization other than locks (e.g., `wait` and `notify`). The algorithm can be extended to detect potential for deadlock involving any number of threads, as described in Section 9.4, but this is more expensive and, we believe, generally not worthwhile in practice.

Chapter 3

Supporting Analyses

This chapter summarizes runtime analyses that support the atomicity checking algorithms presented in Chapters 4 - 6. Dynamic escape analysis determines when an object escapes from its creating thread, *i.e.*, when it becomes accessible to other threads. It provides two benefits for the atomicity checking algorithms: it improves their efficiency by eliminating processing of accesses to unescaped variables, and improves their accuracy by reducing false alarms. Happen-before analysis determines whether two events of different threads are concurrent. It improves the accuracy of the atomicity checking algorithms by eliminating some false alarms.

3.1 Dynamic Escape Analysis

This section describes how to determine when an object escapes from its creating thread. Before an object escapes, all operations on it can be ignored when checking atomicity.

We say that an object o' *refers to* an object o if $o'.f == o$ for some field f of o', or if o' is an array and $o'[i] == o$ for some index i. An object o may escape from its creating thread when:

- o is stored in a static field or a field of an escaped object.
- o is an instance of `Thread` (including its subclasses) and o.`start` is called. Note that, if o was created by a constructor with a `Runnable` argument r, then o refers to r, so (by the next rule) r escapes when o starts.
- If o' refers to o, and o' escapes, then o escapes. This leads to cascading escape.
- o is passed as an argument to a native method that may cause it to escape.

We implemented a dynamic escape analysis according to the above cases. To indicate whether an object has escaped, a boolean instance field `escaped` is added to every instrumented class, its initial value is `false`. To detect when an object escapes, we instrument all method calls, and all stores to static fields, instance fields, and arrays. When an object escapes, it is marked as escaped by setting its `escaped` field to `true`, and all objects to which it refers are marked as escaped (and so on, recursively). The reflection mechanism in Java is used to dynamically find all objects to which a given object refers. When an array escapes, all of its elements are marked as escaped. Since fields cannot be added to Java's built-in array classes, we use a hash table that maps from an array reference to a "shadow" object containing an `escaped` field for the array.

For methods whose bodies are not instrumented, specifically, all native methods and methods of uninstrumented library classes, escape information is maintained conservatively by marking all arguments to those methods as escaped. For some frequently used library classes (in particular, collections and maps), we instead use hand-written wrappers that track escape information more accurately without instrumenting the source code of the library classes. For example, our wrapper for the native method `Vector.add` marks the argument as escaped only if the target object (*i.e.*, the vector) is escaped. Instrumenting library code is sometimes complicated by dependencies between classes, so we instrument library classes only when specifically testing their own atomicity in the experiments of Chapter 7. More details about instrumentation also appear in Chapter 7.

Compared with this escape analysis, the technique in [11] is more expensive and more precise, since our escape analysis does not consider ownership transfer. Our atomicity checking algorithms could easily use a static escape analysis, such as [42], instead of a dynamic one, which is generally more expensive and more precise [11].

3.2 Happen-Before Analysis

The execution of a thread is separated into different *periods* by occurrences of synchronization events. A thread period *happens before* another thread period if it must end before the other thread period starts.

Our happen-before analysis tracks only happen-before relationships induced by `start` and `join` on threads and by barrier synchronization. A barrier is a rendezvous point for a specified number n of threads. Once all n threads reach the barrier, these threads may continue executing. Happen-before analysis can tell whether two events of different threads are concurrent, which is essential for detecting data races and atomicity violations. More details about its usage are described in Chapters 4.5 and 5.1. Happen-before relationships induced by `wait` and `notify` could also be analyzed; we do not do this because we believe that `wait` and `notify` are rarely used to achieve atomicity.

An identification number (ID) is assigned to each period of each thread. For an event e, let $tpID(e)$ denote the ID of the thread period in which e was executed. A directed

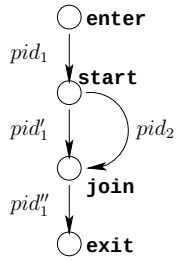

Figure 3.1: The happen-before graph for thread t_1 and t_2.

acyclic graph, called *happen-before graph*, with an edge for each ID is used to store the temporal ordering relations between thread periods. pid_1 happens before pid_2 if the edge labeled with pid_2 is reachable from the edge labeled with pid_1. If two thread periods pid_1 and pid_2 are not related to each other by happen-before relations, then we say that they are concurrent, denoted $pid_1 \parallel pid_2$. An event in thread period pid_1 can be concurrent with an event from thread period pid_2 only if pid_1 is concurrent with pid_2. Each node of the graph is labeled with `start`, `join`, barrier, `exit` (which denotes the end of a thread or the program), or `enter` (which denotes the starting point of the program). All events in one unit (defined in Chapter 2) have the same thread period id.

When a thread t_1 in period pid_1 calls t_2`.start()` to start another thread t_2, we introduce an ID pid'_1 for the new period of t_1, and an ID pid_2 for the first period of t_2, and we add a `start` node, as shown in Figure 3.1. Note that pid_1 happens before pid_2.

When thread t_1 in period pid'_1 calls t_2`.join()` to wait for thread t_2 in period pid_2 to terminate, the ID of t_1 is changed from pid'_1 to pid''_1, as shown in Figure 3.1. Note that pid_1, pid'_1 and pid_2 happen before pid''_1, and pid'_1 is concurrent with pid_2.

When a thread reaches a barrier, the thread changes its period ID. In the happen-before graph, we add a node for that barrier. For each participating thread, that node has an incoming edge labeled with the old period ID of the thread, and an outgoing edge labeled with the new period ID of the thread. For a barrier node, the thread periods on the incoming edges happen before the thread periods on the outgoing edges.

Examining paths in the graph to determine concurrency can be slow when the graph is large, so we cache the results of concurrency queries.

A more efficient but more complicated alternative is to use vector clocks [32], as in [37].

Chapter 4

Runtime Reduction-based Algorithm

4.1 Introduction

This chapter presents the reduction-based algorithm for checking conflict-atomicity [51, 54], defined in Chapter 2. For convenience, "atomicity" is short for "conflict-atomicity" in this chapter.

The reduction-based algorithm is an extension of our original reduction-based algorithm [51] and Flanagan and Freund's Atomizer algorithm [15]. It determines whether there is a data race on each variable and uses this information to classify events. If the sequence of events in each transaction matches a given pattern, then the transactions are atomic.

The reduction-based algorithm can applied *on-line* (*i.e.*, the analysis is applied during execution of the program, and warnings are issued based on the information observed so far) or *off-line* (*i.e.*, the analysis is applied after the program terminates, and warnings are issued based on the entire execution).

The reduction-based algorithm is based on Lipton's reduction theorem [31]. The idea is to infer atomicity from commutativity properties of events.

4.2 Commutativity Properties

Following [31, 20], events are classified according to their commutativity properties. An event is a *right-mover* if, whenever it appears immediately before an event of a different thread, the two events can be swapped (*i.e.*, they can be executed in the opposite order without blocking) without changing the resulting state. A *left-mover* is defined similarly.

For example, if an event e_1 of thread t_1 is a lock acquire, its immediate successive event e_2 from another thread can not be a successful acquire or release of the same lock, because an acquire would block, and a release would fail (in Java, it would throw an exception). Hence e_1 and e_2 can be swapped without affecting the result, so e_1 is a right-mover. Lock release events are left-movers for similar reasons.

An event is a *both-mover* if it is both a left-mover and a right-mover. For example, if there are only read events (no write) on a given variable, the read events commute in both directions with all events, so these read events are both-movers.

Events not known to be left or right movers are *non-movers*.

For Java programs, a classification of events can conveniently be obtained based on synchronization operations. Lock acquire events are right-movers. Lock release events are left-movers. Race-free reads and race-free writes are both-movers [20]. The thread start and join, method enter and exit, and barrier synchronization events are used as transaction boundaries, and are not contained in any transaction.

4.3 Basic Reduction-Based Algorithm

Given an arbitrary interleaving of all events in T, if all events of each transaction can be moved together by repeatedly swapping left-movers with the preceding events, and right-movers with the subsequent events, and if no trace for T ends in deadlock, then T is atomic, because the resulting trace is serial and equivalent to the original trace. If some transaction t contains two or more non-movers, the non-movers could interleave with non-movers in other transactions, preventing the events of transaction t from being moved together. If each transaction t in T has at most one non-mover e, and each event in t that precedes e can be moved to the right (towards e), and each event in t that follows e can be moved to the left (towards e), then all events of each transaction can be moved together. These observations motivate the following theorem, where R, L, and N denote right-mover, left-mover, and non-mover, respectively.

Theorem 4.3.1. *A set T of transactions is atomic if T has no potential for deadlock, and each transaction in T has the form $R^*N^?L^*$.*

Proof. This is a simple variant of Lipton's reduction theorem [31]. □

This theorem, together with a technique for runtime detection of data races (such the lockset algorithm in [44]), leads directly to an efficient runtime algorithm for checking atomicity. But this algorithm reports false alarms in several cases. The following sections show how to improve it.

This algorithm for runtime checking atomicity was first proposed in [51] and then [15], and is regarded as a runtime analogue of Flanagan and Qadeer's atomicity type system [19].

4.4 Improvement 1: Read-only and Thread-local Variables

A variable is *thread-local* if it is accessed by a single thread. A variable is *read-only* if it is never written, except for the initialization when it is allocated. Accesses to thread-local and read-only variables are race-free and hence are both-movers [15, 23].* The following theorem treats an entire synchronization block that contains only read-only or thread-local accesses as a both-mover. This is similar but not identical to ideas in [15] and [23], which treat thread-local and protected locks specially, as described below in Section 4.6. The improvement expressed in the following lemma and theorem makes no assumption about the lock being acquired and released, except that acquiring lock does not lead to potential for deadlock.

Let $AcqRel$ denote an acquire of some lock immediately followed by a release of the same lock. Let $AcqA^*Rel$ denote an acquire of some lock, then followed by accesses to read-only or thread-local variables, finally followed by release of the same lock.

Lemma 4.4.1. *Given a set T of transactions, T is atomic if T has no potential for deadlock and each transaction in T has the form $(R|AcqRel)^*N^?(L|AcqRel)^*$.*

Proof. Based on Theorem 4.3.1, it suffices to argue that $AcqRel$ can be ignored when determining atomicity. The only effect that $AcqRel$ could have is to cause a deadlock. This is avoided by the requirement that T has no potential for deadlock. Thus, $AcqRel$ has no effect on the state of the program and the commutativity properties of other operations (e.g., it does not affect whether any accesses to variables are race-free). □

Theorem 4.4.2. *A set T of transactions is atomic if T has no potential for deadlock and each transaction in T has the form $(R|AcqA^*Rel)^*N^?(L|AcqA^*Rel)^*$.*

Proof. This follows from Lemma 4.4.1 and the fact that events in A commute with all events from other threads, so they have no effect on atomicity. □

On-line (*i.e.,* during execution of the program) classification of variables as read-only or thread-local is based on whether the variable has been read-only or thread-local so far; thus, the classification of a variable may change afterwards. Off-line (*i.e.*, after the program terminates) classification is based on the entire execution and is therefore more accurate.

This improvement can be viewed as synchronization elimination, since the idea is to ignore synchronization operations that do not affect the behavior of the program. Prior work on synchronization elimination, such as [42], is generally based on static analysis and intended for optimization, while we use this idea in a runtime analysis to reduce false alarms in program checking. Also, the particular case of synchronization elimination described here has not been considered in the static context, to the best of our knowledge.

*Although the Java Memory Model allows the initializing write to be involved in a data race [40], we do not consider this or other issues raised by Java's controversial weak memory model.

4.5 Improvement 2: Multi-Lockset Algorithm for Runtime Race Detection

To classify read and write events as both-movers or non-movers, we need to determine whether there is a data race involving these events. This section briefly reviews related work on runtime race detection and then proposes a more precise (and more expensive) algorithm. Naturally, more precise race detection allows more precise reduction-based atomicity checking.

The Eraser algorithm [44], also called the lockset algorithm, is a classic runtime race detection algorithm based on the policy that each shared variable should be protected by a lock that is held whenever the variable is accessed. The algorithm works as follows. For each variable x, a set $lockset(x)$ of locks is maintained. A lock l is in $lockset(x)$ if every thread that has accessed x was holding l at the moment of access. $lockset(x)$ is initialized to contain all locks. Let $locksHeld(t)$ denote the set of locks currently held by thread t. When a thread t accesses x, the lockset is refined (updated) by $lockset(x) := lockset(x) \cap locksHeld(t)$, except during the initialization period when x is assumed to be accessible only by the thread that allocated it and the lockset retains its initial value. [44] supposes that the initialization period ends when the variable is accessed by a second thread; this is a heuristic that may cause the the algorithm to miss some races, but it is easy to implement. When $lockset(x)$ becomes empty, it means that no lock protects x. At that time, if there have been writes to x after the initialization period for x (hence x is not read-only), a warning is issued, indicating a potential data race. To see why this treatment of initialization may miss races, consider four consecutive events by two concurrent threads t_1 and t_2: x is allocated in t_1, then escapes to be accessible to t_2 (note that t_2 does not actually access x yet), and then is accessed by t_1 and then t_2 without holding any locks; a data race occurs, but this algorithm does not report it.

von Praun and Gross [49] modify the lockset algorithm by introducing a more sophisticated condition for determining when initialization ends. [49] supposes that when a variable is accessed by a second thread, its ownership is also transferred. Thus, $lockset(x)$ is not refined until a "third" thread (possibly the same as the first thread) accesses x. This algorithm may miss even more races than the original lockset algorithm. On the positive side, it may produce fewer false alarms. For efficiency, [49] treats an entire object (instead of a field of an object) as a single variable. This reduces the number of maintained locksets but increases the number of false alarms.

[15] improves the lockset algorithm to avoid false alarms in multiple-reader, single-writer scenarios. For each variable, a pair of locksets is used instead of one lockset: the *access-protecting* lockset contains locks held on every access (read or write) to the variable, and the *write-protecting* lockset contains locks held on every write to the variable. The two locksets for a variable are updated based on their definitions at each access to that variable. A read event on a variable x is race-free if the current thread holds at least one of the write-protecting locks for x, otherwise a potential data race is reported. A write event on a variable x is race-free if the access-protecting lockset of x is not empty, otherwise a

potential data race is reported.

We propose the *multi-lockset algorithm*, which is more accurate than the preceding algorithms, *i.e.*, it misses fewer races and reports fewer false alarms. It incurs higher overhead but is still practical, according to the experiments in Chapter 7. The three main improvements are: (1) The algorithm uses a dynamic escape (from thread) analysis, described in Section 3.1, to determine when "initialization" of a variable ends, *i.e.*, when to start refining the variable's lockset. This improves accuracy because only the accesses before the variable escapes are ignored. (2) The happen-before relation based on `start` and `join` operations on threads and barrier operations is considered. (3) The analysis maintains multiple read-protecting locksets besides a write-protecting lockset. The access-protecting lockset used in [15] is equivalent to maintaining only one read-protecting lockset, and this can cause some false alarms, as illustrated below.

For each variable x, we maintain:

- *ReadSets(x)*, which contains $\subseteq$-minimal sets of held locks for read events on x. In other words, for each read of x, we insert $locksHeld(t)$ in $ReadSets(x)$ and then, if $ReadSets(x)$ contains S_1 and S_2 such that $S_1 \subseteq S_2$, we remove S_2.
- *WriteSet(x)*, which is the set of locks held on all writes to x, *i.e.*, for the first write, $WriteSet(x) := locksHeld(t)$, and for each subsequent write to x, $WriteSet(x) := WriteSet(x) \cap locksHeld(t)$.
- *ReadThreadSet(x)*, which contains the IDs of thread periods involving read events on x.
- *WriteThreadSet(x)*, which contains the IDs of thread periods involving write events on x.

$ReadSets(x)$, $WriteSet(x)$, $ReadThreadSet(x)$ and $WriteThreadSet(x)$ are not updated by accesses to x before x escapes. The happen-before analysis described in Section 3.2 determines whether two thread periods can happen concurrently. If there are not concurrent thread periods inside $WriteThreadSet$ or between $ReadThreadSet$ and $WriteThreadSet$, *i.e.*, there are not concurrent conflicting accesses according to the happen-before analysis, the variable must be free of data race. For sets P_1 and P_2 of thread period IDs, $isConcurrentWith(P_1, P_2)$ returns true if some thread period in P_1 is concurrent with some thread period in P_2. When P_1 or P_2 is empty, $isConcurrentWith(P_1, P_2)$ returns false. $isConcurrent(P_1)$ returns true if P_1 contains concurrent thread periods. When P_1 is empty, $isConcurrent(P_1)$ returns false. The pseudo code for the multi-lockset algorithm is shown in Figure 4.1. The first case (*i.e.*, $WriteSet(x)$ is uninitialized) implies that there is no write to x so far, hence no data race. The second case (*i.e.*, $WriteSet(x)$ is empty) uses a conservative test: a potential data race is reported if there is no common lock held on all writes to x, and a write and another access to x occur in concurrent thread periods. The third case (*i.e.*, $WriteSet(x)$ is non-empty) also uses a conservative test: a potential data race is reported if some common locks are held at all writes, and none of

```
if (x is not escaped)
   return no-race-so-far;
/*update protecting locksets and thread period sets for each escaped variable access */
if (e is read){
   /* All locksets in ReadSets(x) are not subset of the lockset held by the current thread. */
   if (∄ls ∈ ReadSets(x). ls ⊆ locksHeld(t)){
      /* Remove each lockset in ReadSets(x) that is a superset of locksHeld(t),
         then put locksHeld(t) into ReadSets(x). */
      ReadSets(x) := (ReadSets(x) − {ls|ls ∈ ReadSets(x)∧ ls ⊇ locksHeld(t)}) ∪ {locksHeld(t)}
   }
   ReadThreadSet(x) := ReadThreadSet(x) ∪ {t}
} else { /* e is write */
   WriteSet(x) := WriteSet(x) ∩ locksHeld(t)
   WriteThreadSet(x) := WriteThreadSet(x) ∪ {t}
}
/* check whether data race occurs */
switch (WriteSet(x)) {
   case uninitialized: /* no write to x so far. */
      return no-race-so-far;
   case empty:
      /* there is no common lock held on all writes to x*/
      if (IsConcurrent(WriteThreadSet(x)) or
         IsConcurrentWith(WriteThreadSet(x),ReadThreadSet(x)))
            return potential-race;
      else return no-race-so-far;
   case non-empty:
      if ∃ls ∈ ReadSets(x).
         ((ls ∩ WriteSet(x) == ∅) and IsConcurrentWith(WriteThreadSet(x),ReadThreadSet(x)))
            /* A potential data race is reported if some common locks are held at all writes, and
               none of those locks are held at some read, and there are at least two concurrent thread
               periods such that one performs a write to x and another performs a read to x. */
            return potential-race;
      else return no-race-so-far;
}
```

Figure 4.1: The pseudo code of multi-lockset algorithm.

those locks are held at some read, and there are at least two concurrent thread periods such that one performs a write to x and another performs a read to x.

According to the experiments in Chapter 7, this algorithm is practical because its storage requirement is reasonable compared with that of the Eraser algorithm, and their runtime overheads are similar.

This algorithm is more accurate than previous lockset-based algorithms. For example, suppose x has escaped from its creating thread, and the threads in Figure 4.2 execute in the order t_3, t_2, t_1 (ignore t_4 for now). According to the definition of data race, there is no data race on x. The algorithms in [44, 49, 15, 9] all report a false alarm (potential data race on x). The multi-lockset algorithm does not.

A similar scenario appears in an actual program `hedc` in our experiments described

```
t1:             t2:           t3:           t4:
sync(o1){       sync(o2){     syn(o1){      read(x);
  sync(o2){       read(x);      read(x);    sync(o1){
    write(x);   }             }               sync(o2){
    write(x);                                   write(x);
  }                                           }
}                                             }
```

Figure 4.2: An example to illustrate the accuracy of the multi-lockset algorithm.

Chapter 7. The specific scenario is: After the object of `MetaSearchResult` escapes, there is a thread, say t_1, reads the field `request` without lock, then another thread, say t_2, writes the field `request` with a lock L. For Eraser or similar algorithms ([44, 49, 15, 9]), when t_1 reads, we do not carry on lockset intersection; when t_2 writes, lockset intersection is performed and the result is the lock L. Therefore, a data race appears but cannot be detected by these algorithms. But the multilockset algorithm caches all lockset for read and write operations, hence, this data race is detected.

Even the multi-lockset algorithm produces some false alarms. For example, consider the threads t_2, t_3 and t_4 in Figure 4.2 (ignore t_1 now). If the execution order is t_3, t_2, t_4, since $ReadSets(x) = \{\emptyset\}$, $WriteSet(x) = \{o_1, o_2\}$, $ReadThreadSet = \{t_2, t_3, t_4\}$, and $WriteThreadSet = \{t_4\}$, the third case "non-empty" in Figure 4.1 is matched, a potential data race is reported, but this is a false alarm, and it causes a false alarm in atomicity checking. In Chapter 5, we will see that the block-based algorithm does not produce such a false alarm for this example.

4.6 Other Improvements

We can refine the classification of all lock acquires and releases as right-movers and left-movers, respectively, in Section 4.2. In the following cases, they are classified as both-movers [15, 23]. (1) *Re-entrant locks*, if the thread already holds the lock, an acquire and the corresponding release on the same lock are both-movers, because they have no effect on the execution of the program. (2) *Thread-local locks*, if a lock is used by only one thread, acquire and release on it are both-movers. (3) *Protected locks*. Lock l_2 is protected by lock l_1 if, whenever a thread holds l_2, it also holds l_1. Acquire and release by a thread t on a protected lock l_2 are both-movers, because adjacent operations of other threads cannot be operations on l_2 (because t holds l_1).

Off-line algorithms can classify thread-local locks and protected locks more accurately than on-line algorithms. For example, if a lock is protected for a while, but is unprotected later, acquire and release operations on the lock that precede this change will be wrongly classified as both-movers by on-line algorithms. This may cause atomicity violations to be missed.

The above improvements can be viewed as synchronization elimination, and static analyses that recognize these situations have been used for program optimization [42]. We follow the approach in [15] to recognize them using runtime analysis.

4.7 Implementation of Reduction-based Algorithm

In practice, many of the sets of locks manipulated by the lockset algorithm have size 0 or 1. To save space and time, each lockset is represented by a structure that contains `null` (if the lockset is empty), a direct reference to the element (if the lockset has size 1), or a collection (if the lockset has size greater than 1). Intersection operations could be optimized by maintaining the contents of each set in sorted order, but we did not implement this because most locksets are small.

Our on-line reduction-based algorithm is implemented following the design in [15] which does not use the transaction tree structure described below. Our implementation of the off-line reduction-based algorithm is described next.

We instrument the program by a source-to-source transformation. The instrumented program constructs a tree structure for each transaction during execution. The root corresponds to the entire transaction. Each node other than the root corresponds to a synchronized block (*i.e.*, an execution of a synchronized statement or synchronized method), and is labeled with the acquired lock. The tree structure reflects the nesting of synchronized blocks. In other words, if an execution σ' of a synchronized block is nested inside the execution σ of a synchronized block, the node for σ' is a descendant of the node for σ in the tree structure. Each node is also labeled with the set of variables accessed only once (denoted $varsOne$) and with the set of variables accessed multiple times (denoted $varsMul$) in the corresponding synchronized block ignoring accesses in sub-blocks, because we need to distinguish the two cases for non-movers, *i.e.*, at most one non-mover or multiple non-movers, if some accesses in the current blocks have data races. Obviously, $varsOne$ and $varsMul$ are disjoint.

For example, Figure 4.3 shows the tree structure for the `Vector` example in Figure 1.1. The four fields of each node contain the acquired lock (none in the root node), pointers to child nodes, the set of variables accessed only once (*i.e.*, $varsOne$), and the set of variables accessed multiple times (*i.e.*, $varsMul$), respectively. In this example, there is no access outside the three synchronization blocks, hence, the last two fields of both root nodes contain empty sets.

The atomicity of a transaction is determined as follows. (1) Determine the commutativity type of accesses to each variable in $varsOne$ and $varsMul$ at each node of the tree; the accesses are both-mover if accesses to the variable are race-free, otherwise they are non-mover. (2) For each node, construct the pattern (of commutativities of events represented by that node) by concatenating, in any order, the commutativity types of the variables in the node's $varsOne$ and $varsMul$ sets, and, if the node represents a synchronization operation, adding a R (right-mover) at the beginning of the pattern and adding a L (left-mover)

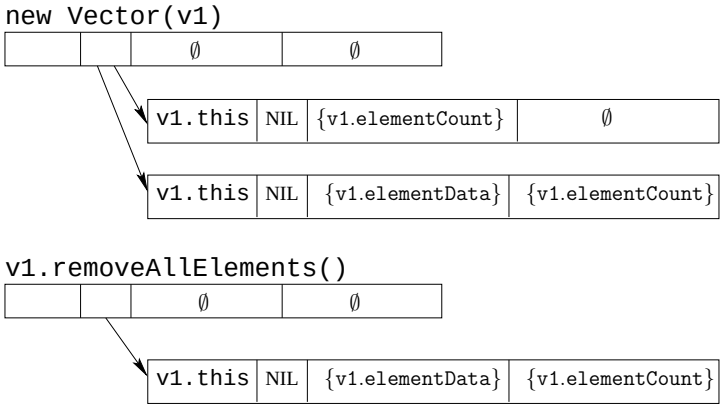

Figure 4.3: Tree structure for the `Vector` example of Figure 1.1.

at the end of the pattern. For each variable in $varsOne$, its commutativity type appears once in the pattern; for each variable in $varsMul$, its commutativity type appears twice. (3) Construct the pattern of commutativities for the transaction by concatenating the pattern for each node during a traversal of the tree (e.g., in-order traversal), except that the R and L representing acquire and release at a node are positioned before and after, respectively, the rest of the pattern constructed from the accesses represented by that node and its descendants. (4) Check whether the pattern of commutativities for the transaction matches the regular expression in Theorem 4.4.2. Note that the tree structure does not indicate the relative order of the accesses represented by a node and its descendants, and the result of matching against the regular expression in Theorem 4.4.2 is insensitive to that order, so any traversal order (in-order, pre-order, or post-order) may be used in step 3.

In Figure 4.3, the transaction for `new Vector(v1)` has the pattern $RBLRBBBL$ which does not match the atomicity in Theorem 4.4.2; the transaction for `v1.removeAllElements()` has the pattern $RBBBL$. Note that `v1.elementCount` is saved in $varsMul$, hence it is represented by two both-movers in the patterns constructed for each transaction.

Although the reduction-based algorithm is efficient for checking atomicity, it produces numerous false alarms in the experiments Chapter 7. This motivates us to design a novel and more accurate approach discussed in the next chapters.

Chapter 5

Runtime Block-based Algorithm

This chapter presents the block-based algorithm for checking view-atomicity [51, 54], defined in Chapter 2. For convenience, "atomicity" is short for "view-atomicity" in this chapter. The block-based algorithm first constructs blocks (*i.e.*, pairs of events plus associated synchronization) from an observed trace, and then compares each block with all blocks in other transactions. If two blocks are found that match any one of a set of unserializable patterns, the transactions containing them are not atomic.

The block-based algorithm is somewhat more expensive than the reduction-based algorithm but is more accurate, *i.e.*, reports fewer false alarms. This is demonstrated by the experiments in Chapter 7.

We first present an algorithm that works for the case of multiple transactions that share exactly one variable (note that locks and barriers are not counted as shared variables), then extend the ideas in it to handle the case of two transactions that share multiple variables. Finally, we extend that algorithm to handle the case of multiple transactions that share multiple variables.

5.1 Multiple Transactions That Share Exactly One Variable

Given a set T of transactions, the algorithm looks for unserializable patterns of operations of T. An *unserializable pattern* is a sequence in which operations from different transactions are interleaved in an unserializable way. If the transactions of T share exactly one variable, the following unserializable patterns are checked.

- a read in one transaction occurs between two writes in another transaction.
- a write in one transaction occurs between two reads in another transaction.
- a write in one transaction occurs between a write and a subsequent read in another transaction.

$R(x)$ $W(x)$ $W(x)$	$W(x)$ $R(x)$ $R(x)$
$W(x)$ $W(x)$ $R(x)$	$FW(x)$ $R(x)$ $W(x)$

Table 5.1: Part of unserializable interleaving patterns for operations on a single variable.

- the final write in one transaction occurs between a read and a subsequent write in another transaction.

Note that all of the operations in the patterns are on the same variable (the single shared variable). The above patterns are exhaust for detecting all atomicity violations based on exactly one variable. These patterns can be drawn as Table 5.1, where each line corresponds to a transaction, and time advances from left to right.

Informally, T is atomic if no trace for T contains a subsequence that matches any of these patterns; this idea is formalized in Theorem 5.1.2 below.

The block-based algorithm looks for these unserializable patterns by considering pairs of "blocks" from different transactions. Intuitively, a block captures the information about two events of the same transaction that is relevant to atomicity checking. Many pairs of events in a transaction may generate the same block. Our algorithm recognizes this, and stores only one copy of it. This can eliminate a significant amount of redundant storage and processing during atomicity checking. If the two events operate on the same variable, the block is called *1v-block*; if the two events operate on the different variables, the block is called *2v-block*. This section discusses only 1v-blocks; 2v-blocks are discussed in Section 5.2. An access to a variable that has not yet escaped is not used to form blocks.

All 1v-blocks for transaction t are generated by the algorithm shown in Figure 5.1. A *transaction-initial read* to a variable v for a transaction t is a read to v that is not preceded by a write to v in t. Similarly, A *transaction-final write* to a variable v for a transaction t is the final write to v in t. If there is only one event in a transaction, a dummy event is added. This dummy event is used only for constructing blocks, not for matching part of an unserializable pattern.

A *1v-block* for e_1 and e_2 is a tuple $\langle v, op(e_1), op(e_2), isFW(e_1), isFW(e_2), pid, held(e_1), held(e_2), held(e_1, e_2)\rangle$. The notations used for each element are explained next. The first element v is the variable on which e_1 and e_2 operate (recall that $var(e_1) = var(e_2)$). The second and third elements $op(e)$ is the operation type, namely, R (for "read"), W (for "write"), or *dummy*. The fourth and fifth elements $isFW(e)$ is a boolean value indicating whether e is the final write on v in t. The sixth element pid identifies the thread period in which e_1 and e_2 were executed, *i.e.*, $pid = tpID(e_1) = tpID(e_2)$; recall from Section 3.2 that each transaction occurs within a single thread period.* The seventh and eighth elements $held(e)$ is the set of locks held by the thread when executing event e. The last element $held(e_1, e_2)$ is the set of locks held continuously from e_1 to e_2.

*For efficiency, in our implementation, multiple 1v-blocks that differ only in the thread period IDs are represented by a single 1v-block that contains a set of thread period IDs.

```
1vBlockSet = ∅;
for each event e in t {
   if (e is not a read or write operation) continue;
   v = var(e);
   if (v is not escaped when e occurs) continue;
   if (there is a write to v before e in t) {
      e_w = the last write to v in t that precedes e;
      b = new1vBlock(e_w,e);
   } else
   if (there is a read to v before e in t) {
      e_r = the last read to v in t that precedes e;
      b = new1vBlock(e_r,e);
   }
   1vBlockSet = 1vBlockSet ∪ {b};
   if ((e is the transaction-final write to v in t) and
      (there is transaction-initial read to v in t)) {
      for each transaction-initial read e_r to v in t {
         b = new1vBlock(e_r,e);
         1vBlockSet = 1vBlockSet ∪ {b};
      }
   }
}
```

Figure 5.1: The algorithm of constructing 1v-blocks for transaction t.

For example, the transaction

$$t : acq(\ell_1)\ R(v)\ acq(\ell_2)\ W(v)\ R(v)\ rel(\ell_2)\ rel(\ell_1) \tag{5.1}$$

has two 1v-blocks, where pid is the thread period ID of t.

$$\begin{aligned} b_1 &: \langle v, R, W, \text{false}, \text{true}, pid, \{\ell_1\}, \{\ell_1, \ell_2\}, \{\ell_1\}\rangle \\ b_2 &: \langle v, W, R, \text{true}, \text{false}, pid, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}\rangle. \end{aligned} \tag{5.2}$$

To determine whether the operations in two blocks can form an unserializable pattern, we need to determine whether an operation of one block can occur between the two operations of another block. This is determined by locking and happen-before analysis. For 1v-blocks $b = \langle v, op_1, op_2, fw_1, fw_2, pid, h_1, h_2, h_{12}\rangle$, and $b' = \langle v, op'_1, op'_2, fw'_1, fw'_2, pid', h'_1, h'_2, h'_{12}\rangle$, an operation op_i ($i \in \{1, 2\}$) of b can occur between operations op'_1 and op'_2 of b', denoted *can-occur-between*($\langle op_i, h_i, pid\rangle$, $\langle op'_1, op'_2, h'_{12}, pid'\rangle$), if the condition ($h_i \cap h'_{12} = \emptyset$) $\wedge$ ($pid \parallel pid'$) is satisfied.

This simple test is accurate provided there is no potential for deadlock in the set of transactions. So we check potential for deadlock, as described in Section 4.7, as part of the block-based algorithm. To see that this test may be inaccurate if there is potential for deadlock, note that $op(e)$ cannot occur between $op(e_1)$ and $op(e_2)$ in the following example, even though the can-occur-between condition defined above is satisfied. This indicates that ignoring deadlock would lead to more false alarms.

$$\begin{array}{l} t : acq(l_1)\ acq(l_2)\ rel(l_2)\ e\ rel(l_1) \\ t' : acq(l_2)\ acq(l_1)\ rel(l_1)\ e_1\ e_2\ rel(l_2) \end{array} \tag{5.3}$$

As mentioned above, many pairs of events may produce the same 1v-block. For example, only one 1v-block is generated for the following transaction.

$$W(v)\ R(v)\ R(v)\ R(v) \tag{5.4}$$

Two 1v-blocks b and b' are *atomic* with respect to each other, denoted $isAtomic1vBlk(b, b')$, if the synchronization prevents the unserializable patterns described above, *i.e.*, the two operations from one block together with an operation from the other block cannot form one of those unserializable patterns. Formally, $isAtomic1vBlk(b, b')$ holds iff for all combinations of three operations op_1, op_2 and op_3, where op_1 and op_2 are from one block, op_3 is from the other block, either op_3 cannot occur between op_1 and op_2 or the sequence $op_1\ op_3\ op_2$ does not match any of the unserializable patterns. Obviously, $isAtomic1vBlk(b, b')$ is symmetric. For example, consider a 1v-block $b' = \langle v, W, dummy, \text{true}, \text{false}, pid', \emptyset, \emptyset, \emptyset \rangle$ in a different transaction t' than transaction t in (5.1); if $pid \parallel pid'$, then $isAtomic1vBlk(b_1, b')$ and $isAtomic1vBlk(b_2, b')$ do not hold because the unserializable patterns can be formed. For another example, consider a 1v-block $b'' = \langle v, R, dummy, \text{false}, \text{false}, pid'', \emptyset, \emptyset, \emptyset \rangle$; if $pid'' \parallel pid$, then $isAtomic1vBlk(b_1, b'')$ and $isAtomic1vBlk(b_2, b'')$ hold.

Let *1v-blocks*(t) denote the set of 1v-blocks for a transaction t. Lemma 5.1.1 describes how to check atomicity of two transactions that share exactly one variable. Theorem 5.1.2 describes how to check atomicity of multiple transactions that share exactly one variable.

Lemma 5.1.1. *Let t and t' be transactions that share exactly one variable with $thread(t) \neq thread(t')$, and suppose they do not have potential for deadlock. $\{t, t'\}$ is atomic iff $\forall b \in$ 1v-blocks$(t).\ \forall b' \in$ 1v-blocks$(t').\ isAtomic1vBlk(b, b')$.*

Proof. For the forward implication ($\Rightarrow$), we prove the contrapositive, *i.e.*, if $isAtomic1vBlk(b, b')$ is false for some pair of 1v-blocks b and b', then t and t' are not atomic. This follows easily from the definition of $isAtomic1vBlk$.

For the reverse implication ($\Leftarrow$), suppose $isAtomic1vBlk(b, b')$ holds for all pairs of 1v-blocks b and b'. Let S be a non-serial trace for $\{t, t'\}$. If neither transaction performs a write, then S is obviously equivalent to a serial trace. Suppose, without loss of generality, that t performs the final write e^t_{FW} in S. There are two cases:

(1). If t' does not read the value written by e^t_{FW}, then all reads and writes in t' precede

e^t_{FW} in S, and we can show that S is equivalent to the serial trace S' in which t' precedes t. The main point is that there is no read event $e^{t'}_R$ that reads the value written by any write e^t_W in S, because if there were, then e^t_W and e^t_{FW} would form a block b that can be interleaved in an unserializable way with $e^{t'}_R$, so $isAtomic1vBlk(b, b')$ would be false for some block b' containing $e^{t'}_R$, a contradiction. Similarly, we can show that each read event of t reads the value written by the same write event in S' and S. According to the definition for equivalence of traces in Chapter 2, S is equivalent to S'.

(2). If t' reads the value written by e^t_{FW}, then we can show that all reads and writes in t' appear after e^t_{FW} in S (because, if one of those events precedes e^t_{FW}, an unserializable pattern and hence a non-atomic pair of 1v-blocks would exist), and that S is equivalent to the serial trace in which t precedes t'. □

Theorem 5.1.2. *Let T be a set of transactions that share exactly one variable. Suppose T does not have potential for deadlock. T is atomic iff for all t, t' in T with $thread(t) \neq thread(t')$, $\forall\, b \in$ 1v-blocks(t). $\forall\, b' \in$ 1v-blocks(t'). $isAtomic1vBlk(b, b')$.*

Proof. For the forward implication ($\Rightarrow$), the proof is straightforward, except for details related to final writes.

We prove the reverse implication ($\Leftarrow$) by induction on the number of transactions in T. Let S be a non-serial trace for T. For $T' \subseteq T$, let $S|T'$ denote the subsequence of S containing only events from transactions in T'. Let t be the transaction that performs the final write e^t_{FW} in S. Let T_2 be the set of transactions in T other than t that read the value written by e^t_{FW}. No read or write from T_2 can precede e^t_{FW} in S (otherwise an unserializable pattern would be formed). This implies that T_2 contains no writes (otherwise e^t_{FW} would not be the final write). Thus, $S|T_2$ is equivalent to some serial trace S_2. Let T_1 be $T - T_2 - \{t\}$. For all $t_1 \in T_1$ with $thread(t) \neq thread(t_1)$, the hypothesis of the contrapositive case and Lemma 5.1.1 imply that $\{t, t_1\}$ is atomic; since t_1 does not read t's write, and t performs the final write in S, t_1 must precede t in every serial trace equivalent to $S|\{t, t_1\}$. Since t can be serialized after every transactions in T_1, S is equivalent to $(S|T_1) \cdot t \cdot S_2$, where the dot denotes concatenation. By the induction hypothesis, $S|T_1$ is equivalent to some serial trace S_1. Thus, S is equivalent to the serial trace $S_1 \cdot t \cdot S_2$. □

Let E be the total number of events in all transactions of T. Let P denote the number of thread periods; P is generally very small except when there are many calls to barrier operations. The algorithm shown in Figure 5.1 for constructing 1v-blocks generates at most $O(E)$ 1v-blocks because each event except for the final writes is combined with at most one preceding event, the number of these 1v-blocks is $O(E)$. All final writes generate at most $O(E)$ 1v-blocks because there is only one final write for each variable in each transaction. Hence, the number of all 1v-blocks is $O(E)$. The cost of checking can-occur-between for a pair of 1v-blocks is $O(P)$. Assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, the worst-case running time of the algorithm based on Theorem 5.1.2 is $O(PE^2)$.

0 read	$W(x)$ $W(y)$ $W(x)$ $W(y)$	$W(x)$ $W(y)$ $W(y)$ $W(x)$
1 read	$R(x)$ $W(y)$ $W(y)$ $W(x)$	$R(x)$ $W(y)$ $W(y)$ $W(x)$
2 reads	$R(x)$ $W(y)$ $W(x)$ $R(y)$	$R(x)$ $R(y)$ $W(y)$ $W(x)$

Table 5.2: Part of unserializable interleaving patterns for operations on two variables.

5.2 Two Transactions That Share Multiple Variables

To check atomicity of two transactions that share multiple variables, the test embodied in Theorem 5.1.2 needs to be strengthened.

Consider two events from transaction t, and two events from transaction t'. If they operate on four or three different variables, they cannot cause a violation of atomicity. If they all operate on the same variable, the analysis in Section 5.1 applies. Suppose they operate on two variables. If they contain no conflicting operations, or exactly one pair of conflicting operations, they do not cause a violation of atomicity. Suppose they contain two pairs of conflicting operations. We can check based on the definition of atomicity in Chapter 2 whether every feasible interleaving of the operations from the four events is serializable; if so, the two blocks are atomic. A few illustrative cases of unserializable interleavings are listed in Table 5.2.

Let $IR(t)$ and $FW(t)$ be the sets of initial reads and final writes, respectively, on shared variables in t. For events e_1 and e_2 of the same thread, let $heldmid(e_1, e_2)$ be the set of locks acquired by that thread after e_1 and released by it before e_2 and not contained in $held(e_1) \cup held(e_2)$; furthermore, re-acquires of held locks are ignored when computing $heldmid$.

A *2v-block* for a transaction t is a tuple $\langle var(e_1), var(e_2), op(e_1), op(e_2), pid, held(e_1), held(e_2), held(e_1, e_2), heldmid(e_1, e_2)\rangle$ formed from two (read or write) events e_1 and e_2 of t such that e_1 precedes e_2 in t, $var(e_1) \neq var(e_2)$, and e_1 and e_2 are in $IR(t) \cup FW(t)$, where $pid = tpID(e_1) = tpID(e_2)$. Let *2v-blocks(t)* denote the set of *2v-blocks* for transaction t. For example, for the following transaction t,

$$R_1(x)\ W_2(y)\ W_3(x)\ W_4(y)\ R_5(x) \tag{5.5}$$

$IR(t) = \{R_1(x)\}$, $FW(t) = \{W_3(x), W_4(y)\}$, and *2v-blocks(t)* contains $\langle x, y, R, W, pid, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ and $\langle x, y, W, W, pid, \emptyset, \emptyset, \emptyset, \emptyset\rangle$, assuming pid is the thread period ID of t. $W_2(y)$ and $R_5(x)$ do not participate in generating 2v-blocks because $W_2(y) \notin FW(t)$ and $R_5(x) \notin IR(t)$.

For 2v-blocks $b = \langle v_1, v_2, op_1, op_2, pid, h_1, h_2, h_{12}, hmid_{12}\rangle$ and $b' = \langle v'_1, v'_2, op'_1, op'_2, pid', h'_1, h'_2, h'_{12}, hmid'_{12}\rangle$, where $(v_1 = v'_1 \wedge v_2 = v'_2) \vee (v_1 = v'_2 \wedge v_2 = v'_1)$, the operations op'_1 and op'_2 of b' can occur between the operations op_1 and op_2 of b if (h_{12}

$\cap\ hmid'_{12} = \emptyset$) $\wedge$ *can-occur-between*($\langle op'_1, h'_1, pid' \rangle$, $\langle op_1, op_2, h_{12}, pid \rangle$) $\wedge$ *can-occur-between*($\langle op'_2, h'_2, pid' \rangle$, $\langle op_1, op_2, h_{12}, pid \rangle$).

Two 2v-blocks b and b' are *atomic* with respect to each other, denoted $isAtomic2vBlk(b, b')$, iff the four operations cannot be interleaved to any of the unserializable patterns described above, according to the can-occur-between conditions described in the previous paragraph and Section 5.1.

To check atomicity of two transactions that share multiple variables, we have the following theorem, which extends Theorem 5.1.2.

Theorem 5.2.1. *Let t and t' be transactions with $thread(t) \neq thread(t')$. Suppose T does not have potential for deadlock. $\{t, t'\}$ is atomic iff (i) $\forall\ b \in$ 1v-blocks(t). $\forall\ b' \in$ 1v-blocks(t'). $isAtomic1vBlk(b, b')$; and (ii) $\forall\ b \in$ 2v-blocks(t). $\forall\ b' \in$ 2v-blocks(t'). $isAtomic2vBlk(b, b')$.*

Proof. For the forward implication ($\Rightarrow$), we prove the contrapositive, which follows easily from the definitions of $isAtomic1vBlk$ and $isAtomic2vBlk$.

For the reverse implication ($\Leftarrow$), suppose all pairs of 1v-blocks and 2v-blocks are atomic. Let S be a non-serial trace for $\{t, t'\}$. We show that S is equivalent to some serial trace by the following three cases:

Case 1: Suppose there exists a variable x written by t and t'. Without loss of generality, we assume that t' performs the final write $e^{t'}_{FW(x)}$ to x in S. Let $e^{t}_{W(x)}$ denote a write to x in t. We can show that S is equivalent to the serial trace S' in which t precedes t', based on the following intermediate results, which can be proved based on the definitions of $isAtomic1vBlk$ and $isAtomic2vBlk$: (i) t cannot read any write of t' to any variable; (ii) if t' reads some write of t in S, t' reads the same write in S'; (iii) for each variable y accessed in both t and t', if there are writes to y in both t and t', $e^{t'}_{FW(y)}$ must occur after $e^{t}_{FW(y)}$ in S.

Case 2: Suppose no variable is written by both transactions, and at least one transaction contains a write. Without loss of generality, suppose t contains a write e^{t}_{W}. If some read in t' reads the value written by e^{t}_{W}, then we can show that S is equivalent to the serial schedule in which t precedes t'; otherwise, we can show that S is equivalent to the serial schedule in which t' precedes t.

Case 3: If neither transaction contains a write, then S is trivially serializable. □

Let E and P be defined as in Section 5.1. The total number of 2v-blocks is $O(E^2)$. The algorithm based on Theorem 5.2.1 considers all pairs of 2v-blocks, so, assuming $|locksHeld(t)|$ is always bounded by a constant for all threads t, its worst-case running time is $O(PE^4)$.

5.3 Multiple Transactions That Share Multiple Variables

In the presence of multiple shared variables, a set T of transactions is not necessarily atomic even if all subsets of T with cardinality two are atomic. This is due to cyclic dependencies. For example, consider the following trace containing three transactions (time increases from left to right):

$$\begin{array}{lccccc} t_1: & W(x) & & & & W(y) \\ t_2: & & R(x) & W(z) & & \\ t_3: & & & & R(z) & R(y) \end{array} \tag{5.6}$$

In any potential serial trace equivalent to this one, t_1 must precede t_2, t_2 must precede t_3, and t_3 must precede t_1 because $t_2.R(x)$ reads the written value by $t_1.W(x)$, $t_3.R(z)$ reads the written value by $t_2.W(z)$, and $t_3.R(y)$ is an initial read. Due to the cyclic dependency, no equivalent serial trace exists. Therefore, $\{t_1, t_2, t_3\}$ is not atomic, even though all three subsets of T with cardinality two are atomic.

Cyclic dependencies between transactions that are pairwise atomic arise from dependencies involving initial reads and final writes. This observation motivates the following extension of Theorem 5.2.1. Let *IR-FW*(T) denote the set of transactions obtained from T by discarding all events other than synchronization events and initial reads and final writes on shared variables.

Theorem 5.3.1. *Let T be a set of transactions. Suppose T does not have potential for deadlock. T is atomic iff for all t and $t' \in T$ with $thread(t) \neq thread(t')$, (i) $\forall\, b \in$ 1v-blocks(t). $\forall\, b' \in$ 1v-blocks(t'). $isAtomic1vBlk(b, b')$; and (ii) $\forall\, tr \in traces($IR-FW$(T))$. tr is serializable.*

Proof. For the forward implication ($\Rightarrow$), the proof is straightforward. For the reverse implication ($\Leftarrow$), we need to prove that there is an equivalent serial trace S' for each trace S of all events in T. Condition (ii) of this theorem implies Condition (ii) of Theorem 5.2.1. Hence, for all t, $t' \in T$, $\{t, t'\}$ is atomic according to Theorem 5.2.1. Thus, only the sequence of initial reads and final writes of t and t' in S affects their possible order in S'. Condition (*ii*) implies there is a serial trace S'' equivalent to *IR-FW*(S). Therefore, S' can be obtained by concatenating the transactions in T in the same order that they appear in S''. □

This algorithm is relatively expensive, because the number of possible traces may be large. On the positive side, this algorithm considers only traces for *IR-FW*(T), and hence may be significantly faster than the naive algorithm that considers all traces for T.

The worst-case time complexity of the algorithm based on Theorem 5.3.1 is exponential in the number of events. This is not surprising, because similar problems, such as determining serializability of a given trace, are NP-complete [38].

5.4 Usage and Comparison of The Three Block-Based Algorithms

The algorithms based on Theorems 5.1.2 and 5.2.1 can be applied to arbitrary executions. For Theorem 5.1.2, this simply means considering one shared variable at a time, *i.e.*, applying the algorithm independently to each shared variable. For Theorem 5.2.1, this means considering the transactions pairwise, and not checking atomicity of larger sets of transactions. Taking this perspective, we have three block-based algorithms that range from a relatively cheap one that detects limited but common kinds of atomicity violations to a relatively expensive one that also detects complex but rare kinds of atomicity violations. Based on the experiments in Chapter 7, we believe that in practice, the algorithms based on Theorems 5.1.2 and 5.2.1 reflect better trade-offs between cost and defect-finding effectiveness. Indeed, in those experiments, there is no atomicity violation that would be reported based on Theorem 5.3.1 and not reported based on Theorem 5.2.1, because the cyclic dependencies between three or more transactions that could cause such warnings never appear in those experiments.

5.5 Comparison of Reduction-based Algorithm and Block-based Algorithm

The block-based algorithm is more expensive than the reduction-based algorithm, but more accurate, according to the experimental results in Chapter 7. For a small example of this, consider the threads t_2, t_3 and t_4 in Figure 4.2. Only x is shared, so the algorithm in Section 5.1 suffices. The 1v-blocks are $\langle x, R, dummy, false, false, pid_2, \{o_2\}, \{\}, \{\}\rangle$, $\langle x, R, dummy, false, false, pid_3, \{o_1\}, \{\}, \{\}\rangle$, and $\langle x, R, W, false, true, pid_4, \{\}, \{o_1, o_2\}, \{\}\rangle$, where pid_i is the ID of the thread period containing the events of transaction t_i. The block-based algorithm shows that $\{t_2, t_3, t_4\}$ is atomic. Recall from Section 4.5 that the reduction-based algorithm reports a false alarm for this example.

5.6 Dynamic Construction of Blocks

We construct 1v-blocks incrementally during execution of a transaction; this avoids storing all events in the transaction until its end.

2v-blocks are constructed when the transaction finishes; this requires storing only initial reads and final writes until the end of the transaction. As an optimization, if two initial reads e_1 and e_2 in a transaction operate on the same variable, and $held(e_1) = held(e_2)$, and $heldmid(e_1, e_2) = \emptyset$, then one of them can be discarded without affecting the result.

To avoid constructing redundant blocks, the most recent several event patterns are cached. When an event pattern in the cache appears again, we do not construct block

for it again. This optimization saves times, because constructing blocks is more expensive than a cache lookup.

The same block could appear in many transactions. We save space by sharing blocks among multiple transactions.

Chapter 6

Runtime Commit-Node Algorithm

The commit-node algorithm [53] has two versions: one is for checking conflict-atomicity; the other is for checking view-atomicity. Conflict-atomicity and view-atomicity are defined in Chapter 2. The algorithm is off-line, *i.e.*, when the program terminates, it is applied to recorded information about the execution. The execution is partitioned into units. Recall from Chapter 2 that a *unit* is a sequence of consecutive events executed by a single thread, and a *transaction* is a unit expected to behave atomically. The commit-node algorithm checks whether every trace (*i.e.*, every feasible interleaving) of these units is equivalent to a serial trace, *i.e.*, a trace in which each transaction is a contiguous subsequence. If so, we say that the transactions are *atomic*; if not, a potential atomicity violation is reported.

When a program runs, the events for each of its units (including transactions) is saved in a tree structure, called an *access tree*. Each node in an access tree denotes an access to an escaped variable (*i.e.*, a variable accessible to multiple threads), or a synchronization operation (*e.g.*, lock acquire and release). After the program terminates, the relationships between nodes in different trees are analyzed, and *inter-edges* are added between them to generate a *forest*. A node with an incident inter-edge is called a *communication node*. A communication node is called a *commit node* if none of its descendants are communication nodes. In a forest, if the access tree for each transaction has only one commit node, then the transactions are atomic. Intuitively, the commit-node is like a commit point [26]. By considering the synchronization, the commit-node algorithm does not merely look for violations of atomicity in the observed execution, but also attempts to determine whether the non-determinism of thread scheduling could allow violations in other executions.

Experiments show that the commit-node algorithm is more efficient in most experiments and more accurate than previous algorithms with comparable asymptotic complexity.

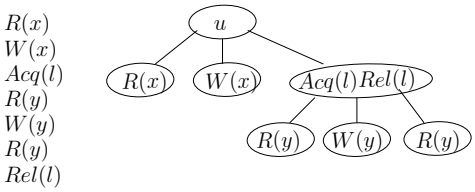

Figure 6.1: The access tree for a unit u. All events are shown on the left; the order of sequence is from top to bottom.

6.1 Data Structures

6.1.1 Access Tree

During execution of the instrumented program, all events for each unit is recorded into an access tree. In such a tree, each leaf node is called an *access node* and denotes an access to an escaped variable. Each non-leaf node except for the root is called a *synchronization node* and denotes a synchronization block. The root node denotes the whole unit. The local orders of events within a unit are denoted by the order of branches in the tree. An example appears in Figure 6.1, where $R(v)$ and $W(v)$ (v is x or y) denote a read event and a write event to v, respectively; $acq(l)$ and $rel(l)$ denote an acquire and a release of lock l, respectively. Since each node in an access tree denotes a set of events, a node and the set of events it denotes are used interchangeably in our description.

6.1.2 Access Forest

An *access forest* consists of a set of access trees and edges called *inter-edges* between access trees from concurrent units. Section 3.2 describes a happen-before analysis to determine whether two units are concurrent. The edges inside each tree are called *tree-edges*. Nodes with an incident inter-edge are called *communication nodes*; they denote a potential interactions between the corresponding units. Checking conflict-atomicity and view-atomicity require different inter-edges. The access forest used for checking conflict-atomicity is called *conflict-forest*; the access forest used for checking view-atomicity is called *view-forest*.

6.1.3 Conflict-Forest

In the conflict-forest, there are two kinds of relationships denoted by inter-edges between two concurrent units. The first kind of relationship is between a node associated with a write in one of the units and a concurrent node associated with a read to the same variable in the other unit, if the read can read the written value by the write in some trace. The second kind of relationship connects two concurrent nodes associated with two writes to the same variable in the two.

```
FOR each read event e_x^r
  FOR each write event e_x^w in a concurrent unit
    addInterEdge(e_x^r, e_x^w);

FOR each write event e_x^w
  FOR each write event e_x^w' in a concurrent unit
    addInterEdge(e_x^w, e_x^w');

/* add appropriate inter-edges between nodes of the units containing e and e'.
e and e' access the same variable.*/
PROCEDURE addInterEdge(e, e'){
  IF (held(e) ∩ held(e') = ∅) {
    add an inter-edge between the access node for e and the access node for e';
  } ELSE {
    /* there must be a common lock in held(e) and held(e'),
    so the next statement finds a suitable node n.*/
    starting at the root node of the unit that contains e, go down the tree along the path to e,
    until reaching a node n corresponding to a synchronization block for a lock l in held(e');
    IF ((e is a write) ∨ (e is read and not preceded by a write
        to the same variable in the subtree rooted at n)) {
      /* otherwise, e is a read and there is a write to the same variable in the subtree
      rooted at n, so e cannot read the write of e' because of lock l.*/
      n' = the outermost ancestor of e' corresponding to a synchronization block for lock l;
      add an inter-edge between n and n';
}}}
```

Figure 6.2: The algorithm to add inter-edges for an arbitrary escaped variable x in the conflict-forest.

We say "associated with" above because the inter-edge is not necessarily added directly between the access nodes representing those two accesses. Instead, for each pair of accesses satisfying the above conditions, if there is at least one lock that is held when both operations are performed, then we find the outermost of those common locks, and add an inter-edge between the corresponding synchronization nodes, because this is the granularity at which the parts of the units containing those accesses can be interleaved; if no such lock exists, then an inter-edge is added directly between the access nodes representing those two accesses. By assumption, the set of units does not have potential for deadlock, so the notion of outermost common lock for two accesses is well defined; if there is potential for deadlock, the threads that execute the two accesses could acquire two locks in different orders without first acquiring a common lock.

Recall the definitions in Chapter 2, T is a set of transactions, and E is a set of non-transactional units. Intuitively, $\langle T, E\rangle$ is conflict-atomic, if in all traces of $\langle T, E\rangle$, the events of each transaction of T can be repeatedly swapped with adjacent events without affecting the rest of the trace, until the trace is serial, *i.e.*, the events of each transaction are

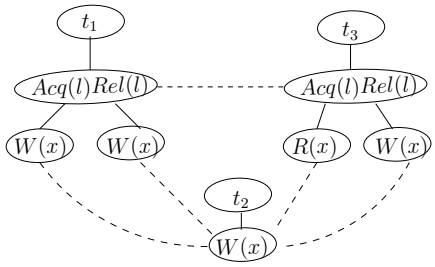

Figure 6.3: A conflict-forest. The inter-edges are shown as dotted lines.

contiguous. If two nodes are connected by an inter-edge, they cannot be swapped. Thus, a node with incident inter-edges is like a non-mover in Lipton's reduction [31, 15, 51, 54].

Figure 6.2 shows the algorithm to add inter-edges. Figure 6.3 shows the conflict forest for a set of three units. Note that an inter-edge can denote multiple relationships of the kinds described above. For example, the inter-edge between t_1 and t_3 in Figure 6.3 denotes two relationships: one is that $t_3.R(x)$ can read the value written by $t_1.W(x)$, and the other is between $t_1.W(x)$ and $t_3.W(x)$.

Besides checking atomicity, the conflict-forest can also be used for detecting data races, since each access node with incident inter-edges indicates a data race.

6.1.4 View-Forest

The view-forest has three kinds of relationships between two concurrent units u_1 and u_2 denoted by inter-edges. (1) The first kind of relationship is between a node of u_1 associated with a write and a node of u_2 associated with a read, if the read can read the written value by the write in some trace. (2) The second kind of relationship connects two nodes associated with two writes to the same variable, respectively, if both writes can be the write-predecessor of the same read in some traces. (3) The third kind of relationship connects two nodes associated with unit-final writes to the same variable.

The algorithm of adding inter-edges for view-forest is shown in Figure 6.4. It is similar to the algorithm in Figure 6.2. When adding an inter-edge between a read and its potential write-predecessor, we also add inter-edges between its all potential write-predecessors. $S(e)$ caches all potential write-predecessors for the read e so far. Besides connecting this kind of writes, we also add inter-edges between the unit-final writes to the same variables, instead of adding inter-edges between every two writes to the same variables in conflict forest.

Figure 6.5 shows the view forest after applying the algorithm to the same three units in Figure 6.3.

```
FOR each read event e_x^r
  S(e_x^r) = ∅;
  FOR each write event e_x^w in a concurrent unit
    addInterEdge(e_x^r, e_x^w);

FOR each unit-final write event e_x^w
  FOR each unit-final write event e_x^w' in a concurrent unit
    addInterEdge(e_x^w, e_x^w');

/* Note that e and e' access the same variable. */
PROCEDURE addInterEdge(e, e'){
  IF (held(e) ∩ held(e') = ∅){
    add an inter-edge between the access node for e and the access node for e';
  } ELSE {
    starting at the root node of the unit that contains e, go down the tree along the path to e,
    until reaching a node n corresponding to a synchronization block for a lock l in held(e');
    IF ((e is a write) ∨ (e is read and not preceded by a write
        to the same variable in the subtree rooted at n)) {
      n' = the outermost ancestor of e' corresponding to a synchronization block for lock l;
      add an inter-edge between n and n';
    } ELSE return;
  }
  IF (e is read) {
    e_w = the preceding write to the same variable and in the same unit as e, if any, otherwise null;
    IF (e_w ≠ null)
      S(e) = S(e) ∪ {e_w};
    FOR each e'' in S(e)
      addInterEdge(e',e'');
    S(e) = S(e) ∪ {e'};
}}
```

Figure 6.4: The algorithm to add inter-edges for an arbitrary escaped variable x in the view-forest.

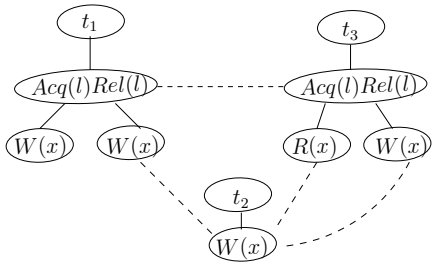

Figure 6.5: A view-forest. The inter-edges are shown as dotted lines.

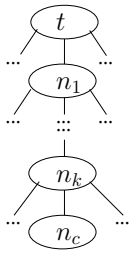

Figure 6.6: A transaction which contains a single commit node n_c.

6.2 Commit-Node Reduction

When a communication node n is an ancestor of another communication node n', we call that n *contains* n'. A communication node is called a *commit node* if it is not contained in any other communication nodes. For example, in Figure 6.5, the communication node "$Acq(l)Rel(l)$" in t_1 contains the communication node "$W(x)$" (*i.e.*, the second write) which is a commit node since it does not contain any other communication nodes.

The intuition for commit-node is as follows. Suppose there is a transaction t as shown in Figure 6.6, where the only one commit is n_c. n_c may be an access node or synchronization node. All synchronization nodes except for the nodes on the path from the root node t to n_c are not communication nodes. Thus, similar to the reduction-based algorithm, we may find a commit point inside n_c, all events in n_c can be moved to the commit point position through swapping without affecting the other units in all traces. All other events in t can be done in the same way. Hence, a transaction with at most one commit node is atomic, but a transaction with two or more commit nodes might be non-atomic. This is described formally in Section 6.3.

6.3 The Commit-Node Algorithm for Checking Atomicity

This section presents the commit-node algorithm for checking conflict-atomicity and view-atomicity.

6.3.1 Conflict-Atomicity

Theorem 6.3.1. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If each transaction of T has at most one commit node in the conflict-forest, then $\langle T, E\rangle$ is conflict-atomic.*

Proof. To prove that $\langle T, E\rangle$ is conflict-atomic, we need to show that there is a conflict-equivalent serial trace tr' for an arbitrary trace tr of $\langle T, E\rangle$. The general idea is to find a location of some event inside the commit node for each transaction, such that when all events

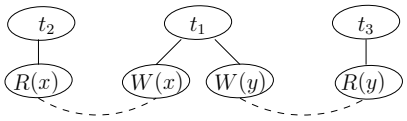

Figure 6.7: $\langle\{t_1, t_2, t_3\}, \emptyset\rangle$ is both conflict-atomic and view-atomic, but t_1 contains two commit nodes.

of each transaction are moved to that location, the resulting trace is conflict-equivalent to the original trace. That location is called a *commit point*.

For a commit node n, if n is an access node, its commit point is the location of the access event at tr; if n is a synchronization node, its commit point is any arbitrary location inside the commit node at tr. According to the assumption in the theorem, each communication node contains only one commit node. tr' is constructed from tr as follows: all events of the communication nodes that contain the commit node are moved to the commit point; all events of each transaction not in any communication node are also moved to the commit point of the transaction; all other events are not moved. tr' is serial because every transaction has only one commit node. In the following, we prove that tr' is a legal trace and is conflict-equivalent to tr.

First, we observe that tr' is consistent with the synchronization events. This holds because tr' is serial, and E does not contain any synchronization. So tr' is a trace for $\langle T, E\rangle$.

Next, we show that tr' is conflict-equivalent to tr. Consider conflicting events e_1 and e_2, where e_1 and e_2 occur in units u_1 and u_2 in $T \cup E$, respectively. Without loss of generality, suppose e_1 precedes e_2 in tr. Because e_1 and e_2 conflict, u_1 must contain a communication node n_1 containing e_1, and u_2 must contain a communication node n_2 containing e_2, and n_1 precedes n_2 in tr. After moving e_1 and e_2 to the commit points of u_1 and u_2, respectively, e_1 and e_2 appear at the same order in tr and tr'. Therefore, tr is conflict-equivalent to tr'. □

The condition in Theorem 6.3.1 for conflict-atomicity is sufficient but not necessary. In Figure 6.7, the set of transactions is conflict-atomic, even though t_1 contains multiple commit nodes. The following theorem shows that the condition in Theorem 6.3.1 is an exact test for conflict-atomicity of two transactions.

Let *held-outside*(n) denote the locks held by the executing thread just before the thread executes the first event of node n. Let *held-outside*(n_1, n_2) denote the locks acquired before the first event node n_1 and released after the last event of node n_2 by the executing thread. Let *held-mid*(n_1, n_2) denote the locks acquired and released between the last event of n_1 and the first event of n_2 by the executing thread.

Theorem 6.3.2. *Suppose $\langle T, \emptyset\rangle$ has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset\rangle$ is conflict-atomic iff each transaction in T has at most one commit node in the conflict-forest.*

Proof. "⇐": it follows from Theorem 6.3.1.

"⇒": Suppose $T = \{t, t'\}$. We show that $\langle T, \emptyset \rangle$ is not conflict-atomic if at least one transaction in T has two or more commit nodes. Without loss of generality, suppose t has at least two commit nodes. Let n_1 and n_2 denote two commit nodes of t. Thus, t' has at least one commit node. Otherwise, t cannot have any commit node. There must be a pair of conflicting events, denoted e_1 and e'_1, $e_1 \in n_1$ and $e'_1 \in n'_1$, where n'_1 is a communication node in t', and there is an inter-edge between n_1 and n'_1. Similarly, there must be another pair of conflicting events, denoted e_2 and e'_2, $e_2 \in n_2$ and $e'_2 \in n'_2$, where n'_2 is a communication node in t', and there is an inter-edge between n_2 and n'_2.

Suppose one of n'_1 and n'_2 contains the other or $n'_1 = n'_2$. Then there is a trace tr where n'_1 and n'_2 happen between n_1 and n_2, because *held-outside*$(n_i) \cap$ *held-outside*(n') = $\emptyset$ for $i = 1, 2$, where n' is the outer of n'_1 and n'_2, or $n' = n'_1$ if $n'_1 = n'_2$. Thus, tr does not have any conflict-equivalent serial trace.

Suppose n'_1 does not contain n'_2 and vice versa. According to the definition of conflict-forest, we know *held-outside*$(n_1) \cap$ *held-outside*$(n'_1) = \emptyset$ and *held-outside*$(n_2) \cap$ *held-outside*$(n'_2) = \emptyset$. Furthermore, because there is no potential for deadlock, *held-mid*(n'_1, n'_2) $\cap$ *held-outside*(n_1, n_2) = $\emptyset$ or *held-mid*$(n_1, n_2) \cap$ *held-outside*(n'_1, n'_2) = $\emptyset$. Otherwise, if there exists a lock $l_1 \in$ *held-mid*$(n'_1, n'_2) \cap$ *held-outside*(n_1, n_2) and a lock $l_2 \in$ *held-mid*$(n_1, n_2) \cap$ *held-outside*(n'_1, n'_2), then t acquires l_2 while holding l_1, and t' acquires l_1 while holding l_2, and there is no outer lock to prevent those acquires from being interleaved, so these would be potential for deadlocks. Thus, there is a trace tr where both n'_1 and n'_2 happen between n_1 and n_2, or n_1 and n_2 happen between n'_1 and n'_2. As in the previous case, tr does not have any conflict-equivalent serial trace. □

For example, in Figure 6.3, $\langle \{t_1, t_2\}, \emptyset \rangle$ is not conflict-atomic according to Theorem 6.3.2 because t_1 contains two commit nodes when ignoring t_3. Similarly, $\langle \{t_2, t_3\}, \emptyset \rangle$ is not conflict-atomic, either.

The following theorem gives a more accurate and expensive (compared to Theorem 6.3.1) condition to decide conflict-atomicity (for any number of transactions). This theorem (unlike Theorem 6.3.1) is accurate enough to show that the set of transactions in Figure 6.7 is conflict-atomic. Note that, when considering cycles in the conflict-forest, tree edges are treated as undirected edges.

Theorem 6.3.3. *Suppose $\langle T, E \rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If all pairs (if they exist) of communication nodes from the same transaction that do not contain each other are not involved in any cycle of the conflict-forest, then $\langle T, E \rangle$ is conflict-atomic.*

Proof. We prove the contrapositive. Suppose $\langle T, E \rangle$ is not conflict-atomic. Thus, there is a trace tr for $\langle T, \emptyset \rangle$ that does not have any conflict-equivalent serial trace, and there is a directed cycle c in the serialization graph for tr. Suppose that c consists of $\langle t_1, t_2, ..., t_n \rangle$ in order, where $t_1, t_2, ..., t_n \in T$. c implies that there must be an event e_1 of t_1 that happens before an event e_2 of t_2, an event e'_2 of t_2 that happens before an event e_3 of t_3, ..., and

```
Instrument the source code of program to be tested as discussed in Section 7.2;
Execute the instrumented program, and dynamically construct the conflict-trees;
After the program terminates, construct conflict-forest by adding inter-edges
between conflict-trees.
If (there is potential for deadlocks)
   Return "cannot decide";
/* Applying Theorem 6.3.1 */
If (every transaction has at most one commit node)
   return "conflict atomic";
/* Applying Theorem 6.3.3 */
if (for every transaction, all pairs of communication nodes from the transaction that
   do not contain each other are not involved in any cycle of the conflict-forest)
   return "conflict atomic";
else /* a conservative condition */
   Return "not conflict-atomic";
```

Figure 6.8: The commit-node algorithm for checking conflict-atomicity.

an event e'_n of t_n that happens before e'_1 of t_1 in tr, where e_1 and e_2 access the same variable, e'_2 and e_3 access the same variable, ..., and e'_n and e'_1 access the same variable. This implies that there is a cycle c' in the conflict-forest. If e_i and e'_i of t_i for $i = 1..n$ are in the same communication node, then both e_i and e'_i happen before e_{i+1} and e'_{i+1} in tr, for $i = 1..n-1$. This contradicts the assumption that e'_n happens before e'_1. Hence, there must be a transaction in $\{t_1, t_2, ..., t_n\}$ that has at least two communication nodes on c' that do not contain each other. □

The *commit-node algorithm* for checking conflict-atomicity is shown in Figure 6.8.

Although this algorithm may report false alarms since the conditions in Theorem 6.3.1 and Theorem 6.3.3 are sufficient but not necessary. But we believe that this happens very rarely. In the experiments of Section 7.4, all the warnings for non-conflict-atomicity reported by the algorithm are confirmed to be true by Theorem 6.3.2.

Let $|T|$, n_t, and n_e denote the number of transactions, the maximum number of events in a transaction, and the number of events in the whole execution (including non-transactional units), respectively. Theorem 6.3.3 requires checking, for each pair of communication nodes of the same transaction, whether they are involved in a cycle, *i.e.*, whether each of them is reachable from the other. There are $O(|T| \times n_t^2)$ such pairs, and checking whether two nodes are reachable from each other takes time $O(n_e^2)$, so the worst-case time complexity of the algorithm is $O(|T| \times n_t^2 \times n_e^2)$.

6.3.2 View-Atomicity

Theorem 6.3.4. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If each transaction of T has at most one commit node in the view-forest, then $\langle T, E\rangle$ is view-atomic.*

Proof. For any trace tr of $\langle T, E\rangle$, we prove that it has a view-equivalent serial trace tr', which is constructed in the same way as in Theorem 6.3.1. The definition for the commit point of each transaction is the same as in Theorem 6.3.1. By the same reasoning as in the proof of Theorem 6.3.1, tr' is serial and consistent with the synchronization operations, so tr' is a trace for $\langle T, E\rangle$.

1. We prove that each read has the same write-predecessor in tr and tr'. Consider an arbitrary read e_x^r of a unit $u \in T \cup E$. If e_x^r does not have potential write-predecessors in units other than u, the e_x^r has the same write-predecessor in tr and tr' because tr' preserves the internal order of events for each unit in $T \cup E$.

Suppose e_x^r has potential write-predecessors in units other than u, consider two potential write-predecessors $e_x^{w_1}$ and $e_x^{w_2}$, and suppose without loss of generality that $e_x^{w_1}$ is the write-predecessor of e_x^r in tr. Note that $e_x^{w_1}$, $e_x^{w_2}$ and e_x^r cannot all be in the same unit. We consider four cases.

(1) If $e_x^{w_1}$, $e_x^{w_2}$ and e_x^r belong to different units, then there are communication nodes such that $e_x^r \in n_r^{w_1}$, $e_x^r \in n_r^{w_2}$, $e_x^{w_1} \in n_{w_1}^r$, $e_x^{w_1} \in n_{w_1}^{w_2}$, $e_x^{w_2} \in n_{w_2}^r$, $e_x^{w_2} \in n_{w_2}^{w_1}$, and such that inter-edges exist between $n_r^{w_1}$ and $n_{w_1}^r$, between $n_r^{w_2}$ and $n_{w_2}^r$, and between $n_{w_1}^{w_2}$ and $n_{w_2}^{w_1}$. The inter-edge implies that $n_{w_1}^{w_2}$ and $n_{w_2}^{w_1}$ cannot interleave, so $n_{w_2}^{w_1}$ happens either before $n_{w_1}^{w_2}$ or after $n_r^{w_2}$ in tr (if $n_{w_2}^{w_1}$ happened between $n_{w_1}^{w_2}$ and $n_r^{w_2}$, it would contradict the fact that $e_x^{w_1}$ is the write-predecessor of e_x^r in tr). Thus, the commit point of $n_{w_2}^{w_1}$ is either before the commit point of $n_{w_1}^{w_2}$ or after the commit point of $n_r^{w_2}$ in tr. Therefore, $e_x^{w_2}$ happens either before $e_x^{w_1}$ or after e_x^r in tr', so e_x^r has the same write-predecessor $e_x^{w_1}$ in both tr and tr'.

(2). If $e_x^{w_1}$ and $e_x^{w_2}$ belong to the same unit, then $e_x^{w_2}$ must happen either before $e_x^{w_1}$ or after e_x^r in tr. For the first case, $e_x^{w_2}$ also happens before $e_x^{w_1}$ in tr'. For the second case, there must be communication nodes $n_{w_2}^r$ and $n_r^{w_2}$ that contain $e_x^{w_2}$ and e_x^r, respectively, and $n_{w_2}^r$ happens after $n_r^{w_2}$ in tr. Thus, $e_x^{w_2}$ also happens after e_x^r in tr. Therefore, e_x^r has the same write-predecessor $e_x^{w_1}$ in tr and tr'.

(3). If $e_x^{w_1}$ and e_x^r belong to the same unit, then $e_x^{w_2}$ must happen either before $e_x^{w_1}$ or after e_x^r in tr. For the first case, there must be communication nodes $n_{w_2}^{w_1}$ and $n_{w_1}^{w_2}$ that contain $e_x^{w_2}$ and $e_x^{w_1}$, respectively, and $n_{w_2}^{w_1}$ happens before $n_{w_1}^{w_2}$ in tr, so $e_x^{w_2}$ also happens before $e_x^{w_1}$ in tr'. For the second case, $e_x^{w_2}$ happens after e_x^r in tr' by the same reason. Therefore, e_x^r has the same write-predecessor $e_x^{w_1}$ in tr and tr'.

(4). If $e_x^{w_2}$ and e_x^r belong to the same unit, then $e_x^{w_2}$ must happen before $e_x^{w_1}$ in tr, and there must be communication nodes $n_{w_2}^{w_1}$ and $n_{w_1}^{w_2}$ that contain $e_x^{w_2}$ and $e_x^{w_1}$, respectively, and $n_{w_2}^{w_1}$ happens before $n_{w_1}^{w_2}$ in tr. This implies $e_x^{w_2}$ also happens before $e_x^{w_1}$ in tr'. Therefore, e_x^r has the same write-predecessor $e_x^{w_1}$ in both tr and tr'.

2. Now, we prove that tr and tr' have the same trace-final write to each variable. Suppose that the trace-final write to a variable x in tr is e_x^{fw}, and another write e_x^w to x happens before e_x^{fw} in tr. We prove that e_x^w also happens before e_x^{fw} in tr. Let $e_x^{w'}$ denote the last write to x in the same unit as e_x^w. Because e_x^{fw} is the trace-final write of tr, $e_x^{w'}$ happens before e_x^{fw} in tr. According to the algorithm in Figure 6.4, there must be two communication nodes n and n' such that $e_x^{fw} \in n$ and $e_x^{w'} \in n'$. Thus, n' must happen

before n. Hence, $e_x^{w'}$ happens before e_x^{fw} in tr'. Since e_x^w can either be $e_x^{w'}$ or happen before $e_x^{w'}$, e_x^w must happen before e_x^{fw}. □

For example, in Figure 6.5, $\langle\{t_1, t_2\}, \emptyset\rangle$ is view-atomic because each of t_1 and t_2 contains only one commit node in the view-forest. Note that $\langle\{t_1, t_2\}, \emptyset\rangle$ is not conflict-atomic since t_1 has two commit nodes in the conflict-forest.

The condition in Theorem 6.3.4 for view-atomicity is sufficient but not necessary. In the example of Figure 6.7, $\langle\{t_1, t_2, t_3\}, \emptyset\rangle$ is view-atomic but t_1 contains multiple commit nodes. The following theorem shows that the condition in Theorem 6.3.4 is an exact test for view-atomicity of two transactions.

Theorem 6.3.5. *Suppose T has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset\rangle$ is view-atomic iff each transaction in T has at most one commit node in the view-forest.*

Proof. "$\Leftarrow$": This implication is justified directly based on Theorem 6.3.4.

"$\Rightarrow$": We prove the contrapositive, *i.e.*, if at least one of the transactions has at least two commit nodes, then $\langle T, \emptyset\rangle$ is not view-atomic. According to the definition of view-forest, there are three kinds of inter-edge. (1) The first kind of inter-edge denotes the relationship between a read e^r in a transaction t and its potential write-predecessor e^w in the other transaction. If the read has a preceding write e_{pre}^w in its own transaction, then a violation of view-atomicity is possible (because e^w can occur between e_{pre}^w and e^r by the definition of potential write-predecessor), so the desired implication holds. If the read does not have any preceding write in its own transaction, the read and its potential write-predecessor in the other transaction act like a pair of conflicting events, in the sense that their order in a trace determines that the two transactions must follow the order in all serial traces view-equivalent to the trace (this is true with two transactions, although it is not true with more transactions). This is the same property of inter-edges in the conflict-forest that is used in the proof of Theorem 6.3.2. (2) The second kind of inter-edge denotes the relationship between two writes that are potential write-predecessor for the same read; this indicates a violation of view-atomicity (because either there is a write to some variable x in t' that can occur between a write to x and a read to x in t, or there is a read to x in t' that can occur between two writes to x in t), so the desired implication holds. (3) The third kind of inter-edge denotes the relationship between unit-final writes of each transaction. With two transactions, these final writes also act like conflicting events, in the sense described above, so these edges have the same property as inter-edges in the conflict-forest.

Thus, depending on the kind of edges present, either we immediately conclude that the desired implication holds, or all of the edges have the property of inter-edges in the conflict-forest used in the proof of ($\Rightarrow$) in Theorem 6.3.2, and the rest of this proof is similar to that proof. □

For example, in Figure 6.5, $\langle\{t_2, t_3\}, \emptyset\rangle$ is not view-atomic because t_3 contains two commit nodes in the view-forest for t_2 and t_3. The following theorem gives a more sophisticated condition to check view-atomicity.

Theorem 6.3.6. *Suppose $\langle T, E\rangle$ has no potential for deadlock, and E does not contain any synchronization operations. If all pairs of communication nodes from the same transaction that do not contain each other are not involved in any cycle of the view-forest, then $\langle T, E\rangle$ is view-atomic.*

Proof. The proof is similar to the proof of Theorem 6.3.3. □

The commit-node algorithm for checking view-atomicity is similar to the algorithm for checking conflict-atomicity shown in Figure 6.8, except that the view-forest is constructed and checked based on Theorems 6.3.4 and 6.3.6. Similarly as before, the worst-case time complexity of the algorithm is $O(|T| \times n_t^2 \times n_e^2)$.

6.4 Comparison with Other Atomicity Checking Algorithms

6.4.1 Reduction-Based Algorithm

In comparison with the reduction-based algorithm presented in Section 4.4, the following theorem and example together show that Theorem 6.3.1 is more accurate than Theorem 4.4.2 for conflict-atomicity.

Theorem 6.4.1. *If a transaction t has the form $(R|AcqA^*Rel)^*N^?(L|AcqA^*Rel)^*$, then t has at most one commit node in the conflict-forest.*

Proof. According to the algorithm in Figure 6.2, the block denoted by $AcqA^*Rel$ does not contain any communication node. All synchronization blocks denoted by $R^*N^?L^*$ must be nested. Thus, for any two communication nodes in t that are also synchronization nodes, one is a descendant of the other. t contains at most one non-mover, and it occurs in the inner-most synchronization block. Thus, there is at most one access node in t that is also a communication node, and that communication node is a descendant of all other communication nodes in t. Thus, for any two communication nodes of transaction t, one is a descendant of the other. Hence, t has at most one commit node. □

Now we give an example that is conflict-atomic according to Theorem 6.3.1 but is wrongly reported to be non-atomic by the reduction theorem. Let $T = \{t_1, t_2\}$, where t_1 consists of $e_x^{r_1}$ followed by $e_x^{w_1}$, and t_2 consists of only $e_x^{r_2}$. t_1 has the form NN which does not match $(R|AcqA^*Rel)^*N^?(L|AcqA^*Rel)^*$, but t_1 and t_2 each have only one commit node.

The commit-node algorithm of Section 6.3.1 contains the benefits of all the improvements to the reduction-based algorithm described in [54], which include the improvements proposed in [15]. For example, for re-entrant locks, thread-local locks, and protected locks, there is no inter-edge connected to the corresponding synchronization nodes.

Therefore, the reduction-based algorithm is less accurate than the commit-node algorithm of Section 6.3.1.

6.4.2 The Block-Based Algorithm

The commit-node algorithm and the block-based algorithm discussed in Section 5.3 are both exact tests for view-atomicity for two transactions, but the former runs much faster in most programs in the experiments in Chapter 7. For three or more transactions, the commit-node algorithm is an efficient conservative test that is very accurate in practice based on the experiments in Chapter 7; the block-based algorithm can provide an exact test but is significantly more expensive.

6.5 Implementation

We implemented the commit-node algorithm for checking conflict-atomicity and view-atomicity in Java. The implementation consists of three parts: instrumentation, monitoring and off-line analysis. Instrumentation is discussed in Section 7.2. The monitor intercepts all events described in Chapter 2 and constructs access trees. Each access tree is optimized to discard the redundant accesses, as discussed in Section 6.5.1. If there are more than two identical access trees, we save only two copies, since the rest are redundant for checking atomicity. A dynamic escape analysis introduced in Section 3.1 is used to determine when a variable escapes. A happen-before analysis introduced in Section 3.2 is used to determine whether two units are concurrent. When the program terminates, the algorithm adds inter-edges between access trees, and then checks conflict-atomicity and view-atomicity using the algorithms in Sections 6.3.1 and 6.3.2, respectively.

6.5.1 Optimization: Trimming the Access Tree

It is not necessary to save all accesses to escaped variables. For access nodes with the same parent node, we preserve only the first two read accesses and the first two write accesses (if they exist) to each escaped variable, because the first two reads and writes to x can represent all discarded accesses for checking (conflict or view) atomicity. The resulting trees and forests are said to be *trimmed*.

Theorem 6.5.1. *For every $\langle T, E\rangle$, and every hypothesis H about the conflict-forest and view-forest in the theorems of Section 6.3, H holds for the trimmed forest iff it holds for the untrimmed forest.*

Proof. 1. Consider reads. Let R be a set of three or more reads to the same variable that share the same parent node. It is easy to see that in both conflict forest and view forest, either all of them are connected to a given write in another unit by inter-edges, or none of them are connected to it. Suppose we evaluate H considering only the first two reads in

R. It is easy to show that considering additional reads in R either does not generate any additional inter-edges, or the generated edges do not affect H.

2. Consider writes in the conflict-forest. Let W be a set of three or more writes to the same variable that share the same parent node. In the conflict forest, either all of them are connected to a given read or write of another unit by inter-edges, or none of them are connected to it. Similarly as for reads, we can show that considering the third and subsequent writes in W either does not generate any additional inter-edges, or the generated edges do not affect H. For the view forest, if W does not contain a unit-final write, then the reasoning is similar to the previous cases; if W contains a unit-final write, it gets removed and the second write in W becomes the unit-final to that variable; it is easy to verify (for each hypothesis H) that this does not affect H. □

Chapter 7

Experiments and Related Work

7.1 Introduction

We implemented the reduction-based, block-based, commit-node algorithms in Java.

To facilitate comparison with [15], we use the same on-line reduction-based algorithm as in [15]; specifically, it uses the lockset algorithm from [49], ignores arrays, uses Theorem 4.3.1 instead of Theorem 4.4.2, and uses the improvements of Section 4.6. It does not use dynamic escape analysis or happen-before analysis.

The off-line reduction-based algorithm is based on Theorem 4.4.2, and incorporates the multi-lockset algorithm (which uses dynamic escape analysis and happen-before analysis) for checking data races, and the improvements of Section 4.6. Note that all these improvements could also be applied to the on-line reduction-based algorithm, but we did not do that, in order to compare our algorithms with [15].

The block-based algorithm in these experiments is based on Theorem 5.2.1. Theorem 5.3.1 is more precise but more expensive than Theorem 5.2.1. In addition, we implemented a check for the presence of cyclic dependencies between three or more transactions, and cyclic dependencies do not appear in these experiments. This implies that we did not miss any atomicity violations by using Theorem 5.2.1 instead of Theorem 5.3.1. We believe that Theorem 5.2.1 is sufficient for most programs.

The commit-node algorithm for conflict-atomicity is shown in Figure 6.8. It first checks the conditions of Theorem 6.3.1; if they are satisfied, reports that conflict-atomicity holds; otherwise, checks the conditions of Theorem 6.3.3, then reports conflict-atomicity holds or not according to whether the conditions are satisfied. The commit-node algorithm for view-atomicity is similar to the algorithm for checking conflict-atomicity, except that the view-forest is constructed and checked based on Theorems 6.3.4 and 6.3.6.

To evaluate the three runtime algorithms, we apply them to 12 programs. The 12 programs are `elevator`, `tsp`, `sor`, and `hedc` from [49]; `moldyn`, `montecarlo`, and `raytracer` from [28]; `StringBuffer`, `Vector`, `Hashtable`, and `Stack` from Sun JDK 1.4.2; and `jigsaw` [29]. `elevator` simulates the actions of two elevators. `tsp` solves the travelling salesman problem; we run it on the accompanying data files `map4`

and `map14`. `sor` is a scientific computing program which uses barriers rather than locks for synchronization. `hedc` is a Web crawler that searches astrophysics data on the Web. `moldyn`, `montecarlo`, and `raytracer` are computation-intensive parallel programs that compute molecular dynamics, Monte Carlo simulation, and ray tracing, respectively. `jigsaw` is a Web server implemented in Java. We instrument only its packages that are related with HTTP service. Table 7.4 shows the number of lines of code that are instrumented, *i.e.*, it excludes code in uninstrumented libraries. For all programs that accept the number of threads as an argument, we use three threads. All experiments are done on a Sun Blade 1500 with a 1GHz UltraSPARC III CPU, 2GB RAM, SunOS 5.8, and JDK 1.4.2*.

We modified `tsp` and `moldyn` slightly. Specifically, for `tsp`, we set `Tsp.MAX_NUM_TOURS` to 100 instead of 5000, and used instances of `Object()` as lock objects instead of instances of `Integer(0)`, since our system identifies locks by their identity hash code. For `moldyn`, we set `md.ITERS` to 1, moved some fields of `mdRunner` into its `run()` method so they became local variables, and marked instances of `particle` as unshared (*i.e.*, accessed by only one thread) and hence did not record accesses to them; this annotation makes the analysis faster and does not affect the result.

We designed test drivers for the classes `StringBuffer`, `Vector`, `Hashtable`, and `Stack`. The pseudo-code is shown in Figure 7.1, where C denotes one of these classes. The driver creates two instances o_1 and o_2 of C. For a pair $\langle m_1, m_2 \rangle$ of methods of C, the driver creates two threads t_1 and t_2, where t_1 executes $o_1.m_1$ and t_2 executes $o_2.m_2$. Each execution of `TestTwoMethods` is analyzed separately for the atomicity checking. When a method requires an instance of C as argument, the other instance is used. For example, if C is `Vector`, and m_1 is `addAll`, then thread t_1 executes o_1.`addAll`(o_2). The driver tests all pairs of methods such that m_1 and m_2 that do not both take an argument of type C; these excluded pairs would lead to potential for deadlock. For example, executing $\langle o_1.\texttt{addAll}(o_2), o_2.\texttt{removeAll}(o_1) \rangle$ may lead to deadlock because $o_1.\texttt{addAll}(o_2)$ locks o_1 then o_2, and $o_2.\texttt{removeAll}(o_1)$ locks o_2 then o_1. The driver does not test scenarios in which t_1 executes $o_1.m_1$ and t_2 executes $o_1.m_2$, because they would not produce any additional information, since these methods are synchronized.

In these experiments, we check atomicity of transactions defined by the defaults in Section 7.2. For arrays, every element is monitored, except that we use a cutoff of 3 at the beginning of arrays for `moldyn`, `montecarlo` and `raytracer`, and we use a cutoff of 10 in the middle of arrays for `sor` to catch more violations.

7.2 Instrumentation

This section describes the instrumentation of the source code.

We modify the pretty-printer in the Kopi [30] compiler to insert instrumentation as it pretty-prints the source code. The instrumentation intercepts the following events:

- reads and writes to all monitored fields (see below).

*All experiments of this dissertation are done in the same environment.

```
TestCollections()
  for (i=0;i<numMethods;i++)
    for (j=0;j<numMethods;j++)
      if (methods i and j of C do not both take args of type C)
        TestTwoMethods(i,j);

TestTwoMethods(int i, int j)
  C o1 = new C(), o2 = new C();
  start a new thread that executes the ith method of o1
        with o2 as a parameter if needed;
  start a new thread that executes the jth method of o2
        with o1 as a parameter if needed;
```

Figure 7.1: Pseudo code for test driver. `C` is `StringBuffer`, `Vector`, `Hashtable`, or `Stack`.

- entering and exiting synchronized blocks, including synchronized methods.
- entering and exiting methods that are considered as transactions (see below).
- calls to thread `start` and `join`.
- barrier synchronization.

The user specifies the classes to instrument as a list of expressions like `java.*` (denoting all classes in sub-packages of `java`), `java.util.*`, or `java.util.Vector`.

As introduced in Chapter 2, by default, executions of the following code fragments in the instrumented classes are considered to be transactions: non-private methods, synchronized private methods, and synchronized blocks inside non-synchronized private methods; as exceptions, the executions of the `main()` method in which the program starts and the executions of `run()` methods of classes that implement `Runnable` are not considered as transactions. These defaults are taken from [15]. We include synchronized blocks here because locks are often used to achieve atomicity. We include non-private methods here because they are abstractions often expected by clients of the class to behave as atomic operations. The defaults can be overridden using a configuration file, *e.g.*, the `run()` method of thread can be defined for atomicity checking. We did not override these defaults in any of the experiments. In addition, start, join and barrier operations are treated as transaction boundaries, as discussed in Section 3.2.

All non-final fields (with primitive type or reference type) of the specified classes are monitored. Accesses to these fields in all methods of all classes are instrumented, because even methods not considered as transactions by themselves might be invoked during a transaction. Local variables are not monitored, because they are necessarily thread-local. The defaults for monitoring non-final fields can also be overridden by a configuration file, *e.g.*, some fields can be defined for non-monitoring because they never escape.

In the reduction-based algorithm, for each monitored field, one or more locksets are maintained. In the block-based algorithm, for each monitored field, a previous event is cached to construct a block with the current event. We originally implemented these maps between monitored fields and their associated information as hash tables, with an object identifier combined with a field name as the key. This is relatively easy to implement but inefficient, since each access requires a look up in the hash table. Our current implementation inserts in each monitored class a new field (call it `shadow_f`) corresponding to each monitored field `f` of the class. `shadow_f` points directly to the information associated with `f`.

There is no way to insert fields into array classes in Java, so we use the less efficient approach described above to associate shadow information with arrays, *i.e.*, we maintain a hash table that maps each array reference a to shadow information a_s. Each array element has its own the shadow information. So a_s is an array with the same dimension and size as a. When an array is created, its associated shadow array is created. As an optimization, parts of the array can be allocated dynamically when needed.

Monitoring every array element causes large slowdown in some programs, so our system allows the user to specify a cutoff; for example, if the array is $[0..99] \times [0..99]$ and the cutoff is 3, then only the subarray $[0..2] \times [0..2]$ is monitored. Dynamic escape analysis is still carried out on the array and every element, regardless of the cutoff.

7.3 Usability

The block-based algorithm provides more detailed diagnostic information than the reduction-based algorithms (on-line and off-line). For example, Figure 7.2 shows part of the output of these algorithms for the `Vector` example in Figure 1.1. The reduction-based algorithms report an atomicity violation because of two consecutive synchronized blocks (*e.g.*, $R...L...R...L$, which does not match the patterns in Theorems 4.3.1 and 4.4.2, see Figure 4.3). The block-based algorithm reports an atomicity violation because it finds the second unserializable pattern described in Section 5.1, *i.e.*, for the field `elementCount` of some instance of `Vector`, a write to that field by some thread (denoted `Thread_2`) executing line 631 of `Vector.java` can occur between two reads of the same field by another thread (denoted `Thread_1`) executing lines 267 and 690 of `Vector.java`. The reduction-based algorithms have inherent limitation in reporting diagnostic information, because they cannot indicate which variables are involved in the atomicity violation (variables involved in data races can be identified, but there is no data race in this example), while the block-based algorithm indicates the specific fields and accesses that violate atomicity. The commit-node algorithm reports an atomicity violation because it finds that the transaction contains two commit nodes which correspond to the two synchronization blocks. The diagnostic information reported by the commit-node algorithm is similar to the reduction-based algorithm.

```
The reduction-based algorithms report:
  transaction Vector(Collection) is NOT atomic because :
    synchronized block @ Vector.java:689
    synchronized block @ Vector.java:266

The block-based algorithm reports:
  transaction Vector(Collection) is NOT atomic because :
  the unserializable pattern is
    VarName = Vector.elementCount
    Thread_1: R  @  Vector.java:267
    Thread_2: W  @  Vector.java:631
    Thread_1: R  @  Vector.java:690

The commit-node algorithm report:
  transaction Vector(Collection) is NOT atomic because :
  there are two commit nodes
    commit-node 1 @ Vector.java:689
    commit-node 2 @ Vector.java:266
```

Figure 7.2: Excerpts of diagnostic information for the `Vector` example.

7.4 Accuracy and Performance

Table 7.1 shows the running times and results of the three algorithms: on-line reduction-based algorithm and off-line reduction-based algorithm for conflict-atomicity, block-based algorithm and commit-node algorithm for view-atomicity. “Base time” is the running time of the uninstrumented program. “Intrcpt time” is the running time when all events relevant to atomicity checking are intercepted but not processed (an empty method is called). For each algorithm, “time” includes the running time of the instrumented program and the analysis. We classify warnings issued by each algorithm into three categories:

- *Bug*: the warning reflects a violation of atomicity that might cause a violation of an application-specific correctness requirement.
- *Benign*: the warning reflects a violation of atomicity that does not affect the correctness of the application.
- *False alarm*: the warning does not reflect a violation of atomicity.

Table 7.1 shows, for each category, the number of methods is issued such that a warning in that category for a transaction that is an execution of that method or part code of that method. If a transaction is correctly reported as not atomic, the corresponding method is counted only under bug or benign even if other warnings (which we do not need to classify) are also reported for that method. For a warning issued by the block-based algorithm, only the methods whose executions contribute two events in the unserializable patterns are

Program	Base	Intrcpt	On-line reduction		Off-line reduction		Block-based		Commit-node	
	time	time	time	report	time	report	time	report	time	report
elevator	0.2s	0.34s	0.2s	0-2-0-0	0.5s	0-2-0	0.6s	0-2-0	0.4s	0-2-0
tsp(map4)	0.24s	0.4s	0.5s	0-0-1-0	0.5s	0-0-1	0.5s	0-0-0	0.5s	0-0-0
tsp(map14)	0.24s	0.4s	31.7s	0-1-0-1	32.5s	0-2-0	8m59s	0-2-0	40.0s	0-2-0
sor	0.47s	47.1s	2.2s	0-0-2-0	53.3s	0-0-0	1m4.1s	0-0-0	52.4s	0-0-0
hedc	0.6s	0.82s	0.7s	0-0-2-0	1.0s	0-0-1	2.1s	0-0-0	1.0s	0-0-0
moldyn	44.03s	24m34s	9m49s	0-0-0-2	38m22.1s	0-0-0	28m54.6s	0-0-0	34m26s	0-0-0
montecarlo	15.85s	7m37s	1m43.3s	0-0-0-0	8m10.1s	0-0-0	8m11.4s	0-0-0	7m43s	0-0-0
raytracer	14.34s	10m8s	35m13.6s	1-0-0-1	11m58.9s	2-0-0	36m17.6s	2-0-0	10m50s	2-0-0
jigsaw	1.60s	2.2s	1.88s	1-2-1-1	2.74s	1-3-1	8m25.4s	1-3-0	3.4s	1-3-0
StringBuffer	-	-	-	0-1-0-0	-	0-1-0	-	0-1-0	-	0-1-0
Vector	-	-	-	4-2-0-12	-	4-4-10	-	4-4-0	-	4-4-0
Hashtable	-	-	-	0-2-3-0	-	0-2-3	-	0-2-0	-	0-2-0
Stack	-	-	-	3-2-0-14	-	3-4-12	-	3-4-0	-	3-4-0

Table 7.1: Performance and Accuracy. The four categories of "report" for the on-line reduction algorithm are bug - benign - false alarm - missed violation. The three categories of "report" for the other two algorithms are bug - benign - false alarm. A dash for "time" means that the running time is negligible.

counted. We aggregate the warnings in this way (instead of simply counting the number of warnings) to facilitate a fair comparison between the reduction-based and block-based algorithms. Note that the reduction-based algorithms and the commit-node algorithm always produce at most one warning per transaction (indicating that the patterns in Theorems 4.3.1 and 4.4.2 are not matched), while the block-based algorithm may produce multiple warnings per transaction, since multiple parts of the transaction may match the unserializable patterns in Chapters 5.1 and 5.2.

Table 7.1 also shows the number of missed errors for the on-line reduction algorithm, *i.e.*, the number of atomicity violations that are reported by the off-line reduction-based algorithm, but missed by the on-line reduction-based algorithm.

We conclude from Table 7.1 that:

(1) The on-line reduction-based algorithm actually misses some atomicity violations in practice, for the reasons mentioned in Chapters 4.5 and 4.6, and because it does not analyze arrays; this occurs for `tsp(map14)`, `moldyn`, `raytracer`, `jigsaw`, `Vector` and `Stack`. For example, in `raytracer`, the on-line reduction-based algorithm misses an atomicity violation because it mis-classifies some accesses to field `JGFRayTracerBench.checksum1` as race-free based on the information observed so far, whereas the off-line algorithm classifies them as races based on the entire execution. Another example is in `moldyn`, the algorithms that analyze arrays (off-line reduction-based and block-based) report atomicity violations involving arrays (these violations can be seen in Table 7.3), although these warnings are removed because of happen-before analysis (and hence are not evident in Table 7.1).

(2) The block-based algorithm is more accurate than the on-line and off-line reduction-based algorithms, in the sense that it reports fewer false alarms. The commit-node algorithm has the same accuracy on these benchmarks as the pairwise block-based algorithm.

The reduction-based algorithms issue false alarms because their analyses are imprecise, not because they are checking conflict-atomicity while the other algorithms used for Table 7.1 check view-atomicity.

(3) For most programs, the off-line reduction-based algorithm is slower than the on-line reduction-based algorithm, because the latter is on-line (this avoids storage overhead), uses less accurate and faster data race analysis, and ignores array accesses. The median slowdown of the off-line reduction-based algorithm compared to the on-line reduction-based algorithm is 46%. In `raytracer`, the off-line reduction-based algorithm is faster because it uses escape analysis (this point is explained Section 7.6).

(4) The block-based algorithm is slower than the off-line reduction-based algorithm; the median slowdown is 20%. The block-based algorithm is relatively much slower for `tsp(map14)` and `jigsaw`, because they execute a lot of code once (or a few times), producing many distinct blocks (see Table 7.4), while the other programs iterate more, producing more duplicate blocks.

(5) The commit-node algorithm is as fast as the reduction-based algorithm (even 0.4% faster on average), and significantly faster than the block-based algorithm (56% faster on average).

(6) Different input data affects the runtime analysis result for some programs, such as `tsp(map4)` and `tsp(map14)`, where `tsp(map14)` exercises more code than `tsp(map4)`.

We also check these programs for conflict-atomicity using the commit-node algorithm presented in Section 6.3.1. It issues exactly the same warnings (including bugs, benign and false alarms) as the commit-node algorithm for checking view-atomicity. The commit-node algorithm for conflict-atomicity is slightly faster (5.9% faster on average) than the commit-node algorithm for view-atomicity, because the former needs less time to construct inter-edges. We also test the programs by comparing pair of transactions each time according to Theorems 6.3.2 and 6.3.5. Checking pairs of transaction for conflict-atomicity and view-atomicity produces the same result as checking the whole set of transactions.

Please note that the performance is closely related to the implementation. If the instrumentation is done on byte code like [37], the performance can be improved significantly.

7.5 Report of Bugs

The bugs in `raytracer` come from atomicity violations involving the field `JGFRayTracerBench.checksum1`, which could get an incorrect value, causing the program to report failure. The bug in `jigsaw` is due to atomicity violations involving the field `w3c.tools.resources.store.ResourceStoreManager.loadedStore` due to statements `loadedStore++` and `loadedStore--` without synchronization; as a result, `loadedStore` may contain an incorrect value. The error in `jigsaw` described in [50] does not appear in our experiments, because the relevant

code was modified in the newer version of `jigsaw` that we tested. The above atomicity violations involve data races. The errors in `Vector` and `Stack` are from atomicity violations on the field `elementCount` (as discussed in Chapter 1).

The off-line reduction-based algorithm produces more false alarms than the block-based algorithm. For example, some `Collection` classes use `modCount` to count modifications. Thus, when an update method m_1 executes `modCount++` (which is a read followed by a write), and another method m_2 checks for recent modifications by reading `modCount`, there is a serializable sequence of events m_1:$read(modCount)$ m_2:$read(modCount)$ m_1:$write(modCount)$. The block-based algorithm does not produce a warning. But the benign data race on `modCount` may cause the reduction-based algorithms (on-line and off-line) to produce an atomicity warning (a false alarm). Similar scenarios exist in `jigsaw` (*e.g.*, on the field `alive` in the method `w3c.util.CachedThread.waitForRunner()`) and other programs.

For `Vector` and `Stack`, the on-line reduction-based algorithm produces fewer false alarms than the off-line reduction-based algorithm, because it misses some data races. By luck, the data races are benign and do not cause atomicity violations, but produce false alarms in the off-line algorithm. The on-line algorithm uses [49]'s race detection algorithm which assumes that the ownership of a variable is transferred when a second thread accesses the variable, but the ownership of a `Collection` class in our driver is not really transferred at that time. On the other hand, missed data races may cause the on-line reduction-based algorithm to miss some atomicity violations, as in `raytracer`, discussed above.

7.6 The Benefits of Different Techniques

Table 7.2 shows the benefits of different improvements to the off-line reduction-based algorithm. For each program, three groups of experimental data are shown: atomicity violations, data races, and execution time. The results for atomicity violations are aggregated as in Table 7.1, *i.e.*, based on the method executed by the transaction. The results for data races are the numbers of fields for which warnings are issued. A field of a class is counted only once, even if warnings are issued for multiple instances of the class. The columns show cumulative improvements. For example, the column labeled with "happen-before" also uses escape analysis, and the last column uses all four improvements.

In Table 7.2, "none" means that the lockset algorithm of [49] is used to detect data races. When "escape" or "happen-before" is used as an option, a revised Eraser lockset algorithm is used: when an object escapes, all its fields are regarded as in "exclusive" state; this corresponds to the state "exclusive2" in the lockset algorithm of [49]. With the option "happen-before", thread period IDs are used to track happen-before relations based on start-join and barriers. With the option "multi-lockset", the multi-lockset algorithm of Section 4.5 is used to detect data races. With the option "AcqA*Rel", Theorem 4.4.2 is

Program		none	escape	happen-before	multi-lockset	AcqA*Rel
elevator	atmcty vlts	0-2-0	0-2-0	0-2-0	0-2-0	0-2-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
	time	0.4s	0.5s	0.5s	0.5s	0.5s
tsp(map4)	atmcty vlts	0-0-1	0-0-1	0-0-1	0-0-1	0-0-1
	data races	0-0-8-0	0-0-8-0	0-0-0-0	0-0-0-0	0-0-0
	time	0.5s	0.5s	0.5s	0.5s	0.5s
tsp(map14)	atmcty vlts	0-2-2	0-2-2	0-2-0	0-2-0	0-2-0
	data races	0-7-3-1	0-7-3-1	0-7-0-1	0-8-0-0	0-8-0
	time	1m1.3s	25.7s	32.6s	34.1s	34.0s
sor	atmcty vlts	0-0-2	0-0-2	0-0-2	0-0-0	0-0-0
	data races	0-0-2-0	0-0-2-0	0-0-2-0	0-0-0-0	0-0-0
	time	51.3s	51.0s	52.0s	53.4s	53.3s
hedc	atmcty vlts	0-0-3	0-0-3	0-0-3	0-0-3	0-0-1
	data races	0-1-0-1	0-1-0-1	0-1-0-1	0-2-0-0	0-2-0-0
	time	0.8s	1.0s	0.8s	1.0s	1.0s
moldyn	atmcty vlts	0-0-2	0-0-2	0-0-0	0-0-0	0-0-0
	data races	0-0-2-0	0-0-2-0	0-0-0-0	0-0-0-0	0-0-0
	time	42m30s	28m23.1s	34m11.7s	35m15.3s	38m22.1s
montecarlo	atmcty vlts	0-0-0	0-0-0	0-0-0	0-0-0	0-0-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
	time	12m40.5s	8m8.5s	8m16.5s	8m22.2s	8m10.1s
raytracer	atmcty vlts	out of	2-0-0	2-0-0	2-0-0	2-0-0
	data races	memory	1-0-0-0	1-0-0-0	1-0-0-0	1-0-0
	time	after 2 hours	12m1.2s	11m54.5s	11m50.8s	12m8.7s
jigsaw	atmcty vlts	1-3-3	1-3-2	1-3-2	1-3-2	1-3-1
	data races	2-8-28-0	2-12-0-0	2-12-0-0	2-12-0-0	2-12-0
	time	2.73s	2.69s	2.64s	2.80s	2.74s
StringBuffer	atmcty vlts	0-1-0	0-1-0	0-1-0	0-1-0	0-1-0
	data races	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0-0	0-0-0
Vector	atmcty vlts	4-4-10	4-4-10	4-4-10	4-4-10	4-4-10
	data races	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0
Hashtable	atmcty vlts	0-2-4	0-2-4	0-2-4	0-2-4	0-2-3
	data races	0-2-1-0	0-2-0-0	0-2-0-0	0-2-0-0	0-2-0
Stack	atmcty vlts	3-4-12	3-4-12	3-4-12	3-4-12	3-4-12
	data races	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0-0	0-1-0

Table 7.2: The benefits of different improvements to the off-line reduction-based algorithms. The three categories for atomicity violations are bug - benign - false alarm. The four categories for data races are bug - benign - false alarm - missed warning.

used instead of Theorem 4.3.1.

Table 7.3 compares the benefits of different improvements to the block-based algorithm. The columns show cumulative improvements. The “none” column means that only locks, not escape and happen-before information, are considered when determining whether an event can occur between two other events. For each improvement, the column “methods” reports the number of methods such that an atomicity warning is issued for a transaction which is an execution of that method or part code of that method, and the column “fields” reports the number of fields such that an atomicity warning is issued involving accesses to that field. The three categories for “methods” are bug - benign - false alarm. The first category of “fields” is the numbers of fields such that an atomicity warning is issued for a 1v-block involving an access to that field; the second category of “fields” is the number of pair of fields such that an atomicity warning is issued for a 2v-block involving accesses to

Program	None			Escape			happen-before		
	methods	fields	time	methods	fields	time	methods	fields	time
elevator	0-2-3	6-6	11.0s	0-2-0	2-0	0.6s	0-2-0	2-0	0.6s
tsp(map4)	0-0-1	1-0	0.5s	0-0-1	1-0	0.5s	0-0-0	0-0	0.5s
tsp(map14)	0-2-1	5-8	10m44.1s	0-2-1	5-8	9m24.3s	0-2-0	3-7	9m16.2s
sor	0-0-2	36-144	1m3.6s	0-0-2	36-144	1m3.7s	0-0-0	0-0	1m4.1s
hedc	0-0-5	5-2	9.8s	0-0-3	1-0	1.9s	0-0-0	0-0	2.1s
moldyn			>13hrs	0-0-2	6-12	29m8.4s	0-0-0	0-0	28m54.6s
montecarlo			>20hrs	0-0-0	0-0	8m18.9s	0-0-0	0-0	8m11.4s
raytracer			>20hrs	2-0-0	1-0	37m7.6s	2-0-0	1-0	37m44.1s
jigsaw			>20hrs	1-3-1	13-112	7m42.4s	1-3-0	13-102	8m25.4s
StringBuffer	0-1-0	1-2	-	0-1-0	1-0	-	0-1-0	1-0	-
Vector	4-4-0	3-0	-	4-4-0	3-0	-	4-4-0	3-0	-
Hashtable	0-2-1	2-0	-	0-2-0	2-0	-	0-2-0	1-0	-
Stack	3-4-0	4-0	-	3-4-0	4-0	-	3-4-0	4-0	-

Table 7.3: The benefits of different improvements to the block-based algorithm. A dash for "time" means that the running time is negligible. A blank for "methods" or "fields" means that the datum is unavailable.

these two fields.

We can see from Table 7.3 that dynamic escape analysis speeds up the block-based algorithm on several programs, because it eliminates processing of accesses to unescaped variables (just checking whether they are escaped is fast). It also speeds up the off-line reduction-based algorithms in several cases; this can be seen by comparing the first two columns in Table 7.2. Table 7.4 shows the ratio of unescaped events to total events on all variables. Besides improving efficiency, dynamic escape analysis can also eliminate some false alarms. For example, Table 7.2 shows that dynamic escape analysis removes many false alarms for data race and atomicity on `jigsaw`; Table 7.3 shows that several false alarms for atomicity are eliminated by dynamic escape analysis on `elevator`, `hedc` and `Hashtable`.

Happen-before analysis can also eliminate some false alarms. For example, the happen-before analysis based on start and join removes false alarms on `tsp(map14)` in Table 7.2 and on `tsp(map4,map14)` and `hedc` in Table 7.3. The happen-before analysis based on barrier removes false alarms on `moldyn` in Tables 7.2 and 7.3.

The multi-lockset algorithm eliminates some false alarms on `sor`; these false alarms remained even after escape and happen-before analysis were applied. The multi-lockset algorithm reveals data races missed by the revised Eraser lockset algorithm in `tsp(map14)` and `hedc`. This can be seen in Table 7.2.

Special treatment of $AcqA^*Rel$ (*i.e.*, using Theorem 4.4.2 instead of Theorem 4.3.1) reduces the number of false alarms for some programs. For example, we can see in Table 7.2 that it eliminates some false alarms for atomicity in `hedc`, `jigsaw`, and `Hashtable`.

The results for "fields" in the column "happen-before" of Table 7.3 shows that for most programs, all of warnings are for 1v-blocks. This suggests that the algorithm in Section 5.1 is sufficient to find most atomicity violations. The algorithm in Section 5.2 is slower and the additional warnings it produces are typically more difficult to diagnose as bug or benign, because they involve two variables, and diagnosis requires understanding how updates to the two variables should be related.

Program	lines of code	total events	fraction unescaped events	off-line reduction storage	# of blocks	commit-node algorithm storage	multi-lockset size	Eraser lockset size
`elevator`	528	26K	43%	184	108	342	86	20
`tsp(map4)`	706	1.3K	40%	60	155	330	80	1
`tsp(map14)`	706	14M	68%	543	13474	3015	532	40
`sor`	251	16M	97%	64	3056	90	49	0
`hedc`	2197	2.9K	21%	350	1085	892	386	31
`moldyn`	1265	1.6G	86%	978	132	3819	2429	0
`montecarlo`	3619	477M	99%	79	159	148	92	0
`raytracer`	1832	3.5G	99%	31	39	106	29	1
`jigsaw`	25012	8.4K	51%	2011	12508	4031	2169	110

Table 7.4: Comparison of storages, and the ratio between unescaped events and escaped events.

7.6.1 Storage

Table 7.4 characterizes the storage used. Results for `Collection` classes are omitted, because the storage used is small and depends mainly on the driver. "total events" is the total number of monitored events, including accesses to unescaped variables. "fraction unescaped events" is the ratio between the number of accesses to unescaped variables and the total number of events. "off-line reduction storage" shows the storage of the off-line reduction algorithm by the total size of all ($varsOne$ and $varsMul$) sets of variables in all transaction tree nodes discussed in Section 4.7. "# of blocks" shows the number of blocks (including 1v-blocks and 2v-blocks) stored by the block-based algorithm. The "storage of the commit-node algorithm" is the sum of the number of nodes and the number of inter-edges (which in these experiments is at most 2/3 of the number of nodes) in the trimmed view-forest. "multi-lockset size" shows the sum of the maximum sizes of all sets (including $ReadSets$, $WriteSet$, $ReadThreadSet$ and $WriteThreadSet$ for each monitored variable) maintained by the multiple-lockset algorithm. "Eraser lockset size" shows the sum of the maximal sizes of all locksets maintained by Eraser [44] algorithms.

The multi-lockset algorithm provides more accurate data race analysis with moderately increased storage. The storage overhead of both algorithms are negligible under the current computer systems. The relatively large difference on `moldyn` between "multi-lockset size" and "Eraser lockset size" is due to the use of barriers, which increase the number of thread period IDs. The Eraser lockset sizes are zero for `sor`, `moldyn`, and `montecarlo` because they use barrier synchronization which is not monitored by the Eraser lockset algorithm.

7.7 Conclusions

Generally, runtime analysis is unsound compared to static analysis because it depends on the input to the program and may miss errors, especially (but not limited to) errors in

unexecuted code. Still, our algorithms are sound in a limited sense. Given a particular execution as input, our dynamic escape analysis and happen-before analysis are conservative, and our off-line reduction-based, block-based, and commit-node algorithms are conservative tests for atomicity of execution fragments (called transactions) as defined in Chapter 2. The on-line version of the reduction-based algorithm is unsound, for reasons discussed in Sections 4.5 and 4.6.

In conclusion, the block-based algorithm is most accurate and produces most specific diagnostic information than the other two algorithms. The commit-node algorithm is less accurate than the block-based algorithm in theory, but they have the same accuracy in practice. The reduction-based algorithm is the least accurate. For efficiency, the commit-node algorithm is as fast as the reduction-based algorithm, and significantly faster than the block-based algorithm.

The experiments do not reveal any simple relationship between the running time and the number of events. This reflects the fact that the running time depends strongly on many other factors, *e.g.*, how many events produce the same blocks, when variables escape, the number of thread periods, lockset sizes, etc.

Escape analysis improves both efficiency and precision (*i.e.*, fewer false alarms). Happen-before analysis, the multi-lockset algorithm, and special treatment of $AcqA^*Rel$ also increase precision but incur some overhead.

7.8 Related Work

7.8.1 Detecting Potential for Atomicity Violations

Flanagan and Freund [15] proposed a reduction-based algorithm with the improvements in Section 4.6. Their tool, called Atomizer, implements the on-line reduction-based algorithm described in Section 7.1.

Compared with Atomizer, this dissertation contributes the following improvements to the reduction-based algorithms: (1) off-line checking, which avoids missing atomicity violations due to miss-classification of events; (2) more accurate treatment of accesses to thread-local and read-only variables, as described in Theorem 4.4.2; (3) a new multi-lockset algorithm that produces fewer false alarms than previous lockset algorithms; (4) use of dynamic escape analysis, which reduces false alarms and often reduces running time; (5) use of happen-before analysis in data race detection to reduce false alarms; (6) on the implementation side, our system analyzes arrays; Atomizer does not.

Model checking can also be used to check atomicity [23, 13]. Model checking provides stronger guarantees than runtime monitoring, because it explores all possible behaviors of a program. Also, many of the supporting analyses, such as dynamic escape analysis, analysis of array, deadlock detection, and special treatment of thread-local and read-only variables, etc., can be performed more easily and precisely in model checking than by program instrumentation [23]. However, model checking is more expensive and feasible only for programs with relatively small state spaces.

Cormac Flanagan and Shaz Qadeer developed a type systems for atomicity [19] which extends a race-free type system [14] and Lipton's reduction theorem [31]. These work are summarized in Chapter 9.

von Praun and Gross [50] present a static analysis to detect violations of *method consistency*, which is an extension of view consistency [4]. Method consistency and atomicity are also incomparable. Although their static analysis is unsound (in order to reduce the cost and the number of false alarms), it considers the entire program and therefore may be more thorough than runtime analysis in some cases. On the other hand, it produces more false alarms than our block-based algorithm, based on a comparison of the false alarms in our Table 7.1 with the false and spurious reports in Table 1 of [50].

Artho *et al.* developed a runtime analysis algorithm to detect *high-level data races* [4]. Absence of high-level data races is similar to atomicity. They introduce a concept of *view consistency* and utilize it to detect high-level data races. A *view* is the entire set of shared variables accessed in a synchronized block. Thread t_1 and thread t_2 are view consistent if the intersections of all views of t_1 with the maximal view of t_2 form a chain (with respect to the subset ordering $\subseteq$), and vice versa. View consistency and atomicity are incomparable (*i.e.*, neither implies the other) [51].

Min Xu *et al.* developed a tool to detect serializability violations [56]. It differs from atomicity detectors because it is concerned with particular program executions. [56] presents a technique to automatically infer transaction boundaries, whereas our runtime analysis for atomicity defines the transaction boundaries by the heuristic default or the user input.

Burrows and Leino presented a static technique based on critical section to detect *stale-value error* by tracking the assignments and uses of local variables [8]. In concurrent programs, a shared variable is often cached at the first access in one thread, the following accesses to the shared variable use the cached value, but another thread could update the shared variable without updating the cached value in the first thread, thus, the cached value becomes *stale*. Atomicity can prevent stale-value errors from happening on all share variables inside each atomic transaction; but it cannot prevent the errors from happening for inter-transactions.

Linearizability [26] is a correctness condition for objects shared by concurrent processes. Informally, a concurrent object o is linearizable if and only if each concurrent operation history h for o is equivalent to some legal sequential history s, and s preserves the real-time partial order of operations in h. The equivalence is based on comparing the arguments and return values of procedure calls. Legality is defined in terms of a specification of the correct behavior of the object. Linearizability is defined semantically, *i.e.*, in terms of the specification (correctness requirements) of the object. In contrast, we define atomicity in terms of operations performed by the implementation. We focus on proving atomicity rather than linearizability, because atomicity does not require a correctness specification. Atomicity can help establish linearizability: first show that the concurrent implementation executed sequentially (*i.e.*, single-threaded) satisfies the sequential specification, and then apply our analysis to show the procedures of the implementation are atomic.

7.8.2 Detecting Potential for Data Races

Related work on runtime data race detection is discussed in Section 4.5.

Several type systems are developed for statically ensuring race-freedom [7, 14]. They are summarized in Section 9.2.1.

[9] combines static analysis and dynamic analysis and considers happen-before relation based on `start` and `join`. [37] extends the happen-before relation to consider `wait` and `notify` as well. Compared to the multi-lockset algorithm, [37] is more accurate but maintains more locksets. Our happen-before analysis ignores `wait` and `notify`, but considers barriers which [37] does not. Furthermore, we use dynamic escape analysis, whereas [9] uses static escape analysis. The reduction-based algorithm could be improved by using the race-detection techniques in [9], but it would still produce more false alarms than the block-based algorithm, because imprecise race detection is only one of causes of the additional false alarms.

Vaziri *et al.* proposed a new definition of a data race as a collection of a few problematic interleaving scenarios, which subsumes traditional data races, stale value errors and inconsistent views [47]. Their problematic interleaving scenarios are similar to our unserializable patterns discussed in Chapter 5, but are more concise. One main difference is that our work focuses on detecting atomicity violations, while their work focuses on statically inferring synchronization to avoid concurrency errors by allowing programmers to specify atomic sets.

7.8.3 Detecting Potential for Deadlocks

Klaus Havelund proposed the GoodLock algorithm detects potential deadlocks at runtime [24]. This GoodLock algorithm is summarized in Section 9.4.1.

Engler *et al.* [12], von Praun [48], and Williams *et al.* [55] developed inter-procedural static analyses that detect potential deadlocks in programs. These static analyses are also based on checking whether locks are acquired in a consistent order by all threads. These static analyses are more sophisticated and more accurate than basic deadlock types but still produce numerous false alarms (Engler *et al.* and Williams *et al.* partially address this problem by using heuristics to rank or suppress warnings that seem more likely to be false alarms), so it would be useful to use them in conjunction with run-time checking, which generally produces fewer false alarms.

Chapter 8

Static Analysis of Atomicity for Non-Blocking Algorithms

8.1 Introduction

Many concurrent programs use blocking synchronization primitives, such as locks and condition variables. *Non-blocking synchronization primitives*, such as Compare-and-Swap, and Load-Linked / Store-Conditional, never block (*i.e.*, suspend execution of) a thread. Non-blocking (also called "lock-free") synchronization is becoming increasingly popular, because it offers several advantages, including better performance, immunity to deadlock, and tolerance to priority inversion and pre-emption [34, 35].

An important use of non-blocking synchronization is in the implementation of non-blocking objects. A concurrent implementation of an object is *non-blocking* if it guarantees that some process can complete its operation on the object after a finite number of steps of the system, regardless of the activities and speeds of other processes [25]. Non-blocking synchronization is also used to implement blocking objects, such as spin locks.

Algorithms that use non-blocking synchronization are often subtle and difficult to design and verify. This chapter presents a static analysis to show that code blocks using non-blocking synchronization are *atomic*. Informally, a code block is atomic if every execution is equivalent to one in which the code block is executed serially, *i.e.*, without interruption by other threads. Atomicity is well known in the context of transaction processing, where it is sometimes called *serializability*.

Atomicity is an important correctness requirement for many concurrent programs. Furthermore, each atomic code block can be treated as a single transition during subsequent static or dynamic analysis of the program; this can greatly improve the efficiency of the subsequent analysis.

This chapter presents a conservative intra-procedural static analysis to infer atomicity. We build on Flanagan *et al.*'s work on atomicity types [19] and purity [18] in order to develop an analysis that is much more effective for programs that use non-blocking synchronization primitives. Our analysis first classifies all actions (*i.e.*, operations) in the

program into different types based on their commutativity and atomicity, which are determined based primarily on how locks and non-blocking synchronization is used in the program. The analysis then combines those atomicity types to determine the atomicity types of larger code blocks.

We formalize the analysis for a language SYNL that allows declaration of top-level procedures (as in an API) that implicitly get concurrently called by the environment. The language does not allow explicit procedure calls; internal procedures are inlined, and we do not handle recursion. The analysis can be extended to be inter-procedural. It applies equally to non-blocking objects and blocking objects.

The analysis is incomplete (*i.e.*, sometimes fails to show atomicity), but is effective for common patterns of non-blocking synchronization, as demonstrated by the applications in Section 8.6. We applied it to three interesting non-trivial non-blocking programs, one of them is the running example in Sections 8.4 and 8.5, the other two are described in Section 8.6. Although in two cases we must modify the algorithm before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. We believe our analysis provides a useful method for manual verification of atomicity, as well as being suitable for automation.

8.2 Related Work

This chapter extends our previous work [52] with a formal semantics for the language SYNL and correctness proofs for core aspects of the analysis.

Gao and Hesselink [21] used simulation relations to prove that a non-blocking (called lock-free in [21]) algorithm refines a higher-level (coarse-grained) specification. Using the PVS theorem prover, they proved correctness of algorithms similar to the ones in Figures 8.1, 8.5 and 8.6. The proofs took a few man-months and are not easily re-usable for new algorithms.

Flanagan *et al.* developed type systems [19] based on Lipton's reduction theorem [31] to verify atomicity. Wang and Stoller [51, 54, 53] and Flanagan *et al.* [15] developed runtime algorithms to check atomicity. All of this work focuses on locks and is not effective for programs that use non-blocking synchronization.

Flanagan *et al.* extended their atomicity type system with a notion of purity [18]. A code block is pure if, when it terminates normally, it does not change the program state. Non-blocking programs often contain code blocks that abort an attempted update to a shared variable if the variable was updated concurrently by other threads; these code blocks are often pure according to our definition of purity, which generalizes the definition in [18] by taking into account liveness of variables and use of unique references. The type system in [18] can show atomicity of simple non-blocking algorithms but not of any of the three algorithms analyzed in this chapter, because it does not accurately analyze usage of non-blocking synchronization primitives; for example, it has no analogue of the notions

of "matching read" or "matching LL" in Section 8.5.2, and does not analyze exceptional variants (also defined in Section 8.5.2) of a procedure separately.

Atomicity used to optimize model checking can be regarded as a partial-order reduction [39], *i.e.*, a method for exploiting commutativity to reduce the number of states explored by a verification algorithm. For non-blocking algorithms, traditional partial-order reductions are less effective than our analysis, because they do not distinguish left-movers and right-movers, and they focus on exploiting commutativity of operations with little regard for the context in which the operations are used, while our analysis considers in detail the context (surrounding synchronization and conditions) of each operation.

The model-checking (*i.e.*, state-space exploration) algorithm in [41] dynamically identifies transactions, which correspond roughly to executions of atomic blocks. Their algorithm relies on a separate analysis to determine commutativity of actions. An inter-procedural extension of our analysis could be used for this. This would allow their algorithm to be applied effectively to non-blocking programs.

Linearizability [26] is a correctness condition for objects shared by concurrent processes. Informally, a concurrent object o is linearizable if and only if each concurrent operation history h for o is equivalent to some legal sequential history s, and s preserves the real-time partial order of operations in h. The equivalence is based on comparing the arguments and return values of procedure calls. Legality is defined in terms of a specification of the correct behavior of the object. We focus on proving atomicity rather than linearizability, because atomicity does not require a specification of correct behaviors. Atomicity can help establish linearizability: first show that the concurrent implementation executed sequentially (*i.e.*, single-threaded) satisfies the sequential specification, and then apply our analysis to show that the procedures of the implementation are atomic.

8.3 Background

8.3.1 Non-Blocking Synchronization Primitives

Non-blocking synchronization primitives include *Load-Linked* (LL) and *Store-Conditional* (SC), supported by PowerPC, MIPS, and Alpha, and *Compare-and-Swap* (CAS), supported by IBM System 370, Intel IA-32 and IA-64, Sun SPARC and the JVM in Sun JDK 1.5.

LL($addr$) returns the content of the given memory address. SC($addr, val$) checks whether any other thread has written to the address $addr$ (by executing a successful SC on it) after the most recent LL($addr$) by the current thread; if not, the new value val is written into $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and `false` is returned to indicate failure. Another primitive VL (validate) is often supported. VL($addr$) returns `true` iff no other thread has written to $addr$ after the most recent LL($addr$) by the current thread.

In a run of a program, the *matching* LL (if any) for a SC(v, val) or VL(v) action is the last LL(v) before that action in the same thread. If there is no matching LL for a SC action,

the SC action fails.

CAS($addr, expval, newval$) compares the content of address $addr$ to the expected value $expval$; if the two values are equal, then the new value $newval$ is written to $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and the operation returns `false` to indicate failure.

CAS is also supported in the JVM of Sun JDK 1.5. There are few `synchronized` blocks or `synchronized` methods in the `java.util.concurrent` package. A `Lock` class implemented using CAS, which offers higher performance, is used instead.

8.3.2 A Language: SYNL

We formalize our analysis for a language SYNL (Synchronization Language). The syntax of SYNL is shown in Table 8.1. There is no explicit procedure call, as discussed in Section 8.1.

In SYNL, global and local variables are distinguished syntactically, as described below. An *unshared object* is an object accessed by only one thread. An *unshared variable* is a local variable or a field of an unshared object. A *shared variable* is a global variable or a field of a shared object. A simple escape analysis is used to determine when objects become shared.

A program consists of global variable declarations, and procedure definitions.

An execution of a SYNL program consists of an arbitrary number of invocations (by the environment) of its procedures with arbitrary type-correct arguments (for brevity, we leave the type system implicit), and with an arbitrary amount of concurrency. Therefore, SYNL does not need constructs to create threads.

Expressions in SYNL include constant values, variables, field accesses, array accesses, non-blocking synchronization, `new` operation to allocate objects, and calls to primitive operations. Variables may have primitive types and reference types. A local variable may contain a reference to a shared object. For example, a field access `x.fd` may access both a local variable `x` and a shared variable (*i.e.*, a field of a shared object). Primitive operations (such as arithmetic operations) have no side effect.

Statements include assignments, `synchronized` (for lock synchronization), sequential composition, conditionals, `local` blocks, loops, `return`, `break`, and `skip`. `synchronized` has the same semantics as in Java. A `local` statement introduces a scoped variable. The loop statement defines an unconditional loop: "`loop` *s*" is equivalent to "`while (true)` *s*". Every `while` loop can be re-written using `loop`, `if`, and `break`. All loops in SYNL are unconditional.

As syntactic sugar, we allow non-blocking primitives to be used as statements when their return values are not needed; for example, SC(x, e) used as a statement is syntactic sugar for: `local` $dummy$ = SC(x, e) `in skip`.

An *execution* is an initial state and a sequence of transitions. A program state is a tuple which consists of a global store G, a heap H, each thread's local store L and program statements (this indicates the next statement to execute; it takes the place of a program counter).

$Program$	::=	`global` var^* ; `procedure` $proc^*$
$Procedure$	::=	$pn(var^*)\ stmt^*$
$Statement$	::=	$loc := e$ | `synchronized`$(e)\ s$ | $s; s$ | `if` $e\ s\ s$ | `local` $x := e$ `in` s | `loop` s | `return` | `return` e | `break` | `skip`
$Expr$	::=	val | loc | `CAS`(loc, e, e) | `LL`(loc) | `VL`(loc) | `SC`(loc, e) | `new` C | $prim(e, ...)$
$Location$	::=	x | $x.fd$ | $x[e]$
$proc$	$\in$	$Procedure$
s	$\in$	$Statement$
e	$\in$	$Expr$
loc	$\in$	$Location$
pn	$\in$	$ProcedureName$
val	$\in$	Val
x	$\in$	$Variable$
fd	$\in$	$Field$
$prim$	$\in$	$PrimitiveOperation$
C	$\in$	$Class$

Table 8.1: Syntax of SYNL.

Each transition corresponds to one step of evaluation of an expression or statement in a standard way. The formal definitions of states and transitions are in Appendix B.2. For each transition, we consider the action performed by it. These actions capture the relevant behavior of the transition for our analysis and are described in Section 8.3.3. Note that all constructs in SYNL are deterministic, so the intermediate states during an execution are uniquely determined by the initial state and the sequence of transitions, and we will sometimes talk about executions as if those states were present in it.

Code blocks in a program P are *atomic* if: for all reachable states s of P, if all threads are executing outside those code blocks in s, then s is also reachable in an execution of P in which those code blocks are executed atomically, ***i.e.***, without interruption by other threads (note that the reverse implication trivially holds).

8.3.3 Commutativity and Atomicity Types

A reference variable p in a program P is *unique* if, in every reachable state of P, there is at most one location containing the reference value (***i.e.***, an object identifier or address) contained p. Please note that a unique reference can be contained only in a local variable. In this chapter, "reference" is often short for "reference variable". A *local action* is an access to an unshared variable (***i.e.***, a local variable or a field of unshared object) or a field of a shared object accessed by dereferencing only unique references, starting with a unique reference stored in a local variable. Both of these kinds of accesses are always both-movers (defined below) and are treated the same way in our analysis, so it is convenient to refer to both of them as local actions. Any static uniqueness analysis may be used to identify

unique references.

Conservatively, other variable accesses can be treated as *global actions*. Acquire and release on shared locks are also global actions. Thus, there are four kinds of global actions: read, write, acquire lock, and release lock. Let $R(v)$, $W(v)$, $acq(v)$, and $rel(v)$ denote these global actions, respectively, where v denotes the accessed variable or lock. LL and VL are global reads. SC and CAS are global writes to their first argument and, if their second or third argument are shared variables, also global reads of those variables. Note that arithmetic operations, *etc*., do not affect our analysis and hence are not treated as actions.

Following [31], actions are classified according to their commutativity. An action is a *right-mover*/*left-mover* if, whenever it appears immediately before/after an action from a different thread, the two actions can be swapped without changing the resulting state. An action is a *both-mover* if it is a left-mover and a right-mover. An action not known to be a left-mover or right-mover is *atomic* (since a single action is executed in a single step of execution).

Theorem 8.3.1. *Local actions are both-movers.*

Proof. Accesses to unshared variables are obviously both-movers. For a field access performed by dereferencing only unique references, suppose that the field is f, and thread t executes an action a that accesses $o.f$ by dereferencing some local variable l (*i.e.*, l contains a unique reference to o) or local variable l and a sequence of fields $f_1, f_2, ..., f_n$ where $l.f_1, l.f_1.f_2, ..., l.f_1...f_2$ are unique references and $l.f_1...f_n$ contains the reference to o. Before another thread can access $o.f$, t must transfer the unique reference in l, f_1, ..., or f_n into a global variable. This implies that a is a right-mover (because any action of another thread that occurs immediately after a cannot access $o.f$). Symmetrically, a is a left-mover because, between a and the closest preceding access to $o.f$ by another thread, t must transfer the unique reference from a global variable into l, $l.f_1$, ..., or $l.f_1...f_n$. □

Theorem 8.3.2. *Lock acquires are right-movers. Lock releases are left-movers.*

Proof. See [19], from which this theorem is taken. Here is a proof sketch. For $acq(v)$, its immediate successor global action a from another thread can not be a successful $acq(v)$ or $rel(v)$, because $acq(v)$ would block, and $rel(v)$ would fail (in Java, it would throw an exception). Hence $acq(v)$ and a can be swapped without affecting the result, so lock acquire is a right-mover. For similar reasons, lock release is a left-mover. □

Theorem 8.3.3. *(1) For a global read $R(v)$, if no global write $W(v)$ from other threads can happen immediately before/after $R(v)$, $R(v)$ is a left/right mover. (2) For a global write $W(v)$, if no global read $R(v)$ or write $W(v)$ from other threads can happen immediately before/after $W(v)$, $W(v)$ is a left/right mover.*

Proof sketch. The main observations are that two reads commute, and accesses to different variables commute. □

We briefly review atomicity types, which were introduced by Flanagan and Qadeer [19]. An atomicity type is associated with an expression or statement. The atomicity types are: right-mover (R), left-mover (L), both-mover (B), atomic (A), and non-atomic (N, called compound in [19]). The first three mean that all actions executed by the expression or statement have the specified commutativity. Atomic has the same meaning as in Section 8.3.2. Non-atomic is used when none of the other atomicity types are known to apply. Atomicity types are partially ordered such that smaller ones give stronger guarantees. The ordering is: $B \sqsubset t \sqsubset A \sqsubset N$ for $t \in \{L, R\}$. The atomicity type of an expression or statement can be computed from the atomicity types of its parts using the following operations on atomicity types. The *join* (*i.e.*, least upper bound) operation based on this ordering is used to compute the atomicity of `if` statements from the atomicity types of the `then` and `else` branches. The *iterative closure* t^* of an atomicity type t denotes the atomicity of a statement that repeatedly executed a sub-statement with atomicity type t. It is defined by: $B^* = B$, $R^* = R$, $L^* = L$, $A^* = N$, $N^* = N$. The *sequential composition* $a; b$ is defined by the following table (the rows are labeled by the first argument; the columns are labeled by the second argument):

;	B	R	L	A	N
B	B	R	L	A	N
R	R	R	A	A	N
L	L	N	L	N	N
A	A	N	A	N	N
N	N	N	N	N	N

8.4 Pure Loops

For a loop (recall that all loops in SYNL are unconditional, like `while (true)` s), if an iteration terminates exceptionally via a `break` or `return` statement, it is called an *exceptional iteration*; otherwise, it is called a *normal iteration*. Note that we do not consider nested loops in this chapter for simplicity. We define pure loops based on the notion of pure statements introduced in [18]*. Informally, a loop is *pure* if all normal iterations of the loop have no side effect. Typically, a normal iteration performs checking operations for some state conditions by using actions with no side effects, and finds that the conditions are not satisfied, so another iteration is needed. When these conditions are satisfied, an exceptional iteration occurs; it may have side effects and exit the loop. Therefore, following the idea proposed in [18], to determine the atomicity of a pure loop, we may ignore its normal iterations and focus on its exceptional iterations.

Note that pure is not the same as side-effect free, because a pure loop may have side effects in exceptional iterations.

*In our framework, unlike [18], purity is a property (of loops) that has no effect on the operational semantics.

A simple example of a pure loop appears in the following implementation of the `Down` operation on a semaphore. Iterations that end at `return` are exceptional. Iterations that end at line 4 (*i.e.*, when `tmp > 0` is false) or line 5 (*i.e.*, when SC returns `false`) are normal and have no residual side effects.

```
1  Down(sem) {
2    loop
3      local tmp = LL(sem) in
4        if (tmp > 0)
5          if (SC(sem, tmp-1))
6            return;
7  }
```

8.4.1 Formal Definition of Pure Loops

8.4.1.1 Classification of References

A reference variable p is *quasi-unique* in a program P if p is unique (as defined in Section 8.3.3) when locations accessed in normal iterations of all loops are ignored (*i.e.*, such locations may contain the same reference as p).

If a quasi-unique reference variable p, which is not truly unique, is accessed only during normal iterations of loops, p is called a *secondary reference*. Other quasi-unique reference variables (including truly unique references) are called *primary references*. Obviously, quasi-unique references accessed in exceptional iterations are primary references. All truly unique references are primary references.

An example of a pure loop appears in Figure 8.1 which shows Herlihy's algorithm for non-blocking concurrent implementation of small objects [25]. Suppose a small object (*i.e.*, small enough to be copied efficiently) is shared by a set of threads. The main steps on each thread in the algorithm are: (1) read the shared object reference Q using LL; (2) copy the data from the shared object into a private (*i.e.*, currently unshared) working copy of the object, *i.e.*, the object referred by prv; (3) perform the requested computation on the private object; (4) switch the reference values in Q and prv between the shared object and the private object using SC and an assignment statement. Note that, the formerly shared object becomes a private copy, and the formerly private object becomes the current shared copy.

Before a thread t_1 switches the reference of the shared copy o with the private copy of t_1, another thread t_2 may read the reference to o using `LL(Q)`. Even though o becomes the private copy of t_1, t_2 may still hold the reference to o, though the SC of t_2 will fail later, causing t_2 to loop and read the current reference from `Q`. Thus, t_1 may write to o while t_2 copies data from o. If t_2 tried to perform a computation on a copy of the data that reflects only part of some update, it might suffer a fatal error, such as divide by zero. Line

```
void proc(Val input)
1   loop
2     local m = LL(Q) in
3       copy(prv.data,m.data);
4       if (!VL(Q)) continue;
5       computation(prv.data,input);
6       if (SC(Q,prv))
7         prv = m;
8         break;
```

Figure 8.1: Herlihy's non-blocking algorithm for small objects. `Q` is a shared variable.

a3 prevents this: if $o.data$ (accessed as `m.data`) is modified by another thread during the copy in line a2, the VL will fail.

At the beginning of any loop, both `Q` and `prv` are unique references. All references are quasi-unique according to its definition. When reference values in Q and prv have been switched between the shared object and the private object, some other loop may still keep the old reference to the shared object which has become private. Thus, prv may be alias of m and Q during some iterations from line 2 to line 6. The reference prv is a primary reference, all its aliases can be only Q and m in some other normal iterations, which are secondary references.

In this dissertation, primary and secondary reference are identified manually. Developing conservative static analysis to automatically identify them is future work.

8.4.1.2 Pure Actions and Pure Loops

Informally, an action in a normal iteration of a loop is *pure* if any update performed by the action is not visible to other threads or to the current thread after the current normal iteration; in other words, there is no data flow from the action to outside (considering this thread and other threads) of the normal iteration in which it occurs. Formally, a pure action should satisfy the following two conditions:

(1) If it performs an update, the target location loc must satisfy the following conditions: (*i*) loc is a local variable, a field of an unshared object, or a field of a shared object accessed by dereferencing only primary references; and (*ii*) for all paths in the control flow graph from the end of the loop body (*i.e.*, the program point at the end of the loop body from which control flows back to the beginning of the loop body) to the end of the loop, (*1.ii.a*) if there is any access to loc, the first one must be a write, and (*1.ii.b*) if loc is not accessed on some such path, then loc is loop-local (*i.e.*, a local variable, or a field of an unshared object).

(2) If it is a LL action, each SC(loc,-) that can match it is also in the loop and there is a LL(loc) on every path from loop entry to the SC.

Condition (1) as a whole ensures that the value of loc at the end of the loop body is dead, *i.e.*, will not be read. Condition (*1.ii.a*) implies that the value written by the pure action is overwritten before the next read from loc in this loop. Condition (*1.ii.b*) ensures that, if that value is not overwritten on some path, then loc (hence that value) is not visible outside this loop. Condition (2) ensures that, in every execution, the matching SC for that LL occurs in the same iteration as the LL. This special condition for LL is needed because LL implicitly performs an update that can affect subsequent SCs by the same thread.

A loop is *pure* if, for each normal iteration of the loop, every action that can occur in it is pure. To check whether a loop is pure, we construct a control flow graph (CFG), analyze it to identify actions that can occur in normal iterations of the loop, and then check whether those actions are pure according to the above definition. There is a special case for SC and CAS. When a SC is used as the test condition of an `if` statement (*e.g.*, the SC in Figure 8.1), if only the false branch of the `if` statement can be executed under normal iterations, the SC is treated as a read (not an update). CAS is handled similarly.

In Herlihy's small object algorithm shown in Figure 8.1, all actions in normal iterations are pure, so the loop is pure.

8.4.1.3 Normal Iterations of Pure Loops Can Be Deleted

Theorem 8.4.1 shows that normal iterations of a pure loop can be deleted from an execution without affecting the result of the execution. Informally, deleting a transition from an execution means removing it and adjusting the subsequent states. Details are in Appendix B.3. Theorem 8.4.1 is the basis for proving in Section 8.5 that normal iterations of pure loops can be ignored when analyzing atomicity.

Theorem 8.4.1. *Let σ be an execution of a program P. Let σ' be an execution obtained from σ by deleting all transitions in all normal iterations of all pure loops in P. Then σ' is also an execution of P, and σ and σ' contain the same states in which all threads are executing outside pure loops.*

Proof sketch. A detailed proof appears in Appendix B.3. Here is a proof sketch.

Recall that loops in SYNL are equivalent to `while (true)` s. When the body of a loop terminates normally, the thread begins another iteration of the same loop body.

According to the definition of pure loop, normal iterations perform no updates to global variables, no live residual updates to local variables and fields of unshared objects, and no updates to fields of shared objects except by dereferencing only primary references, even if the program's execution is interleaved with actions of other threads. The syntax of SYNL ensures that acquire and release actions occur in matching pairs in an execution of a loop body, so deleting them does not affect the resulting state or operations on the lock by other threads that could have occurred while this thread held the lock.

Consider an update in a normal iteration to a field of a shared object accessed by dereferencing only primary references. Let τ denote the transition that performs the update. Let t denote the thread that executes τ. Let loc denote the updated field of the shared object.

Condition (1) in the definition of pure action ensures that τ has no effect on the subsequent execution of t. We show below that τ has no effect on other threads. We consider two cases.

Case 1: all primary references dereferenced by τ are unique. If there is no access to loc from the end of normal iterations of the pure loop to the end of the pure loop, loc must be loop-local according to condition (*1.ii.b*) in the definition of pure action, hence τ has no effect to the other threads. If loc is updated from the end of normal iterations of the pure loop to the end of the pure loop, the first access to loc must be a write, let it be τ_w. Since all references are unique, there must be a read to loc before loc escapes to other threads. Thus, τ happens before τ_w, and τ_w happens before the escape point. Therefore, τ has no effect on other threads.

Case 2: some dereferenced primary reference of τ are not unique. If a primary reference is not unique, all its aliases are secondary references. By the same reasoning as in case 1, τ has no effect on other threads, except for accesses to these secondary references in normal iterations of pure loops. Because all normal iterations of pure loops are removed simultaneously, all effects of τ on other threads are eliminated. □

8.5 Checking Atomicity

The main issue in applying Theorem 8.3.3 is determining whether a global action can happen immediately before or after another global action. Our analysis determines this based on how synchronization primitives are used.

8.5.1 Lock Synchronization

Lock synchronization is well studied. We sketch a simple treatment of lock synchronization, to illustrate how analysis of locks fits into our overall analysis algorithm.

Theorem 8.5.1. *If expressions e_1 and e_2 appear in the bodies of different* `synchronized` *statements that synchronize on the same lock, then e_1 cannot be executed immediately before or after e_2.*

Proof sketch. Since e_1 and e_2 are protected by the same lock, at least one acquire and release of the lock must occur between e_1 and e_2 in any execution. □

Alias analysis may be used to determine whether two `synchronized` statements synchronize on the same lock.

8.5.2 Non-Blocking Synchronization

8.5.2.1 Exceptional Variants

Based on Theorem 8.4.1, for pure loops, it suffices to analyze atomicity of each exceptional iteration of the loop. For each `break` or `return` statement in a loop, the backward

slice of the loop body starting at that `break` or `return` and ending at the loop's entry point is called an *exceptional slice* of the loop.

The atomicity of a procedure can be determined by analyzing atomicity of its *exceptional variants*. Each exceptional variant is a specialized version of the procedure, and corresponds to a selection of exceptional slices of its pure loops, with each pure loop replaced by its selected exceptional slice. If the selected exceptional slice includes only the true branch of an "`if` $e\ S_1\ S_2$" statement, then we replace the `if` statement with "`TRUE`(e); S_1" in the corresponding exceptional variants of the procedure; if the slice includes only the false branch, we replace the `if` statement with "`TRUE`$(!e); S_2$". A SC expression in `TRUE(SC(`v, val`))` must be successful, and we call it a *successful* SC expression. Non-pure loops appear unchanged in the exceptional variants. As an example, the procedure in Figure 8.1 appears has one exceptional variant, shown below.

```
1    local m = LL(Q) in
2      copy(prv.data,m.data);
3      TRUE(VL(Q));
4      computation(prv.data,input);
5      TRUE(SC(Q,prv));
6      prv = m;
7      break;
```

Theorem 8.5.2. *If all exceptional variants of a procedure p are atomic, then p is atomic.*

Proof Sketch. Let P denote the original program that contains p. Let σ be an execution of P. Let φ be a state in σ in which all threads are executing outside p.

According to Theorem 8.4.1, an execution σ' of P can be obtained from σ by deleting all transitions in normal iterations of pure loops in procedure p, and φ is reachable in σ'. By the definition of exceptional variant, there must be exceptional variants of p which can produce the same exceptional iterations of pure loops of p as in σ'. Let P' denote the program obtained by replacing procedure p with such exceptional variants. Thus, σ' is also an execution of P'.

By hypothesis, all exceptional variants of p are atomic. Based on σ', by the definition of atomicity, there exists an execution σ'' of P' in which the exceptional variant of p are executed atomically and in which φ is reachable.

By the definition of exceptional variants of a procedure, every execution of an exceptional variant of p is also an execution of p. Therefore, σ'' is also an execution of P, and all executions of p in σ'' are executed atomically, and φ is reachable in σ''. Thus, by the definition of atomicity, p is atomic. □

8.5.2.2 Atomicity Analysis of Non-Blocking Synchronization Primitives LL/SC/VL

There is a unique matching LL action for each successful SC action in an execution. In program code, there might be multiple LL expressions or statements that can produce the

matching LL action for an occurrence of SC. We call these the *matching* LL expressions of the SC expression. For example, if there is an `if` statement before a SC, and both branches of the `if` statement contain LL, both of the LL expressions can possibly match the SC.

For a SC(v, val) in a program, to find its matching LL expressions, we do a backward DFS on the control flow graph starting from the SC, and not going past edges labeled with LL(v). All of the visited occurrences of LL(v) match the SC. For a VL(v), its matching LLs can be found in the same way.

We implicitly assume hereafter that each SC expression has a unique matching LL expression. This assumption is not essential, but it simplifies the analysis and is satisfied by the non-blocking algorithms we have seen. We also implicitly assume that a variable updated by a SC is updated only by SC, not by regular assignment or CAS.

Theorem 8.5.3. *A successful SC or VL is a left-mover, and the matching LL is a right-mover.*

Proof. By the semantics of LL, SC and VL, for a successful SC(v, val) or VL(v) action and its matching LL(v), any other SC action (successful or failed) on v executed by another thread cannot be executed between them. Therefore, the successful SC or VL is a left-mover, and the matching LL is a right-mover. □

Theorem 8.5.4. *Let SC and LL denote a successful SC(v, val) expression and its matching LL(v) executed by a thread t, respectively. Let SC' and LL' denote a successful SC(v, val') expression and its matching LL(v) executed by another thread t', respectively. SC', LL', and all transitions of t' between them cannot be executed between SC and LL.*

Proof sketch. According to Theorem 8.5.3, SC' cannot happen between LL and SC. Thus, there are two cases: SC' happens before LL, or SC' happens after SC. For the first case, the theorem obviously holds. For the second case, if LL' happens between LL and SC, SC must also happen between LL' and SC' (because SC succeeds), which is impossible according to Theorem 8.5.3; if LL' happens after SC, the theorem obviously holds. □

8.5.2.3 Atomicity Analysis of Non-Blocking Synchronization Primitive CAS

CAS is often used in a similar way as LL/SC. CAS takes an address, an expected value, and a new value as arguments. There is often an assignment before CAS to save the old value into a temporary variable that is used as the expected value. For a CAS, its *matching read*, if any, is the action which reads the old value and saves it as the expected value. Note that a CAS can succeed even without a matching read; a SC cannot succeed without a matching LL. We use a backward search on the control flow graph to find the matching reads for a CAS expression. We implicitly assume hereafter that there is a unique matching read for each CAS.

CAS-based programs may suffer from the ABA problem: if a thread reads a value A of a shared variable v, computes a new value A', and then executes CAS(v, A, A'), the CAS may succeed when it should not, if the shared variable's value was changed from A to B and

then back to A by CASs of other threads. A common solution is to associate a modification counter with each variable accessed by CAS [34]. The counter is read together with the data value, and each CAS checks whether the counter still has the previously read value. A successful CAS increments the counter. With this mechanism, variants of Theorem 8.5.3 and 8.5.4 hold for CAS: just replace "matching LL" with "matching read", and replace "SC" with "CAS".

8.5.3 Condition-based Non-Blocking Synchronization

A predicate $p(lvar)$ is called a *local condition* of a code block `local` $lvar$ = e `in` $stmt$ (which is called a *local block* on $lvar$), if it satisfies the following two conditions: (*i*) $lvar$ is not updated in $stmt$, and (*ii*) $p(lvar)$ holds throughout execution of $stmt$.

Condition (*i*) is easy to check, because there is no aliasing of local variables in SYNL. When condition (*i*) holds, a local condition for a block can easily be obtained from the `TRUE` statements in $stmt$ that depend only on $lvar$. For example, in the exceptional variant of procedure *Down* shown in Section 8.4, a local condition for the code block in lines 3-6 is $tmp > 0$. If condition (*i*) does not hold, or no appropriate `TRUE` statements appear in the local block, its local condition is `true`.

A local block of the form `local` $lvar$ = `LL`$(svar)$ `in` $\{stmt$; `TRUE(SC(`$svar, val$`));`$\}$ is called a *LL-SC block* on $svar$.

Theorem 8.5.5. *Suppose a shared variable $svar$ is updated only by SC expressions in LL-SC blocks, and every LL-SC block* `local` $lvar$ = `LL`$(svar)$ `in` $\{stmt$;`TRUE(SC(`$svar, val$`));`$\}$ *in the program has the same local condition $p(lvar)$. Suppose a local block S* `local` $lvar'$ = $svar$ `in` $stmt'$ *has a local condition* $!p(lvar')$.

(a) Any successful SC($svar$) in the LL-SC blocks cannot happen inside S.

(b) No transition in local block S can be executed inside any LL-SC block on $svar$, and no transition in any LL-SC block on $svar$ can be executed inside local block S.

Proof of (a). We prove (*a*) by contradiction. Suppose a successful SC($svar$) in a LL-SC block executed by another thread happens inside S. Without loss of generality, we consider the first such SC($svar$). According to the assumption, $!p(lvar)$ holds during $stmt'$. Because $svar$ is updated only by SC actions from LL-SC blocks, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until SC($svar$) happens. This implies that $!p(svar)$ holds when SC($svar$) happens. The LL-SC block has local condition $p(lvar)$, and $lvar == svar$ holds until the SC, because $lvar$ is not updated in the LL-SC block, and $svar$ is not updated before the first successful SC on it, so $p(svar)$ holds when SC($svar$) happens.

Proof of (b). According to Proof of (*a*), no successful SC($svar$) can happen inside S. Consider an execution of a LL-SC block on $svar$. There are two cases:

case 1: the successful SC happens before S. Thus, the whole LL-SC block happens before S. Obviously, the theorem holds in this case.

case 2: the successful SC happens after S. If the matching LL also happens after S, the whole LL-SC block happens after S. Hence the theorem holds. Suppose the matching LL happens inside S or before S. Similar to the proof of (*a*), because $svar$ is updated only by SC actions from LL-SC blocks, and no successful SCs happen inside S or between the matching LL and the SC, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until the SC($svar$) happens. Thus, $!p(svar)$ holds when SC($svar$) happens. By the same reasoning as in the proof of (*a*), $p(svar)$ holds when SC($svar$) happens. This contradicts the previous conclusion. Therefore, when SC happens after S, the matching LL cannot happen inside S or before S. □

The definition of LL-SC block and the above theorem can be generalized, so that the LL does not need to occur at the beginning of a local block, and the SC does not need to occur at the end of a local block. A similar theorem exists for CAS.

8.5.4 Atomicity Inference

To analyze atomicity of each procedure in a program, we identify pure loops, then check atomicity of its exceptional variants, by computing atomicity types for all expressions and statements. The algorithm is as follows:

- Step 1: Identify all local actions and lock actions. According to Theorem 8.3.1, all local actions have atomicity type B. According to Theorem 8.3.2, all lock acquires and releases have atomicity type R and L, respectively. A simple escape analysis is used to identify accesses to objects that have not escaped from the creating threads; those accesses are like accesses to unshared variables and have atomicity type B.

- Step 2: According to Theorem 8.5.3, if all updates on a variable v are done through SC, all successful SC(v, val) and VL(v) have atomicity type L, and their matching LL(v) have atomicity type R. A successful VL(v) between a successful SC(v, val) and the matching LL(v) is a both-mover. The analogous theorem for CAS is used for successful CAS and the matching reads.

- Step 3: Infer local conditions for local blocks, as described in Section 8.5.3.

- Step 4: Using Theorems 8.5.1, 8.5.3, 8.5.4 and 8.5.5, for each read, check whether there is a write on the same variable that can happen immediately before/after it; for each write, check whether there is a read or write on the same variable can happen immediately before/after it. For access to variables on the heap, the analysis does a case split on whether two field accesses refer to the same location; we consider both cases, unless alias analysis shows one is impossible. Our current alias analysis just checks whether the references have the same type and whether the same field is being accessed; if not, the two field accesses must be to different locations. Assign atomicity types to the reads and writes based on Theorem 8.3.3 if they are still not given atomicity types in previous steps.

- Step 5: For actions not given an atomicity type in previous steps, conservatively assign them atomicity type A.
- Step 6: Propagate atomicity types from the actions up through the abstract syntax trees of the procedures using the atomicity calculus in [19]. The atomicity type of a compound program construct is computed from the atomicity types of its parts using join, sequential composition, and iterative closure as appropriate.
- Step 7: For each procedure p in the original program, if every exceptional variant of p has a procedure body with atomicity type A, then by Theorem 8.5.2, p has atomicity type A.

As an example, we compute the atomicity of Herlihy's non-blocking algorithm for small objects in Figure 8.1. Recall that the procedure has one exceptional variant, given in Section 8.5.2.1. In step 1, local actions in line a7 and line a8 are identified and assigned atomicity type B. Since prv is a primary reference, the computation action in line a4 and the copy action in line a2 (the read in line a2 will consider later) are local actions, hence, they are assigned atomicity type B. In step 2, successful SC in line a5 has atomicity type L, the matching LL in line a1 has atomicity type R, and successful VL in line a3 has atomicity type B. Step 3 is skipped in this algorithm. In step 4, we know any write to `m.data` (which must through successful SC) in other threads cannot happen immediately before or after the read to `m.data` in line a2, since a successful SC is followed. Hence, both actions in line a2 have atomicity type B. Now we can propagate atomicity types from the actions, and conclude the only exceptional variant is atomic. Therefore, the whole procedure is atomic.

```
local m = LL(Q) in            a1:R local m = LL(Q) in
copy(prv.data,m.data);        a2:B   copy(prv.data,m.data);
if (!VL(Q)) continue;         a3:B   TRUE(VL(Q));
computation(prv.data,input);  a4:B   computation(prv.data,input);
if (SC(Q,prv))                a5:L   TRUE(SC(Q,prv))
  prv = m;                    a6:B   prv = m;
  break;                      a7:B   break;
```

8.6 Applications

This chapter demonstrates the applicability of our analysis to three non-trivial non-blocking algorithms from the literature. One is the illustration example; the other two are presented in this section. Although in two cases we must modify the algorithm before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatic) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions.

```
void Enq(int value)
 local node = new Node();
   node.value = value;
   node.next = null;
   loop
     local t = LL(Tail) in
       local next = LL(t.Next) in
         if (!VL(Tail)) continue;
         if (next != null)
           SC(Tail,next);
           continue;
         if (SC(t.Next,node))
           // optional
           [SC(Tail,node);]
           return;
```

```
int Deq()
 loop
   local h = LL(Head) in
     local next = h.Next in
       if !VL(Head) continue;
       if (next == null)
         return EMPTY;
       if (h == LL(Tail))
         SC(Tail,next);
         continue;
       local value = next.Value in
         if (SC(Head,next))
           return value;
```

Figure 8.2: Michael and Scott's Non-Blocking FIFO Queue (NFQ). `Head` and `Tail` are global variables.

8.6.1 Michael and Scott's Non-Blocking FIFO Queue Using LL/SC/VL

8.6.1.1 NFQ and NFQ′

Figure 8.2 contains code for a non-blocking FIFO queue (NFQ) that uses LL/SC/VL [33]. It is similar to the well-known CAS-based algorithm in [35]. It uses a singly-linked list whose head and tail are pointed to by global variables `Head` and `Tail`. Enqueue consists of three main steps: create a node, add it to the end of the list, and update `Tail`. A blocking implementation would use a lock around the second and third steps to achieve atomicity. In the non-blocking algorithm, if a thread gets delayed (or killed) after the second step, other threads may update `Tail` on its behalf; in that case, if the delayed thread later tries to update `Tail`, its SC will harmlessly fail. To avoid blocking, the dequeue operation also updates `Tail`. Dequeue is also non-blocking. A dummy node is used as the head of the queue to avoid degenerate cases. The code for `Deq` in [33, 35] stores the value of `LL(Tail)` in a local variable; the code in Figure 8.2 does not. This does not affect the correctness or performance of the algorithm but makes it easier to analyze.

We would like to show that NFQ is linearizable, using the two-step approach described in Section 8.2: one step is to show that the concurrent implementation executed sequentially satisfies the sequential specification; the other step is to apply our analysis to show that the procedures of the implementation are atomic.

An obstacle to apply our atomicity analysis to NFQ is that the loops in `Enq` and `Deq` are not pure, because of the updates to `Tail` in normal iterations. Therefore, we modify the program to make the loops pure before applying our analysis algorithm; specifically,

```
void AddNode(int value)
  local node = new Node() in
    node.Value = value;
    node.Next = null;
    loop
      local t = LL(Tail) in
        local next = LL(t.Next) in
          if !VL(Tail)
            continue;
          if (next != null)
            continue;
          if SC(t.Next,node)
            return;
```

```
void UpdateTail()
  loop
    local t = LL(Tail) in
      local next = t.Next in
        if !VL(Tail)
          continue;
        if (next != NULL)
          SC(Tail,next);
          return;
```

```
int Deq'()
  loop
    local h = LL(Head) in
      local next = h.Next in
        if (!VL(Head))
          continue;
        if (next == null)
          return EMPTY;
        if (h == LL(Tail))
          continue;
        local value = next.Value in
          if (SC(Head,next))
            return value;
```

Figure 8.3: NFQ′, a modified version of NFQ.

we consider the modified program NFQ′ in Figure 8.3, and we prove in Appendix B.1 that the modification preserves linearizability. In NFQ′, all updates to `Tail` are performed in a separate procedure `UpdateTail`. `UpdateTail` may be invoked (by the environment) at any time, so NFQ′ is effectively more non-deterministic than NFQ.

8.6.1.2 Atomicity of NFQ′

All exceptional variants for the procedures of NFQ′ are listed in Figure 8.4. Each line of code is labeled on the left with a line number and the atomicity type of the code on that line. A line may contain multiple actions; we refer to the sequential composition of their atomicity types as the atomicity type of the line. Next we describe how the atomicity analysis algorithm in Section 8.5.4 works on these procedures.

In step 1, a1, a2, a3, a7, a9, b4, b6, c4, c5, d4 and d8 are classified as both-movers,

```
void AddNode(int value)
a1:B   local node = new Node() in
a2:B     node.Value = value;
a3:B     node.Next = null;
a4:R     local t = LL(Tail) in
a5:R       local next = LL(t.Next) in
a6:B         TRUE(VL(Tail));
a7:B         TRUE(next == null);
a8:L         TRUE(SC(t.Next,node));
a9:B         return;
```

```
void UpdateTail()
b1:R   local t = LL(Tail) in
b2:R     local next = t.Next in
b3:B       TRUE(VL(Tail));
b4:B       TRUE(next != NULL);
b5:L       TRUE(SC(Tail,next));
b6:B       return;
```

```
int Deq'1()
c1:R  local h = LL(Head) in
c2:A    local next = h.Next in
c3:L      TRUE(VL(Head));
c4:B      TRUE(next == null);
c5:B      return EMPTY;
```

```
int Deq'2()
d1:R  local h = LL(Head) in
d2:R    local next = h.Next in
d3:B      TRUE(VL(Head));
d4:B      TRUE(next != null);
d5:A      TRUE(h != LL(Tail));
d6:B      local value = next.Value in
d7:L        TRUE(SC(Head,next));
d8:B        return value;
```

Figure 8.4: Exceptional variants for procedures of NFQ′.

because they access local variables.

In step 2, a4, a5, b1, c1 are d1 are classified as right-movers, because they are matching LLs for successful SCs or VLs; a6 (which is reclassified as a both-mover in step 4), a8, b5, c3, and d7 are classified as left-movers because they are successful SCs or VLs; b3 and d3 are classified as both-movers because they are between matching LLs and successful SCs.

In step 3, the local condition for a5-a9 and c2-c5 is $next == null$. The local condition for b2-b6 and d2-d8 is $next\,! = null$.

Now consider step 4. Let $\texttt{t}_a$ and $\texttt{t}_u$ denote the local variable `t` in `AddNode` and `UpdateTail`, respectively. If $\texttt{t}_a$.`Next` of the LL-SC block in `AddNode` is aliased with $\texttt{t}_u$.`Next` of the local block in `UpdateTail`, then according to Theorem 8.5.5, the update on `Tail` (*i.e.*, b5) cannot happen between a6 and a7, so a6 is a both-mover. a8 cannot happen between b2 and b3, so b2 is a right-mover. Suppose $\texttt{t}_a$.`Next` is not aliased with $\texttt{t}_u$.`Next`; this implies $\texttt{t}_a$ is not aliased with $\texttt{t}_u$, *i.e.*, $\texttt{t}_a \neq \texttt{t}_u$, so even if a8 happens between b2 and b3, b2 is a right-mover by Theorem 8.3.3. $\texttt{t}_a \neq \texttt{t}_u$ implies that the value of `Tail` read in line of a4 in `AddNode` is not equal to the value of `Tail` read in line of b1 in `UpdateTail`. Thus, even if b5 happens between a6 and a7, a6 is still a both mover by Theorem 8.3.3. For d2, if `h.Next` is aliased with `t.Next` of `AddNode`, a8 cannot happen between d2 and d3 according to Theorem 8.5.5, hence d2 is a right-mover

program	without atomic		with atomic	
	states	time	states	time
unbounded `AddNode` threads	4500	$>$ 19h	13	3.0s
unbounded `Deq'` threads	1285	88 m	10	1.7s
incorrect `AddNode`	13	5 s	13	3.0s

Table 8.2: Experimental results for verification of NFQ′ with TVLA.

by Theorem 8.3.3; if `h.Next` is not aliased with `t.Next`, d2 is again a right-mover. Also in step 4, d6 is inferred to be a both-mover, because there is no write to the `Value` field of any shared object; the only write a2 to the `Value` field is on an object that has not escaped.

In step 5, the unclassified c2 and d5 are given atomicity type A. Step 6 infers that each procedure in Figure 8.4 has atomicity type A. Step 7 infers that all procedures in NFQ′ are atomic.

8.6.1.3 Linearizability of NFQ′ and NFQ

We showed in Section 8.6.1.2 that the procedures in NFQ′ are atomic, and we showed in Appendix B.1 that NFQ′ can simulate all behaviors of NFQ. To conclude that NFQ′, and hence NFQ, are linearizable with respect to a sequential specification of FIFO queues, we need to show that NFQ′ executed sequentially satisfies that specification. One approach is to use a powerful verification tool such as TVLA [57], which is a model checker based on static analysis. With our approach, TVLA only needs to consider sequential executions of NFQ′, so the verification is much faster and use much less memory than the verification in [57], where TVLA was used to show directly that NFQ satisfies some complicated temporal logic formulas.

To evaluate the speedup that our atomicity analysis can provide for subsequent verification, we used TVLA to verify several correctness properties of NFQ′, similar to the properties in [57, Table 2]. We analyzed the correct program with two different environments: in the first one, the number of threads that concurrently call `AddNode` is unbounded (there is only one thread that performs dequeues, and there is only one `UpdateTail` thread, since it contains a non-terminating loop); in the second one, the number of threads that concurrently perform dequeues is unbounded (there is only one thread that performs `AddNode`, and one that calls `UpdateTail`). We also checked the properties for an incorrect version of NFQ′; specifically, we deleted the statement `if (next != null) continue` in the `AddNode` procedure; TVLA catches this error. We performed all experiments twice: once with each procedure body declared as atomic, as inferred by our analysis algorithm, and once without those declarations. The atomicity declarations had little effect on the time needed for TVLA to find an error in the incorrect program, but it reduced the time and space needed to verify the correct versions by a factor of 100 or more. The experimental results appear in Table 8.2.

8.6.2 Gao and Hesselink's Non-Blocking Algorithm for Large Objects

For large objects, copying is the major performance bottleneck in Herlihy's algorithm. Gao and Hesselink [21] proposed an algorithm that avoids copying the whole object. The fields of each object are divided into disjoint groups such that each operation changes only fields in one group. When copying data between the shared and private copies of an object, only the modified groups are copied. To efficiently detect modifications, a version number is associated with each group of fields of each copy of the object. The algorithm works as follows: (1) read the shared object reference using LL; (2) copy data and version numbers in all modified groups of fields of the currently shared copy of the object into the corresponding groups of fields of the current thread's private copy; (3) do the computation on the private copy, updating fields in some group and incrementing the corresponding version number; (4) switch the references between the shared object and the private object using SC. The algorithm is more complicated than Herlihy's algorithm for small objects in Figure 8.1 mainly because of the loop over groups of fields, the conditional behavior depending on which groups of fields changed, and the use of version numbers to efficiently detect changes.

Our analysis cannot directly show that the algorithm is atomic, due to the use of version numbers. Our analysis algorithm is able to show that a version of the algorithm that does not use version numbers is atomic. We then show that the transformations that optimize the algorithm by introducing and using version numbers preserve atomicity; this is relatively easy.

We show that the non-blocking algorithm for large objects (specifically, algorithm 3 in Figure 8.6) is atomic. For clarity, we use `continue` in the pseudo-code, even though SYNL does not have `continue`. It is easy to rewrite the program to eliminate the `continue`, with no significant effect on the atomicity analysis. The procedure call `copy(prvObj.data[i],m.data[i])` copies the data in `m.data[i]` to `prvObj.data[i]`. The procedure `compute(prvObj,g)` does computation based on the data in `prvObj` and writes the result into `prvObj.data[g]`.

Algorithm 3 in Figure 8.6 differs in some minor ways from the original algorithm in [21]. It does not contain the redundant array `old` used in [21]. Like Herlihy's algorithm in Figure 8.1, it uses VL (line a13) to prevent errors due to inconsistent states of `prvObj` that may result from updates during copying (line a8). [21] simply assumed that such errors do not occur. Also, we omit the guard predicate used in [21] to optimize cases where `compute` is applied in a state in which it performs no updates.

Algorithm 1 in Figure 8.5 is a simplified version of the algorithm in which all data of the shared object (*i.e.*, `m`) are copied into the working object (*i.e.*, `prvObj`) of the current thread in every iteration of the outer loop. All field accesses through `prvObj` are local actions, because `prvObj` contains a primary reference. Moreover, `prvObj.data[i]` is dead at the end of the outer loop's body under all normal terminations. Therefore, the outer loop is pure. By the same reasoning as for the non-blocking algorithm for small objects in Figure 8.1, the procedure of Algorithm 1 in Figure 8.5 is atomic.

```
Algorithm 1
 void proc(Object SharedObj, int g)
 a1     loop
 a2         local m = LL(SharedObj) in
 a3         local i = 1 in
 a4         loop
 a5             if (i>W) break;
 a6             copy(prvObj.data[i],m.data[i]);
 a7             if (!VL(SharedObj))
 a8                continue a2;
 a9             i++;
 a10        if (!VL(SharedObj)) continue a2;
 a11        compute(prvObj,g);
 a12        if (SC(SharedObj,prvObj))
 a13            prvObj = m;
 a14            return;

Algorithm 2
 void proc(Object SharedObj, int g)
 a1     loop
 a2         local m = LL(SharedObj) in
 a3         local i = 1 in
 a4         loop
 a5             if (i>W) break;
 a6             if (prvObj.data[i] != m.data[i])
 a7                 copy(prvObj.data[i],m.data[i]);
 a8                 if (!VL(SharedObj))
 a9                    continue a2;
 a10            i++;
 a11        if (!VL(SharedObj)) continue a2;
 a12        compute(prvObj,g);
 a13        if (SC(SharedObj,prvObj))
 a14            prvObj = m;
 a15            return;
 a16        //else continue a2;
```

Figure 8.5: Gao and hesselink's non-blocking algorithm for large objects: algorithms 1 and 2.

Algorithm 2 in Figure 8.5 is an improved version of Algorithm 1 in which the copy is omitted from `m.data[i]` to `prvObj.data[i]` when those two locations already contain the same value. Algorithm 2 clearly has the same behavior as Algorithm-1. Therefore, the procedure in Algorithm 2 is atomic.

Algorithm 3 in Figure 8.6 is an improved version of Algorithm 2 in which version numbers are used to efficiently and conservatively check whether `m.data[i]` and `prvObj.data[i]` are equal. "Conservatively" here means that the check might return false when they contain the same value (*e.g.*, because the values stored in `m.data[i]`

```
Algorithm 3: Full Algorithm with Modification
 void proc(Object SharedObj, int g)
 a1     loop
 a2         local m = LL(SharedObj) in
 a3         local i = 1 in
 a4         loop
 a5             if (i>W) break;
 a6             local newVersion[i] = m.version[i] in
 a7                 if (newVersion[i] != prvObj.version[i])
 a8                     copy(prvObj.data[i],m.data[i]);
 a9                     if (!VL(SharedObj))
 a10                        continue a2;
 a11                    prvObj.version[i] = newVersion[i];
 a12            i++;
 a13        if (!VL(SharedObj)) continue a2;
 a14        compute(prvObj,g);
 a15        prvObj.version[g]++;
 a16        if (SC(SharedObj,prvObj))
 a17            prvObj = m;
 a18            return;
 a19        else
 a20            prvObj.version[g] = 0;
```

Figure 8.6: Gao and hesselink's non-blocking algorithm for large objects: algorithm 3.

and `prvObj.data[i]` happen to be equal), but this merely causes the code in the full algorithm to do an unnecessary copy (*i.e.*, the copy does not actually change the value of `prvObj.data[i]`). The last statement `prvObj.version[g] = 0` is needed so that the update to `prvObj.version[g]` from line a15 will be discarded if the SC fails. Algorithm 3 clearly has the same behaviors as Algorithm 2. Therefore, the procedure in Algorithm 3 is atomic.

To evaluate the benefit of our atomicity analysis compared to a traditional partial-order reduction, we implemented Algorithm 3 in the model checker SPIN [27]. We wrote a driver with 3 threads that concurrently invoke arithmetic operations on a shared object with 3 integer fields, each in its own group. The input files are available at `http://www.cs.sunysb.edu/~liqiang/nonblocking.html`. The numbers of reachable states are: 4,069,080 with no optimization; 452,043 with SPIN's built-in partial-order reduction; 69,215 with the procedure body declared as atomic, as inferred by our analysis algorithm; and 4619 with both optimizations.

8.7 Conclusions

This chapter presents a static analysis to infer atomicity of code blocks in programs with non-blocking synchronization. Although we need to modify the program before applying

our analysis in two out of the three examples, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. Theorem-proving approaches, such as [21], can verify atomicity of the original programs but require much more manual effort than our approach, even if our analysis algorithm is applied manually. Our analysis significantly reduces the number of states considered during subsequent analysis and verification.

Chapter 9

Hybrid Analysis

9.1 Introduction

Runtime checking cannot guarantee absence of errors (such as deadlocks, data races or atomicity violations) in all executions of the program. Runtime checking also incurs a significant overhead. On the other hand, runtime checking generally produces fewer false alarms than static analysis; this is a significant practical advantage, since diagnosing all of the warnings from static analysis of large codebases may be prohibitively expensive.

Type systems can statically ensure race-freedom and atomicity [7, 14, 19, 43]. Type systems can ensure that methods are race-free or atomic in all possible executions. However, some aspects of a program's behaviors, such as happen-before relations due to start-join on threads, are harder to analyze statically than dynamically, and are not considered by the type system because of false alarms (*i.e.*, type errors for methods that are atomic). Moreover, type inference for type systems is NP-complete [16, 17], so type system may require manual annotation of the program. This is a significant burden, especially for legacy code.

Type inference reduces the annotation burden by automatically determining types for all or parts of a program. Unfortunately, complete (*i.e.*, inferring types for all typable programs) type inference is NP-complete [16, 17]. This motivates the development of incomplete type inference algorithms. Type discovery is an inexpensive approach to type inference that employs both runtime monitoring and static analysis to infer types for all or part of a program. Type discovery is incomplete but experience shows it is very effective in practice, discovering 98% of the annotations in the experiments described in [1].

This chapter describes the use of static analysis to significantly decrease the overhead of runtime checking [43, 2, 3]. First, type discovery is used to discover types for all or part of the program. The discovered types are then given to the typechecker, which issues warnings. Runtime deadlock, race or atomicity checking is then focused on fragments of code for which the type checker issued warnings. The approach is completely automatic, scalable to very large programs, and significantly reduces the overhead of runtime checking for data races and atomicity violations. Although type discovery requires running an instrumented program, the cost is much less than full runtime checking, because sampling

(*e.g.*, monitoring a few instances of each class) suffices for type discovery; furthermore, the cost of type discovery is amortized across all subsequent testing in which the discovered types are used to reduce the overhead of runtime checking.

9.2 Hybrid Analysis of Data Races

9.2.1 Type System for Race-Freedom

In Parameterized Race Free Java (PRFJ) [7] as in its predecessor Race Free Java [14], types are extended to indicate the synchronization discipline (also called "protection mechanism" or "owner") used to co-ordinate accesses to each object. To allow different instances of a class to use different protection mechanisms, each class is parameterized by formal owner parameters which may be instantiated with other formal owner parameters, final expressions (*i.e.*, expressions whose value does not change) representing locks, or special owners (described below). The first owner parameter of each class indicates the owner of the `this` object; the other owner parameters are used to propagate ownership information to the object's fields and methods.

A final expression used as an owner specifies a lock that must be held when the object is accessed. There are four special owners: `thisThread`, `self`, `readonly` and `unique`. `readonly` indicates that the object is readonly and cannot be updated. `unique` means that there is a unique reference to the object. `thisThread` means that the object is thread-local (*i.e.*, unshared). `self` means that the object is protected by its own lock (*i.e.*, a self-synchronized object). The owner of an object is said to *guard* all of its fields.

Method declarations may have a `requires` clause that contains a set of final expressions; the locks on the owners of these expressions must be held when the method is invoked.* The special owners `thisThread`, `unique` and `readonly` are always assumed to be in the lockset. PRFJ ensures that whenever a field of an object is accessed, either the object is readonly, or the accessing thread either has a unique reference to the object or holds the lock on the root owner of the object, thus avoiding races.

9.2.2 Discovery of Race-Free Types

The type discovery algorithm for race-free types has three main steps. First, the target program is instrumented by an automatic source-to-source transformation and executed on test inputs. The instrumented program monitors accesses to fields of certain objects of each class and writes a log containing relevant information: which locks were held when the object was accessed, whether multiple threads accessed the object, etc. Second, the information in the log file is used to infer owners for fields, method parameters and return values, and owners in class declarations. Third, the intra-procedural type inference algorithm in [7] is used to infer the owners in the types of local variables and in the types of

*For simplicity, we ignore the distinction between owners and root owners in this overview.

allocation sites whose owners have not already been determined. Local type-inference has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions.

Type discovery is not guaranteed to produce correct typings for all typable programs, but experience shows that it is very effective in practice. For example, 98% of the race-free types were automatically discovered in the experiments described in [1].

9.2.3 Integrating Static Analysis and Runtime Checking

This section presents a technique for utilizing the warnings from the type checker to identify parts of the program from which runtime race checking can safely be omitted; in other words, we focus runtime checking on parts of the code that may contain races. We developed three approaches: *field-based*, *method-based* and *combined-field-and-method based*.

Field-based focused runtime checking works as follows. Given a program annotated (in whole or part) with (possibly incorrect) PRFJ types, we run the PRFJ typechecker and compute, based on the warnings it issues, a list of fields that might potentially be involved in data races. We refer to these as *race-unsafe* fields. The other fields are guaranteed to be race-free, *i.e.* accesses to them cannot be involved in data races. Thus, during runtime race detection, there is no need to monitor accesses to race-free fields.

Method-based focused runtime checking works as follows. Type discovery is effective only for methods executed in the test suite during type discovery. Executed methods are likely to be well-typed, *i.e.*, there are no errors in the types discovered for them. But many methods will be "untyped", either because there is an error in the discovered types (possibly because the method is untypable), or because the methods were not executed hence no types were discovered for them, or because source code for them is unavailable. The method-based approach is to perform runtime checking on all accesses in untyped methods and omit it from all accesses in well-typed methods. This can potentially perform better than the field-based approach, because the test suite used for type discovery typically exercises the most frequently executed methods; if these methods are well-typed, then the method-based approach succeeds in eliminating runtime checking in places where this provides the most benefit.

The combined field- and method-based approach exploits the observation that not all field accesses in untyped methods need to be monitored. Specifically, only race-unsafe fields need to be monitored in untyped methods. Furthermore, at call sites to typed methods in untyped methods, checking whether arguments' owners or attributes conform to the declared owners of the corresponding method parameters is necessary only for parameters whose types are race-unsafe classes.

9.2.4 Experiments with the Focused Runtime Race Checking

The conclusion of the experiments are summarized here, details appear in [2]. The field-based approach does not introduce any additional checking (compared to the pure runtime race detection), so field-based focused checking is always at least as fast as full checking, and combined focused checking is always at least as fast as method-based focused checking. The method-based approach introduces additional checking at the boundary between untyped and typed code, *i.e.*, at calls to typed methods from untyped methods. This can make method-based focused checking slower than full checking. There is little correlation between the percentage of fields classified as race unsafe or methods classified as untyped and the speedups achieved, because different fields and methods may be used with much different frequencies.

The field-based approach outperformed the method-based approach in all of the smaller benchmarks, *i.e.*, all except Jigsaw. This is not surprising, since most of the methods in them are exercised by the sample inputs, and most of the fields are classified as race-free by the typechecker. For Jigsaw, the average speedups for the field-based, method-based, and combined approaches are 21%, 68%, and 72%, respectively.

9.3 Hybrid Analysis of Atomicity

9.3.1 Atomicity Types

Flanagan and Qadeer's type system for atomicity [19] extends a race-free type system [14] to associate an atomicity with each expression and statement (for brevity, "expression" means "expression or statement" in the rest of this section). The atomicity of each method is declared in the program; atomicities of other expressions are implicit. An atomicity is a *basic atomicity* or a *conditional atomicity*. The basic atomicities and their meanings are: `const`: evaluation of the expression does not depend on or change any mutable state; `mover`: the expression left-commutes with every operation of another thread that could occur immediately before it and right-commutes with every operation of another thread that could occur immediately after it (*i.e.*, the two operations can be swapped, and this still leads to the same state); `atomic`: evaluation of the expression is always equivalent to evaluation of the expression without interleaved actions of other threads; `cmpd` (compound): none of the preceding atomicities apply; `error`: evaluation of the expression violates the locking discipline specified by the race-free types.

Conditional atomicities are used when the atomicity of an expression depends on which locks are held by the thread evaluating it. A conditional atomicity $l\,?\,a : b$ is equivalent to atomicity a if lock l is held when the expression is evaluated, and is equivalent to atomicity b otherwise. $l\,?\,a$ abbreviates $l\,?\,a : \texttt{error}$.

Let α and a, b range over basic atomicities and atomicities, respectively. Each atomicity a is interpreted as a function $[\![a]\!]$ from the set ls of locks currently held to a basic atomicity: $[\![\alpha]\!]\,(ls) = \alpha$ and $[\![l\,?\,a_1 : a_2]\!]\,(ls) = \text{if } l \in ls \text{ then } [\![a_1]\!]\,(ls) \text{ else } [\![a_2]\!]\,(ls)$. A partial

order $\sqsubseteq$ on atomicities is defined. The ordering on basic atomicities is `const` $\sqsubseteq$ `mover` $\sqsubseteq$ `atomic` $\sqsubseteq$ `cmpd` $\sqsubseteq$ `error`. The ordering on conditional atomicities is the pointwise extension of the ordering on basic atomicities, *i.e.*, $a \sqsubseteq b$ iff $\forall ls : [\![a]\!]\,(ls) \sqsubseteq [\![b]\!]\,(ls)$. Rules for effectively determining the ordering on atomicities appear in [19].

The typing rules express the atomicity of an expression in terms of the atomicities of its subexpressions using five operations on atomicities: sequential composition $a; b$, iterative closure a^*, join $a \sqcup b$ (based on the partial order $\sqsubseteq$ described above), conditional $l\,?\,a : b$, and the operation $S(l, a)$ described below. Sequential composition for basic atomicities is defined by: $\alpha_1; \alpha_2$ equals `cmpd` if α_1 and α_2 are both `atomic`, and equals $\alpha_1 \sqcup \alpha_2$ otherwise. The iterative closure a^* denotes the atomicity of an expression that repeatedly executes an expression with atomicity a. For basic atomicities, it is defined by: a^* equals `cmpd` if a is `atomic`, and equals a otherwise. Sequential composition and iterative closure for conditional atomicities are defined as follows [19].

$$(l\,?\,a : b)^* = l\,?\,a^* : b^*$$
$$(l\,?\,a_1 : a_2); b = l\,?(a_1; b) : (a_2; b)$$
$$\alpha; (l\,?\,b_1 : b_2) = l\,?(\alpha; b_1) : (\alpha; b_2)$$

Our atomicity type system is called Extended Parameterized Atomic Java (EPAJ) [43]. It combines Flanagan and Qadeer's atomicity types with a more expressive race-free type system, which extends PRFJ to allow a different owner for each field of an object.

9.3.2 The Focused Reduction-Based Algorithm

We use types to focus the runtime reduction-based algorithm. Chapter 4 describes two reduction-based algorithms: online (*i.e.*, atomicity is checked on-the-fly as the program executes) and offline (*i.e.*, atomicity is checked after execution of the program). The online algorithm avoids the overhead of storing and retrieving data but may miss atomicity violations by misclassifying an access to a field as race-free, since a subsequent access might lead to a possible race. The offline algorithm augments the online algorithm by incorporating dynamic escape analysis and start-join analysis. Therefore, the offline algorithm is more precise, but slower than the online algorithm.

As described in Section 9.2.3, applying the EPAJ typechecker to discovered types produces warnings from which we can compute a list of fields that are not involved in a data race (race-free fields). The type-checker can also list the methods that are verified to be atomic. We focus the reduction-based algorithm by monitoring only the fields that are not verified to be race-free by the type-checker and by analyzing atomicity only for methods that are not verified to be atomic by the type-checker.

9.3.3 Experiments with the Focused Reduction-Based Algorithm

The results of our experiments are summarized in Table 9.1. "Base" gives the execution time of the uninstrumented benchmark. "Online Slowdown" and "Offline Slowdown" give

Benchmark	Base (sec)	Slowdown			
		Online	Offline	OptOnl	OptOffl
elevator	0.2	1.25	3.30	1.25	1.5
tsp	1.8	119.17	421.11	2.56	2.57
moldyn	25.49	22.97	73.88	1.73	4.37
raytracer	13.88	111.07	45.82	6.87	6.31
montecarlo	16.07	8.47	30.42	1.12	1.12
hedc	0.53	1.13	1.89	1.04	1.51
median	7.84	15.72	38.12	1.49	2.09

Table 9.1: Optimization of runtime atomicity analysis.

the slowdown for the online and offline reduction-based algorithms compared to the Base time. "OptOnl Slowdown" and "OptOffl Slowdowm" are similar but reflect the effect of the above optimization. The table demonstrates that the optimization reduces the median slowdown for the online algorithm from 15.7 to 1.5 and the median slowdown for the offline algorithm from 38.1 to 2.1.

Type discovery uses a lockset algorithm, but the overhead for type discovery is low (less than 20%) because it monitors only a sampling of objects. Furthermore, types discovered after running a program once can be used to focus runtime atomicity checking for an entire test suite, making the amortized cost of type-discovery negligible.

9.3.4 The Focused Block-Based Algorithm

This section presents an approach to focus the block-based algorithm using atomicity types automatically produced by type discovery and type inference. The EPAJ type-checker lists all methods that it has verified to be atomic. The list contains well-typed methods with a basic atomicity less than or equal to `atomic` and methods with a conditional atomicty that simplifies at all call sites to a basic atomicity less than or equal to `atomic`. We focus the block-based algorithm by reducing the number of blocks constructed using events in executions of those methods. Note that the accesses in those methods cannot be completely ignored because they may participate in forming unserializable patterns with events from other methods.

For a transaction t, the focused block-based algorithm does not directly construct blocks from pairs of events in t. Instead, it records a few items characterizing the accesses performed by t, and uses that information as described below. For each escaped variable x accessed in t (a dynamic escape analysis is introduced in Section 3.1), it records whether t writes (and possibly reads) x or merely reads x. It also records the set $held(t)$ of locks held at any point during t (*i.e.*, the set of locks held when t starts or acquired at any time during t). This information is stored for all transactions in a global table. The focused block-based algorithm treats each transaction as if it consisted of the following events: for each escaped variable accessed by t, a single write or read (depending on whether t wrote

x) event protected by the locks in $held(t)$. These reduced (*i.e.*, relatively small) sets of events for the transactions are used in the following three ways.

First, for each 1v-block from a possibly non-atomic unit, the focused algorithm checks whether an unserializable pattern can be formed from the two events represented by the 1v-block and one event from the reduced set of events of a transaction. Specifically, information about the locks held is used to determine whether the latter event can occur between the former two, and if so, whether the resulting pattern of three reads and writes matches an unserializable pattern, *i.e.*, the middle event does not commute with the first and last events.

Second, for a possibly non-atomic unit u that contains calls to atomic methods, the focused block-based algorithm treats those method calls as if they consisted of the reduced sets of events described above, *i.e.*, 1v-blocks and 2v-blocks for u are constructed from the events in those sets and the other events in t, except that 1v-blocks are not constructed from two events from the same call to an atomic method. Note that the original block-based algorithm processes events in method calls in a transaction in exactly the same way as events in the top-level method call in the transaction; the block-based algorithm does not incorporate a concept of nested transactions.

Third, 2v-blocks are constructed from the reduced set of events for each transaction. Thus, the focused block-based algorithm may construct 2v-blocks from two events from the same transaction, but it never constructs 1v-blocks from two events from the same transaction.

Now we sketch a proof that treating $held(t)$ as the set of locks held at each access in a transaction t does not affect the result of the block-based algorithm. The proof relies on the structure of the EPAJ type system, specifically, the fact that a transaction statically verified as atomic does not acquire any lock after releasing a lock. It also relies on the use of block-structured (*i.e.*, properly nested) locking as in Java, and the assumption that there is no potential for deadlock. Section 9.4 presents an algorithm to detect potential for deadlock. Therefore, it suffices to show that our optimization is correct in the absence of potential for deadlock.

Let $held(e)$ be the locks held by the thread that executes event e when e occurs. Let $held(s)$ be the locks ever held by the thread during execution of a sequence s of events. For a block b, let $h_{12}(b)$ denote the set of locks held continuously from the first event to the second event of b, and let $hmid(b)$ denote the set of locks acquired and released in t between the two events of b. An event e can happen inside a block b if $held(e) \cap h_{12}(b)$. A block b can happen inside another block b' if both events of b can happen inside b', $h_{12}(b) \cap h_{12}(b') = \emptyset$, and $hmid(b) \cap h_{12}(b') = \emptyset$.

The execution of an atomic method is called an atomic sequence. All events of an atomic sequence can be moved together to some place without changing the resulting state. If the place is an event e_x that accesses a variable x, obviously, we can replace $held(e_x)$ with $held(s)$ without affecting the atomicity analysis. If the place is not e_x, we need to prove that this replacement does not affecting the atomicity analysis.

Theorem 9.3.1. *Suppose that a set T of transactions has no potential for deadlock. For an*

event e_x to variable x in an atomic sequence s in transaction $t \in T$, it does not affect the atomicity analysis for other transactions in the focused block-based algorithm if $held(e_x)$ is replaced by $held(s)$.

Proof. If e_x is a non-mover, all events of s can swapped together. Obviously, we can replace $held(e_x)$ with $held(s)$ in this case without affecting the atomicity analysis.

If e_x is not a non-mover, according to the algorithm of EPAJ, it must be a both mover. e_x is a both mover if (1) all events to x are read operations in the program; or (2) all events to x hold a common lock. For the first case, any 1v-block or 2v-block which contains events to x does not violate atomicity, hence, the theorem also holds.

Now we prove the theorem holds for the second case. Let $L = held(x)$. Note that $L \subseteq held(s)$. If $L = held(s)$, obviously, the theorem holds. In the following, we prove that the theorem holds if $L \subset held(s)$. Let L' = $held(s) - L$. According to the atomicity pattern for types (*i.e.*, $R^*N^?L^*$), the scope of L' must be totally inside the scope of L. Suppose that there are two blocks b and b', where b is built from e_x and another event e_y, and b' is built from two events e'_x and e'_y in another transaction t'. Note that x and y may denote the same variable. The following explanation is applicable for 1v-block and 2v-block.

Now we prove that the replacement does not affect the atomicity checking between b and b'. Let $L_1 = held(e'_x) \cap L$ and $L_2 = held(e'_x) \cap L'$. According to the above discussion, $held(e'_x)$ must also contain some lock in L, *i.e.*, $L_1 \neq \emptyset$. If $L_2 = \emptyset$, it is easy to show that the theorem holds. Suppose $L_2 \neq \emptyset$. Because the scope of L' is inside the scope of L in t, the scope of L_1 is inside the scope of L_2 in t. Thus, the scope of L_2 must be inside the scope L_1 in t', otherwise, there will be potential for deadlock. Recall that we assume locking is block-structured. Therefore, for any lockset, if it contains L_2, L_1 must be also contained in it. In the following, we prove that if b can happen inside b', after extending L to $L \cup L'$, b can still happen inside b'. For the other cases, the proofs are similar.

If b can happen inside b', according to the condition discussed above, we have $held(e_x) = L$, $held(e_x) \cap h_{12}(b') = \emptyset$, $held(e_y) \cap h_{12}(b') = \emptyset$, $h_{12}(b) \cap h_{12}(b') = \emptyset$, and $hmid(b) \cap h_{12}(b') = \emptyset$. Extending L to $L \cup L'$ affects $held(e_x) = L \cup L'$, h_{12}, may affects $h_{12}(b)$ by adding L', and my affects $hmid(b)$ by adding or removing L'. If the above conditions are changed by these modification, the corresponding lockset of b' must contain $L_2 \subseteq L'$. Thus, L_1 will also be contained the lockset. This contradicts with these original conditions. □

9.3.5 Experiments with the Focused Block-Based Algorithm

We evaluated the focused block-based atomicity checking algorithm on the benchmarks described in the previous section except Jigsaw, because Soot failed to construct the call graph for it. The call graph is needed to compute the list of race-free fields. The results appear in Table 9.2. Running times are measured in seconds on a 1GHz Sun Blade 1500 with Sun JDK 1.4 and are the average over five runs. Base Time is the running time of the original program. Intcpt Ovhd is the overhead of intercepting events, *i.e.*, the increase in

Program	Base Time	Intcpt Ovhd	Unopt Ovhd	Foc Ovhd	Spdup	Frac Ovhd
elevator	0.2	0.14	0.32	0.30	8.6%	46.5%
tsp(12)	0.3	9.59	8.89	8.49	4.5%	46.2%
tsp(14)	0.48	17.06	387.0	393.0	-1.5%	95.7%
hedc	0.6	0.22	0.64	0.55	15.0%	40.0%
moldyn	44.03	1430	172.7	129.5	25.0%	8.1%
montecarlo	15.85	443.2	10.3	0	100%	0%
raytracer	14.34	594	44.8	24.2	46.0%	3.8%

Table 9.2: Comparison of running time between the focused algorithm and the block-based algorithm.

running time when all events relevant to atomicity checking (field accesses, method calls, synchronized statements, etc.) are intercepted but not processed; *i.e.*, code is inserted to call a method with arguments describing the event, and that method simply returns. Unopt Ovhd is the cost of the unoptimized block-based algorithm, *i.e.*, the increase in running time relative to the version that intercepts events without processing them. Foc Ovhd is the cost of the focused block-based algorithm, measured the same way. The same code is used to intercept events for the unoptimized and focused versions; our current optimization only affects the cost of processing those events. Spdup is (Unopt Ovhd - Foc Ovhd) / Unopt Ovhd. The average speedup is about 32%, *i.e.*, the cost of the block-based algorithm is reduced by about one third.Frac Ovhd is the overhead of the focused block-based algorithm as a fraction of the total running time, *i.e.*, Foc Ovhd / (Base Time + Intcpt Ovhd + Foc Ovhd).

However, in some benchmarks, the overhead of intercepting events exceeds the overhead of the block-based algorithm itself, so an important direction for future work is to avoid intercepting some events. The focused block-based algorithm provides good opportunities for this. Using that algorithm, for each execution of a method classified as atomic by the type checker, for each shared variable it accesses, it is sufficient to intercept one write event or, if there is none, one read event. Thus, if static analysis can be used at instrumentation time to determine, for example, that expression e reads the same field of the same object that statement s writes, and that s is executed whenever e is executed (*i.e.*, e dominates s in the control flow graph), then e does not need to be instrumented.

The focused reduction-based algorithm completely ignores events in methods shown by the typechecker to be atomic. This greatly reduces the interception overhead as well as the cost of the reduction-based algorithm itself, reducing the median overall slowdown from 38.1 to 1.5 in the experiments. The opportunity for this more drastic improvement is related to the fact that the reduction-based algorithm is less accurate (*i.e.*, more conservative, producing more false alarms) than the block-based algorithm.

9.4 Hybrid Analysis of Deadlocks

This section presents a runtime algorithm for detecting potential deadlocks, a deadlock type system, and the technique to focus the runtime detection using the analysis result of the type system [3].

9.4.1 Runtime Detection of Potential Deadlocks

The GoodLock algorithm [24] detects potential deadlocks at runtime. It records a runtime lock tree for each thread. The runtime lock tree for a thread represents the nested pattern in which locks are acquired by the thread. Each node of the runtime lock tree is labeled with a lock and represents the thread acquiring that lock. There is an edge from a node n_1 to a node n_2 if n_1 represents the most recently acquired lock that the thread holds when it acquires the lock associated with n_2. At each instant, each runtime lock tree has one node designated as the *current node*; the path from the root of the tree to that node represents the nested acquires of locks held by that thread at that instant. If a thread re-acquires a lock that it already holds, its runtime lock tree does not contain a node representing the re-acquire. When a thread acquires a lock that it does not already hold, if there is already a child of the current node labeled with that lock, that child becomes the current node, otherwise a new child labeled with that lock is created and becomes the current node. At the end of the execution of the program, if there exist threads t_1 and t_2 and locks l_1 and l_2 such that t_1 acquires l_2 while holding l_1, and t_2 acquires l_1 while holding l_2, then a warning of potential deadlock is issued, unless there is a common lock, called a gate lock, that is held by both threads when they acquire l_1 and l_2; the gate lock prevents the acquires of l_1 and l_2 from being interleaved in a way that leads to deadlock. The worst-case time complexity of the algorithm is $O(|T|^3 \times |Thread|^2)$, where $|T|$ is the size of the largest runtime lock tree, and $Thread$ is the set of threads. However, this algorithm only detects potential deadlocks caused by interleaving of lock acquires in two threads.

We present a generalized version of the GoodLock algorithm that detects potential deadlocks involving any number of threads. In particular, it checks whether there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$. Note that we always ignore a thread re-acquiring a lock it already holds, so a thread acquiring $l_{i+1 \bmod m}$ while holding l_i implies $l_{i+1 \bmod m}$ and l_i are different locks. In the absence of other constraints on the schedule (*e.g.*, due to gate locks or start-join synchronization), such acquires can be interleaved in a way that leads to deadlock. We call this the Potential for Deadlock from Locks Ignoring GateLocks (PDL-IGL) condition.

The algorithm constructs a runtime lock tree for each thread during execution, as described above. At the end of the execution, it constructs a runtime lock graph, which is a directed graph $G = (V, E)$, where V contains all the nodes of all the runtime lock trees, and the set E of directed edges contains (1) *tree edges:* the directed (from parent to child) edges in each of the runtime lock trees, and (2) *inter edges:* bidirectional edges between

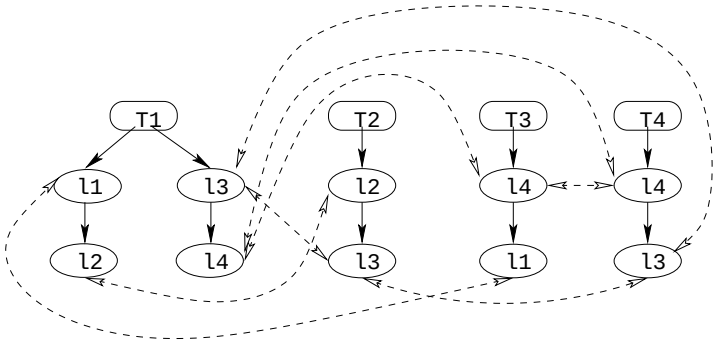

Figure 9.1: Runtime lock graph.

nodes that are labeled with the same lock and that are in different runtime lock trees.

For a runtime lock graph G, a *valid path* is a path that does not contain consecutive inter edges and such that nodes from each lock tree appear as at most one consecutive subsequence in the path. Similarly, a *valid cycle* is a cycle that does not contain consecutive inter edges and nodes from each thread appear as at most one consecutive subsequence in the cycle.

As an example, Figure 9.1 shows the runtime lock graph for the illustrative program in Figure 9.2. The graph in Figure 9.1 contains several cycles including the following three, where $\texttt{l}i^{\mathrm{T}j}$ denotes the node for lock $\texttt{l}i$ in the runtime lock tree for thread j: $\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$,

$\texttt{l1}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T1}}$, and

$\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$.

The first cycle is not valid because it contains two or more consecutive inter edges. The second cycle is not valid because nodes from thread T1 appear in more than one subsequence. The third cycle is valid and hence indicates a potential deadlock. Specifically, it indicates that the program in Figure 9.2 can deadlock if thread 1 acquires lock `l3` and waits for lock `l4` and thread 4 acquires lock `l4` and waits for lock `l3`.

Now we show that PDL-IGL holds iff the runtime lock graph G contains a valid cycle. Suppose there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$. Let n_i and n'_i denote the nodes in T_i corresponding to the acquire of l_i and the acquire of $l_{i+1 \bmod m}$ nested within it, respectively. Since thread t_i acquires lock l_i and waits for lock $l_{i+1 \bmod m}$, there is a path from n_i to n'_i in runtime lock tree T_i for t_i (because, n'_i is nested below n_i). Note that this path is made of tree edges. The locks l_i and $l_{i+1 \bmod m}$ are distinct, so this path contains at least one tree edge. Also, there is an inter edge from n'_i in runtime lock tree T_i to $n_{i+1 \bmod m}$ in runtime lock tree $T_{i+1 \bmod m}$ in G (by construction). These tree edges and inter edges together form a valid cycle.

Next, we show that existence of a valid cycle C in G implies that the PDL-IGL condition holds. The cycle involves nodes from more than one lock tree, because nodes of

```
Thread 1:     Thread 2:     Thread 3:     Thread 4:
  sync(l1) {    sync(l2) {    sync(l4) {    sync(l4) {
   sync(l2) {    sync(l3) {    sync(l1) {    sync(l3) {
   }             }             }             }
  }             }             }             }
  sync(l3) {
   sync(l4) {
   }
  }
```

Figure 9.2: Synchronization behavior of 4 threads. `sync` abbreviates `synchronized`.

a single tree cannot be involved in a cycle. Suppose, C had nodes n_i and n_i' in runtime lock tree T_i for thread t_i, $i \in 0..m-1$ (without loss of generality, we can just consider the beginning and end nodes in the consecutive subsequence from the same thread). Also, nodes n_i' and $n_{i+1 \bmod m}$ are labelled with the same lock (they are consecutive nodes from different lock trees and this is only possible through an inter edge which connects two similar labeled locks). Thus, existence of C implies there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ (node n_i corresponds to lock l_i and node n_i' corresponds to lock $l_{i+1 \bmod m}$) such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$. Hence, the PDL-IGL condition holds.

Our algorithm to detect existence of a valid cycle traverses all valid paths starting from the root of each lock tree in G using a modified depth-first search (DFS) algorithm, called DFS-ValidCycle, which differs from standard DFS in two ways. First, it traverses only valid paths, because it extends the current path (on the search stack) only with edges satisfying both criteria for validity. Second, a node all of whose neighbors have been explored may be explored multiple times (along incoming inter edges); this is necessary because the set of threads with some lock-tree nodes on the stack might be different on different visits, so the set of valid paths that can be explored by continuing the search from that node is different. The algorithm terminates when a valid cycle is found or all valid paths have been explored. Pseudo-code for the algorithm appears in Figure 9.3.

To see that the algorithm traverses every valid path, consider a valid path P that begins at a node n in a lock tree T. Extending P by prepending the edges on a path from the root of T to n produces a valid path that is explored by the algorithm when DFS-ValidCycle is started from the root of T. Note that a cycle involving P will be detected, because we check in the algorithm whether n' is anywhere on the stack (not just on the bottom).

To show the worst-case complexity of the algorithm, we consider the number of valid paths in the runtime lock graph. Let $S(k)$ be the number of valid paths in k lock trees $T_1, \ldots, T_k$, assuming the path visits those lock trees in that order. Then $S(k) = S(k-1) + N_k \times N_{k-1}$, where N_k and N_{k-1} are the number of nodes in lock trees T_k and T_{k-1}

```
/* check whether the graph contains a valid cycle. initially, the stack is empty, and all nodes
   are unvisited. edge(n,n1) denotes the edge from n to n1. */

DetectValidCycle() {
   stack = empty;
   for each lock tree T
      r = the root of T;
      DFS-ValidCycle(r, treeEdge);
}

DFS-ValidCycle(Node n, EdgeType lastEdgeType) {
   stack.push(n);
   /* Determine which neighbors of n to visit. */
   /* A neighbor of n is a node reachable from n by traversing a single edge. */
   if (lastEdgeType = interEdge)
      nbors = neighbors of n in the same lock tree;
   else {
      if (some neighbor of n is on the stack){
         print "valid cycle";
         halt;
      }
      nbors = all neighbors of n that
            (1) are in the lock tree of n or
            (2) are in a lock tree no node of which is on the stack.
   }
   for n_i in nbors /* iterate over the selected neighbors of n. */
      DFS-ValidCycle(n_i, edgeType(n,n_i));
   stack.pop(); /* remove n from the stack */
}
```

Figure 9.3: Algorithm to detect valid cycles.

respectively, because for each node n in T_{k-1}, the valid paths ending at n can be extended in N_k different ways. Thus, the total number of valid paths is $O(|V|^{|Thread|})$, where $|V|$ is the total number of nodes in the graph, and $|Thread|$ is the total number of threads. There are $|Thread|!$ permutations of $T_1, \ldots, T_k$, and each step of extension or backtracking takes constant time, so the overall worst-case complexity of this algorithm is $O(|V|^{|Thread|} \times |Thread|!)$.

The algorithm can be optimized by observing that many valid paths share a common suffix. Define an ordering on edge types: tree-edge $\geq$ inter-edge. This reflects the fact that in the definition of validity, a tree edge implies fewer restrictions on the next edge in the path. For each node n, n.`visits` is a set of pairs <ts, et>, where ts is a set of threads, and et is an edge type. The meaning of <ts, et> $\in n$.`visits` is that n has been visited along an edge with type et with a stack containing nodes from the lock trees of the threads in

```
/* check whether the graph contains a valid cycle. */
DetectValidCycle() {
   stack = empty;
   for each node s in each lock tree
      for each node r in each lock tree {
         r.visits = emptySet;
      DFS-ValidCycle(s,interEdge,s);
   }
}

DFS-ValidCycle(Node n, EdgeType lastEdgeType, StartNode s) {
   ts = {t in Threads — some node from the lock tree of t is on the stack};
   if ( ∃et ⩾ lastEdgeType.∃ts1 : n.visits.contains(⟨ts1, et⟩) and ts.containsAll(ts1))
      return;
   stack.push(n);
   n.visits.insert(⟨ts, lastEdgeType⟩);
   /* determine which neighbors of n to visit. */
   if (lastEdgeType = interEdge)
      nbors = neighbors of n in the same lock tree and not on the stack;
   else {
      if (s is a neighbor of n){
         print "valid cycle";
         halt;
      }
      nbors = neighbors of n that
            (1) are in the lock tree of n and are not on the stack, or
            (2) are in the lock tree of a thread not in ts.
   }
   for n_i in nbors /* iterate over the selected neighbors */
      DFS-ValidCycle(n_i,edgeType(n,n_i),s);
   stack.pop(); /* remove n from the stack */
}
```

Figure 9.4: Optimized algorithm to detect valid cycles.

ts. If we start the modified DFS at every node n, we do not need to explore a node n' if n'.`visits` contains a pair <$ts1, et$> such that the current stack contains all nodes from the lock trees of the threads in $ts1$ and n' is being visited along an edge with type less than or equal to et. If those conditions hold, then no valid cycles are reachable by continuing the search from n'. This is because there is no valid path from n' back to n that avoids the lock trees on the stack, because if there were, the search would have detected the cycle (containing n and n') and terminated during the visit that added that tuple to n'.`visits`. Pseudo-code for the optimized algorithm appears in Figure 9.4.

The worst-case complexity of the optimized algorithm is $O(2^{|Thread|} \times |V|^3)$, It is easy

to see that each node can have $O(2^{|Thread|})$ items in its visits set. Hence, each node can be explored $O(2^{|Thread|})$ times and during each visit it may need to visit its out-edges. There are at most $|V|$ out-edges from each node. Since we repeat the algorithm for each node, the overall worst-case complexity of the algorithm is $O(2^{|Thread|} \times |V|^3)$.

If the number of threads is a constant, then the algorithm is polynomial in the number of nodes in the runtime lock graph.

However, the algorithm does not consider gate locks and therefore produces false alarms whenever some common lock acquired by at least two threads prevents deadlocks. To eliminate these false alarms, we extend the algorithm to check whether there exist distinct $t_0 \ldots t_{m-1}$ and locks $l_0 \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$ and there do not exist t_i, t_j, and l such that t_i and t_j hold l when acquiring l_i and l_j, respectively. (Such a lock l is called a *gate lock* for the cycle). We call this the Potential for Deadlocks from Locks (PDL) condition.

To check the PDL condition, we modify the algorithm to backtrack (instead of halting) when a valid cycle is encountered, so the algorithm explores all valid cycles, and we check for every valid cycle generated whether there is a gate lock, *i.e.*, whether no two nodes in different runtime lock trees have ancestors labeled with the same lock. This can be done in $O(|V|^2 \times |Lock|)$ time for each valid cycle , where $|Lock|$ is the number of locks. If a valid cycle without a gate lock is found, potential for deadlock is reported.

9.4.2 Deadlock Types

Deadlock types associate a lock level with each lock. The typing rules ensure that if a thread acquires a lock l_2 (which the thread does not already hold) while holding a lock l_1, then l_2's level is less than l_1's level; in other words, locks are acquired in descending order. Lock levels and the partial order on them are defined by statements of the form `LockLevel` $\mathtt{l}_1$ `= new;` $\mathtt{l}_2$ `<` $\mathtt{l}_1$. In PRFJ, only locks on objects with owner `self` can be acquired (acquiring locks on other objects is not useful for showing race-freedom), so lock levels are associated only with objects with owner `self`. In this chapter, we consider only basic deadlock types, in which all instances of a class are associated with the same lock level.

In the deadlock type system, each method m is annotated with a locks clause that contains a set of lock levels. These lock levels are the maxima amongst the levels of locks that may be acquired when m is executed. To ensure that a program is free of deadlocks, the typing rule for method calls ensures that the caller only holds locks that are of a higher level than the levels in the called method's `locks` clause. A `locks` clause may also contain a lock l, which indicates that the thread invoking the method may hold a lock on object l. The typing rule for synchronized expression checks that the lock being acquired is l or has a lower level than l. This allows typing of programs in which, for example, a synchronized method of a class calls a synchronized method of the same class on the same object.

We designed a type inference algorithm for the above deadlock type system, which can infer deadlock types for programs [3].

9.4.3 Integrating Static Analysis and Runtime Checking

Deadlock types enforce a conservative strategy for preventing deadlocks. Runtime checking can safely be omitted for parts of the program guaranteed to be deadlock-free by the type system.

To optimize the generalized version of the GoodLock algorithm that does not handle gate locks, we find all the cycles of the form $l_1 > l_2 ... > l_1$ among lock level orderings produced by the deadlock type inference algorithm. We instrument only lock acquires and releases of expressions whose lock level is part of a cycle. Other synchronized expressions do not need to be instrumented. This leads to fewer intercepted events and smaller lock trees that need to be analyzed. It is easy to determine which lock levels are part of cycles. Construct a graph $G = (V, E)$, where each lock level is a node in V and there is an edge from l to l' if the inferred typing declares $l > l'$. A simple depth first search can find all nodes that are part of some cycle.

To optimize the generalized version of the GoodLock algorithm that handles gate locks, we find all the cycles among lock level orderings produced by the type inference algorithm as discussed above. All lock levels that are comparable to lock levels involved in a cycle in the ordering of lock levels need to be instrumented (not just the lock levels involved in a cycle).

9.4.4 Experiments

We implemented the unoptimized and optimized generalized Goodlock algorithms without gate locks described in Section 9.4.1 and used them to analyze the elevator program. Table 9.3 shows the running times for the `elevator` program with 3,7,15, 30 and 60 `Lift` threads. The "Base time" row gives the execution time of the original program without any instrumentation. The "Full" and "Focused" rows give the execution results of the program augmented with full and focused runtime checking, respectively. For "Full" and "Focused" rows, sub rows "Size", "Unopt" and "Opt" give the the number of nodes in all runtime lock trees, execution times of the unoptimized algorithm, and optimized algorithm respectively. As discussed in Section 9.4.3, focused runtime checking in this example intercepts lock acquires and releases only on instances of `Vector`. The results demonstrate that the focused analysis significantly decreases the runtime overhead of deadlock checking and the size of runtime lock trees. Let $\mathrm{Ofull} = \mathrm{Full} - \mathrm{Base}$ denote the overhead of full checking, and $\mathrm{Ofoc} = \mathrm{Focused} - \mathrm{Base}$ denote the overhead of focused checking. The average speedup (*i.e.*, fractional reduction in overhead) is $(\mathrm{Ofull} - \mathrm{Ofoc})/\mathrm{Ofull}$, which is 55.8% for the unoptimized algorithm, and 58.4% for the optimized algorithm. The average size of the runtime lock trees is reduced by 41%. Surprisingly, the optimized algorithm runs slower than the unoptimized algorithm, although its asymptotic worst-case time complexity is better. The main reason is that the optimized algorithm uses more complicated data structures, and for the modified `elevator` example, where the runtime lock graph is relatively simple, the benefit of caching explored paths falls short of the overhead of data structure maintenance.

		3 threads	7 threads	15 threads	30 threads	60 threads
Base time		0.23s	0.30s	0.52s	2.60s	6.60s
Full	Size	621	1037	1848	3359	5734
	Unopt	0.76s	0.94s	14.93s	1m23.08s	3m42.9s
	Opt	1.10s	1.66s	17.32s	1m28.05s	4m3.0s
Focused	Size	433	646	1063	1824	2947
	Unopt	0.40s	0.53s	11.71s	34.91s	1m22.66s
	Opt	0.51s	0.72s	12.40s	36.35s	1m28.28s

Table 9.3: Running times of dynamic deadlock checking for the modified elevator example.

Bibliography

[1] R. Agarwal, A. Sasturkar, and S. D. Stoller. Type discovery for parameterized race-free Java. Technical Report DAR-04-16, Computer Science Department, SUNY at Stony Brook, Sept. 2004.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM Press, Nov. 2005.

[3] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*. Springer-Verlag, Nov. 2005.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Proc. First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, Apr. 2003.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison Wesley, 1987.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, Nov. 2002.

[7] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, Nov. 2001.

[8] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. In *SRC Technical Note 2002-004*. Compaq Systems Research Center, 2002.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[11] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[12] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)*, 2003.

[13] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proc. 11th Int'l. SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*, pages 252–266. Springer-Verlag, 2004.

[14] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[15] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–267. ACM Press, 2004.

[16] C. Flanagan and S. N. Freund. Type inference against races. In *Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*. Springer-Verlag, Aug. 2004.

[17] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press, Jan. 2005.

[18] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 221–231. ACM Press, 2004.

[19] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[20] C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12. ACM Press, 2003.

[21] H. Gao and W. H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 44–56, 2004.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[23] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*. Springer-Verlag, Jan. 2004.

[24] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*, pages 245–264. Springer-Verlag, Aug. 2000.

[25] M. P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, Nov. 1993.

[26] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[27] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[28] Java Grande Forum. Java Grande Multi-threaded Benchmark Suite. version 1.0. Available from http://www.javagrande.org/.

[29] Jigsaw, version 2.2.4. Available from http://www.w3c.org.

[30] Decision Management Systems GmbH, Kopi compiler. Available from http://www.dms.at/kopi/.

[31] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[32] F. Mattern. Virtual time and global states of distributed systems. In M. Corsnard, editor, *Proc. International Workshop on Parallel and Distributed Algorithms*, pages 120–131. North-Holland, 1989.

[33] M. M. Michael. Private communication, 2004.

[34] M. M. Michael. Scalable lock-free dynamic memeory allocation. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 2004.

[35] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceeding of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275. ACM Press, 1996.

[36] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[37] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 167–178. ACM Press, 2003.

[38] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[39] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[40] W. Pugh. Fixing the Java memory model. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98, New York, NY, USA, 1999. ACM Press.

[41] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proc. 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–255. ACM Press, 2004.

[42] E. Ruf. Effective synchronization removal for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 208–218. ACM Press, June 2000.

[43] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[45] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297. ACM Press, 1991.

[46] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts (Seventh Edition)*. John Wiley & Sons, 2004.

[47] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 334–345. ACM Press, 2006.

[48] C. von Praun. Detecting synchronization defects in multi-threaded object-oriented programs. In *Ph.D. Dissertation, ETH Zurich*, 2004.

[49] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.

[50] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology, vol.3, no. 6*, June 2004.

[51] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[52] L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

[53] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, 2006.

[54] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, Feb. 2006.

[55] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *Proc. 2005 European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer-Verlag, July 2005.

[56] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2005.

[57] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In *Proc. Workshop on Software Model Checking (SoftMC'03)*, volume 89(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

Appendix A

Polynomial Equivalence of Conflict- and View-Atomicity

The following theorem shows that the problems of checking conflict-atomicity and checking view-atomicity are polynomially reducible to each other. This result is somewhat surprising, considering that checking conflict serializability is in P [5] and checking view serializability is NP-complete [38], so they are not polynomially reducible unless P=NP. To simplify the problem, we consider only transactions, *i.e.*, we assume there are no non-transactional units; we expect that the result also holds without this restriction. In this chapter, we first prove two lemmas and then the main theorem.

Lemma A.0.1. *Suppose $\langle T, \emptyset \rangle$ has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset \rangle$ is not conflict-atomic iff there are at least two inter-edges in the conflict-forest.*

Proof. Suppose that the two transactions are t and t'.

"$\Rightarrow$": We prove the contrapositive holds. If there is only one or no inter-edge between t and t', then it is easy to show that each of them has at most one commit node, so T is conflict-atomic according to Theorem 6.3.2.

"$\Leftarrow$": Suppose that one inter-edge connects node n_1 of t and node n_1' of t'; another inter-edge connects node n_2 of t and node n_2' of t'. These are two different edges, so $n_1 \neq n_2$ or $n_1' \neq n_2'$. If n_1 or n_2 is the ancestor of the other, n_1' and n_2' cannot be ancestor for each other according to the algorithm in Figure 6.2 (because the "outmost common lock" condition would imply $n_1 = n_2$ and $n_1' = n_2'$), so there are at least two commit nodes in t', hence T is not conflict-atomic according to Theorem 6.3.2. Otherwise, t contains at least two commit nodes, so T is not conflict-atomic according to Theorem 6.3.2. □

Lemma A.0.2. *Suppose $\langle T, \emptyset \rangle$ has no potential for deadlock, and T contains only two transactions. $\langle T, \emptyset \rangle$ is not view-atomic iff there are at least two inter-edges in the view-forest.*

Proof. The proof is similar as the proof for Theorem A.0.1. □

Theorem A.0.3. *The problems of checking conflict-atomicity and checking view-atomicity are polynomially reducible to each other when restricted to problem instances where the set E of non-transactional units is empty.*

Proof. 1. We first prove that the problem of checking conflict-atomicity can be polynomially reducible to the problem of checking view-atomicity. We transform T as follows. Let l be a lock not used in T. For each variable x, each read e_x^r is replaced by $e_l^{acq}\ e_x^r\ e_l^{rel}$; and each write e_x^w is replaced by $e_l^{acq}\ e_x^r\ e_x^w\ e_l^{rel}$. Let T' denote the resulting set of transactions.

Now we prove that T is conflict-atomic iff T' is view-atomic.

"$\Rightarrow$": For any trace tr of T, T is conflict-atomic implies that tr has an conflict-equivalent serial trace tr_s, We transform tr and tr_s in the same manner that is used to transform T, yielding tr' and tr'_s of T', respectively. For each read in the original trace tr, the corresponding read in tr' has the same write-predecessor in tr' and tr (because the insertions do not affect this relationship) and the same write-predecessor in tr and tr_s (because they are conflict equivalent), and the same write-predecessor in tr_s and tr'_s (because the insertions do not affect this), so it has the same write-predecessor in tr' and tr'_s. Similarly, we can show that the final write to each variable is the same in tr' and tr'_s. For each read inserted next to a write event, that write has the same order with respect to all other writes in tr and tr_s, so the associated read has the same write-predecessor in tr' and tr'_s. Thus tr' has a view-equivalent serial trace tr'_s. Because the lock l is not used in T, and is added in the pattern described above, there is a one-to-one correspondence between traces of T and traces of T'. According to the previous analysis, each trace of T' has an view-equivalent serial trace, so T' is view-atomic.

"$\Leftarrow$": For each trace tr' of T', we remove the operations inserted when constructing T' from T. This yields a trace tr of T. By assumption tr' has a view-equivalent trace tr'_s. Writes to the same variable must occur in the same order in tr' and tr'_s. Otherwise, the inserted read next to some write would have a different write predecessor in tr' and tr'_s. Using this observation, and similar reasoning as above, we can show that view-atomicity of T' implies conflict-atomicity of T.

2. Now we prove that the problem of checking view-atomicity is polynomially reducible to the problem of checking conflict-atomicity. We can check view-atomicity of pairs of transactions in T in polynomial time based on Theorem 6.3.5. In the following, we prove that checking view-atomicity can be reduced to checking conflict-atomicity when all pairs of transactions in T are view-atomic.

Because all pairs of transactions of T are view-atomic, all unit-non-final writes cannot be read by other transaction, and each read either reads a preceding write of its own transaction in all traces, or reads writes of other transactions in all traces. We remove from each transaction all unit-non-final writes and all unit-non-initial reads, *i.e.*, all unit-final writes and unit-initial reads are retained. Let T_f denote the resulting set of transactions.

Now we prove that T is view-atomic iff T_f is conflict-atomic.

"$\Rightarrow$": We prove the contrapositive by showing: if T_f has a trace tr that is not conflict-serializable, then tr is not view-serializable. Restoring the writes and reads removed when constructing T_f does not affect view serializability of the trace tr, because those writes are

not read by other transactions, and those reads do not read other transactions. Therefore, the resulting trace is a non-view-serializable trace for T, so T is not view-atomic.

Let g be the serialization graph for tr. Since tr is not conflict-serializable, g contains a cycle c of length two or more, *i.e.*, c contains two or more transactions. In the following, we prove by contradiction that c cannot contain exactly two transactions. Suppose that c contains exactly two transactions t_j and t_j, thus, c contains exactly two edges. We first show that the two edges indicate two edges in the conflict-forest for t_i and t_j which are also in the view-forest for t_i and t_j. For the edge $t_i \rightarrow t_j$, there must be two conflicting events e_i and e_j from t_i and t_j, respectively, and e_i happens before e_j in tr; similarly, for the edge $t_j \rightarrow t_i$, there must be two conflicting events e'_j and e'_i from t_j and t_i, respectively, and e'_j happens before e'_i in tr. There should be at least two inter-edges in the conflict-forest for t_i and t_j. Otherwise, *i.e.*, if there is only one inter-edge between t_i and t_j (it is easy to know there must be inter-edge(s) because of these conflict events), both e_i and e'_i happen either before or after both e_j and e'_j according to the algorithm in Figure 6.2. This contradicts with the conclusion discussed above. Therefore, there must be at least two inter-edges in the conflict-forest for t_i and t_j. Because all non-final writes and the following reads are removed, each inter-edge in the conflict-forest must be one of the following two kinds of inter-edges:(*i*) between a unit-initial read of one transaction and a unit-final write to the same variable of the other transaction; or (*ii*) between a unit-final write of one transaction and a unit-final write to the same variable of the other transaction. These two kinds of inter-edges also exist in the view-forest, *i.e.*, all edges in the conflict-forest also exist in the view-forest that consists of the same transactions. Thus, if c contains only two transactions, the view-forest would contain two inter-edges between nodes in those two transactions, so according to Lemma A.0.2, the two transactions would not be view-atomic; this contradicts with the assumption that all pairs of transactions are view-atomic. Hence c contains at least three transactions. Let T' denote the set of transactions contained on c. Hereafter we focus on the transactions in T'. Let tr' denote the subsequence of tr obtained by removing all events of transactions not in T'.

The algorithm in Figure A.1 shows an algorithm to generate a shortened cycle c_v from c. We will show that existence of this cycle implies T is not view-atomic. The first three cases identify and mark the edges that denote precedence between transactions for all serial traces view-equivalent to tr, *i.e.*, if an edge $t_1 \rightarrow t_2$ is marked, then t_1 precedes t_2 in all serial traces view-equivalent to tr. The next two cases add shortcut edges to shorten c; the new shortcut edges are immediately marked because they also denote precedence between transactions for all serial traces view-equivalent tr. In the algorithm, c is updated in each iteration, and tr' is updated accordingly by removing the transactions not on the current c. Let c_v and tr_v denote the cycle c and trace tr' when the algorithm terminates. Note that tr_v is a sub-sequence of the original tr'. tr_v does not have any view-equivalent serial trace, because of the cyclic precedence of transactions indicated by c_v. More details are discussed next.

Each edge on c must imply one of the six cases shown in the cases view write-read, view trace_initial_read-write, view write-trace_final_write, write-read, write1-write2, and

read-write. The added edge eg' in each case also belongs to one of the six cases.

An edge from t_i to t_j marked in the case by "view write-read" implies that t_i contains a write e_x^w whose written value is read by t_j. Thus, for any sub-trace of tr' obtained by deleting transactions other than t_i and t_j, if it contains the write and read to x in t_i and t_j, respectively, t_i precedes t_j in all serial traces view-equivalent to that sub-trace. Edges marked in the cases view trace_initial_read-write and view write-unit_final_write have the similar implication.

In the case write-read, the write-predecessor of the read must be in a third transaction, *i.e.*, t_k is neither t_i nor t_j. To see this, first note that the write-predecessor of e_x^r cannot be e_x^r, otherwise this edge would be handled in the case "view write-read". Therefore, if t_k were t_i, t_i would contain two writes to x contradicting the fact that each transaction in T_f contains at most one write to each variable. If t_k were t_j, e_x^r would not be a unit-initial read in t_j, contradicting the fact that unit-non-initial reads are removed when constructing T_f. In the following, we show by contradiction that c contains at least three transactions after being shortened. Suppose c contains only two transactions, namely t_k and t_j, after being shortened. c contains an edge $t_i \rightarrow t_k$, which was on c already, and the new-edge $t_k \rightarrow t_j$. From the definition of view-forest, the view-forest contains inter-edges corresponding to these two edges, so according to Lemma A.0.2, t_j and t_k are not view-atomic; this contradicts the assumption. The algorithm marks the edge $t_k \rightarrow t_j$ because t_k must precede t_j in all serial traces view-equivalent to sub-traces of tr' that contain t_k and t_j.

In the case write1-write2, the trace-final write must be in a third transaction. If t_k were t_i, t_i would contain two writes to the same variable, which implies a contradiction. If t_k were t_j this edge would be processed in the case view write-unit_final_write instead. After being shortened in this case, c must contain at least three transactions. Otherwise, suppose c contains only two transactions, namely t_i and t_k, by the similar reasoning as above, the shortened cycle c contains at least three transactions after this step. The added edge eg' always implies that t_i must happen before t_k in all serial traces view-equivalent to sub-traces of tr' that contain t_i and t_k.

In the case "read-write", t_k cannot be t_i because the e_x^r is a unit-initial read. t_k cannot be t_j because each transaction cannot have two writes to the same variable. t_h cannot be t_i because e_x^r is a unit-initial read. Note that t_h may be t_j. Similar as before, c must contains at least three transactions after this step of shortening cycle by the similar reasoning as above.

After each iteration of the algorithm in Figure A.1, c is always a cycle with at least three nodes (*i.e.*, transactions), hence, c_v is a cycle and contains at least three transactions. Each edge $t_i \rightarrow t_j$ of c_v, implies that transaction t_i must happen before t_j in all serial traces view-equivalent to tr_v. Thus, existence of cycle c_v implies that there is no serial trace view-equivalent to tr_v. If we consider only the transactions contained by tr_v, after restoring the deleted writes and reads, the resulting trace is still not view-serializable because these deleted writes and reads do not interact with any transaction except for its own. Because the set of transactions in this trace is a subset of T, T is not view-atomic.

"$\Leftarrow$": Suppose T_f is conflict-atomic. This implies T_f is also view-atomic. For each

while(some edges of c are unmarked){
 let $t_i \rightarrow t_j$ be an unmarked edge of c;

 /* view write-read: this edge is kept in c_v. */
 if $\exists x$.(the written value by a write e_x^w of t_i is read by a read e_x^r of t_j in tr')
  mark $t_i \rightarrow t_j$; continue;

 /* view trace_initial_read-write: this edge is also kept in c_v. */
  if $\exists x$.(t_i contains a trace-initial read of x in tr' and t_j contains a write to x)
   mark $t_i \rightarrow t_j$; continue;

 /* view write-trace_final_write: this edge is also kept in c_v. */
 if $\exists x$.(t_i contains a write to x and t_j contains a trace-final write to x in tr')
    mark $t_i \rightarrow t_j$; continue;

 /* write-read: find the write-predecessor of the read, delete the current path on c from the
 write-predecessor to the read, and add an edge from the write-predecessor to the read.*/
 if $\exists x$.(a write e_x^w of t_i happens before a read e_x^r of t_j in tr')
  let t_k be the transaction that contains the write-predecessor of e_x^r in tr';
  c := c - {path from t_k to t_j on c};
  remove from tr' transactions no longer on c;
  c = $c \cup \{t_k \rightarrow t_j\}$;
  mark edge $t_k \rightarrow t_j$;
  continue;

 /* write1-write2: find the trace-final write to x in tr', delete the current path on c
 from write1 to write2, and add an edge from write1 to the trace-final write to x.*/
 if $\exists x$.(a write e_x^w of t_i happens before a write e_x^w of t_j in tr')
  let t_k be the transaction that contains the trace-final write to x in tr';
  c := c - {path from t_i to t_k on c};
  remove from tr' transactions no longer on c;
  c := $c \cup t_i \rightarrow t_k$;
  mark $t_i \rightarrow t_k$;
  continue;

 /* read-write: find the write-predecessor of the read and the trace-final write
    to the same variable, connect the write-predecessor and the trace-final write,
    at last delete the path that contains the read.*/
 if $\exists x$.(a read e_x^r of t_i happens before a write e_x^w of t_j in tr')
  let t_k denote the transaction that contains the write-predecessor of e_x^r in tr';
  let t_h denote the transaction that contains the trace-final write to x in tr';
  c := c - {path from t_k to t_h on c};
  remove from tr' transactions no longer on c;
  c = $c \cup t_k \rightarrow t_h$;
  mark $t_k \rightarrow t_h$;
  continue;
}

Figure A.1: The algorithm to reduce a conflict-atomicity violating cycle to a view-atomicity violating cycle.

trace tr of T, there is a corresponding trace tr_f of T_f. Since T_f is view-atomic, there is a serial trace tr_f^s that is view-equivalent to tr_f. We can expand tr_f^s into a serial trace tr^s for T by restoring the removed unit-non-initial reads and unit-non-final writes for each transaction. tr^s is view-equivalent to tr because all reads have the same write-predecessor in tr and tr^s, and all trace-final writes are the same in tr and tr^s. Hence, T is view-atomic. □

Appendix B

Atomicity Analysis of Non-Blocking Algorithm

B.1 Simulation of NFQ′

By construction, NFQ′ is more non-deterministic than NFQ and can simulate all behaviors of NFQ. We show below that linearizability of NFQ with respect to any specification (of the kind defined in [26]) follows from linearizability of NFQ′ with respect to that specification augmented freely with calls to `UpdateTail`.

Following [26], a specification $Spec$ is a prefix-closed set of single-object sequential histories.

Based on a given $Spec$ for NFQ, the specification $Spec'$ for NFQ′ is defined by introducing a new thread P_{Tail} that executes the `UpdateTail` procedure: for each $H \in Spec$, for each well-formed sequential history H' that can be obtained by inserting $\langle Q.UpdateTail(), P_{Tail}\rangle$ (which denotes an invocation of `Q.UpdateTail` by P_{Tail}) and $\langle Q.OK(), P_{Tail}\rangle$ (which denotes a return, also called response, from `Q.UpdateTail` by P_{Tail}) in H, add H' to $Spec'$.

Recall that $complete(H_r)$ is the subsequence of H_r obtained by deleting the invocations without matching responses. Recall that H is linearizable with respect to $Spec$ if H can be extended to some history H_r by adding response events, such that

$L1$. $complete(H_r)$ is equivalent to some $S \in Spec$, *i.e.*, $\forall$ thread P : $complete(H_r)|P = S|P$, and

$L2$. $<_H \subseteq <_S$

Theorem B.1.1. *If NFQ′ is linearizable with respect to $Spec'$ then NFQ is linearizable with respect to $Spec$.*

Proof. Let σ be an execution of NFQ. Let H be the corresponding history of NFQ obtained by deleting all actions except for call/return. Construct an execution σ' from σ as follows.

- Replace each successful SC(Tail,_) with an execution of `UpdateTail()` by P_{Tail}. Success of the original SC implies that the SC in `UpdateTail()` succeeds.

- Delete each unsuccessful execution of SC(Tail,_).

One can show that σ' is an execution of NFQ′. Let H' denote the corresponding history. Linearizability of NFQ′ with respect to $Spec'$ implies that there is an execution H'_r of H' and a sequential history $S' \in Spec'$ such that

$L1'$. $complete(H'_r)$ is equivalent to S', and

$L2'$. $<_{H'} \subseteq <_{S'}$

Let H_r and S be the subsequences of H'_r and S', respectively, obtained by deleting all invocations of `UpdateTail()` and the matching responses. Note that H_r is an extension of H by adding response events. Also, $S \in Spec$, by design of $Spec'$. Note that $complete(H_r)$ is equivalent to S; this follows from $L1'$, and the fact that P_{Tail} does not appear in H_r or S, so the projection of both onto P_{Tail} is the empty sequence.

Let $f(<)$ denote the projection of an ordering $<$ onto operations of all threads other than P_{Tail}. $L2'$ implies $f(<_{H'}) \subseteq f(<_{S'})$. Note that $<_H = f(<_{H'})$ and $<_S = f(<_{S'})$. So $<_H \subseteq <_S$. □

The converse of the above theorem can be proved similarly. Therefore, this approach, *i.e.*, proving the linearizability of NFQ by showing the linearizability of NFQ′, is complete.

B.2 Semantics of SYNL

B.2.1 Domains

The semantic domains used in the semantics of SYNL are shown in Table B.1. $A \rightharpoonup B$ is the type of partial functions from A to B. A program state is a tuple $\langle G, H, T \rangle$ containing a global store G, a heap H, and a sequence T containing, for each thread, a local store L and a statement to be executed next. The address of a record structure (an object or array) is often stored in a reference variable. To access the record structure, there are two maps: the first map is from reference variable to address, and has type $GStore$ or $LStore$; the second map is from address to structure, and has type $Heap$. $H[p \mapsto d]$ denotes a new heap that is identical to H except it maps address p to record d.

The *Struct* domain allows arrays with gaps in the set of legal indices. This generality is unnecessary but harmless; our proofs remain valid for semantic domains that exclude arrays with gaps.

The semantics of LL/VL/SC and CAS associate a set of thread identifiers with each global variable, each field of each object, and each element of each array. We call this information *synchronization state* and represent it using the *SyncState* domain. The set of thread identifiers denotes that what threads have read the corresponding variable before they successfully submit their updates. For example, suppose variable v has a set Y of thread identifiers. A LL(v) by thread i adds i into Y. A successful SC(v, val) by thread j resets Y to empty.

$\rightarrow_i$ is the transition relation of thread i. $\rightarrow$ is the transition relation of the program.

$$
\begin{array}{rcl}
\varphi & \in & State = GStore \times Heap \times Thread^* \\
H & \in & Heap = Addr \rightharpoonup Struct \\
d & \in & Struct = (Field \rightharpoonup Val \times SyncState) \cup (Index \rightharpoonup Val \times SyncState) \\
L & \in & LStore = LVar \rightharpoonup Val \\
t & \in & Thread = LStore \times Statement \\
v & \in & Val = Addr \cup int \cup bool \\
i & \in & TID = Nat \\
G & \in & GStore = GVar \rightharpoonup Val \times SyncState \\
Y & \in & SyncState = Set(TID) \\
idx & \in & Index = Nat \\
p & \in & Addr \\
T & \in & Thread^* \\
\rightarrow_i & \subseteq & State \times State \\
\rightarrow & \subseteq & State \times State
\end{array}
$$

Table B.1: Semantic domains for SYNL.

B.2.2 Evaluation Contexts

Based on the syntax and semantic domains of SYNL defined in Tables 8.1 and B.1, respectively, the evaluation contexts of SYNL are defined in Table B.2. Evaluation contexts are used to identify the next part of an expression or statement to be executed. An evaluation context E is an expression or statement with a hole in place of the next sub-expression or sub-statement to be evaluated. Expressions evaluate to expressions and eventually become values. Statements evaluate to statements and eventually become the `done` statement or get blocked or stuck.

Table B.2 also introduces additional expression and statement forms to help keep track of computations. The `inlet` statement denotes that execution is proceeding inside a statement `local`. The `inloop` and `insync` statements are similar.

Let $T[i]$ denote the i^{th} element of sequence T. In a state G, H, T where $T[i]$ contains `insync` p, we say that thread i holds lock p. In a state where no thread holds lock p, we say that lock p is free. We refer to this as the state of the lock.

Note that LL(E) is not an evaluation context; if it were, LL(x) would evaluate to, *e.g.*, LL(3), if the value of x is 3.

$$
\begin{array}{rcl}
Expr & ::= & p \\
Statement & ::= & \texttt{inloop}\ s\ s \mid \texttt{done} \mid \texttt{inlet}\ x\ s \mid \texttt{insync}\ p\ s \\
E & ::= & [\,] \mid E.fd \mid E[e] \mid x[E] \mid prim(e_1, ..., e_n, E, ...) \mid \text{SC}(loc, E) \\
 & & \mid \text{CAS}(loc,E,e) \mid \text{CAS}(loc,v,E) \mid loc := E \mid \texttt{if}\ E\ s\ s \mid \texttt{loop}\ E \\
 & & \mid \texttt{inloop}\ s\ E \mid E;s \mid \texttt{local}\ x := E\ \texttt{in}\ s \mid \texttt{inlet}\ x\ E \\
 & & \mid \texttt{return}\ E \mid \texttt{synchronized}\ E\ s \mid \texttt{insync}\ p\ E
\end{array}
$$

Table B.2: Evaluation contexts of SYNL.

B.2.3 Transition Rules

Let π_i select the i^{th} component of a tuple. For example, $\pi_2(\langle a, b, c\rangle) = b$. For a mapping L, let $L - x$ denote L with x removed from its domain.

The transition rules of SYNL are shown in Figures B.1 and B.2, where $1 \leqslant i \leqslant |T|$, and

$$val(x, G, L) = \begin{cases} \pi_1(G(x)) & \text{if } x \in dom(G) \\ \pi_1(L(x)) & \text{if } x \in dom(L) \\ \bot & \text{otherwise} \end{cases}$$

The transition rule $G, H, T \rightarrow_i G, H, T.\langle L, s\rangle$ in Figure B.2 models the environment calling a procedure in a new thread.

Recall that SYNL allows procedures called implicitly and concurrently by the environment, and SYNL does not allow explicit procedure calls (internal procedures are inlined).

The rule $G, H, T.\langle L, \texttt{done}\rangle.T' \rightarrow_i G, H, T.\langle L', s\rangle.T'$ in Figure B.2 models the environment calling a procedure in an existing thread that finished its previous procedure calls.

Note that the only actions performed by the environment are calls to procedures defined in the program.

This semantics does not model garbage collection of unreachable structures or terminated threads. This semantics allows the heap, procedure arguments, *etc.*, to contain invalid addresses, *i.e.*, addresses not in $dom(H)$. Attempting to dereference them causes the thread to get stuck. We can consider $null$ to be such an invalid address, if getting stuck is appropriate semantics for an attempted dereferencing of $null$. Otherwise, we could introduce a special $null$ value in $Addr$, and add appropriate transition rules for dereferencing of $null$. The semantics for `return` does not explicitly model the communication of the return value to the environment; it could easily be modified to do so.

B.3 Proof of Theorem 8.4.1

Let $\sigma = G_0, H_0, T_0 \rightarrow_{t_0} G_1, H_1, T_1 \rightarrow_{t_1} \ldots$ be an execution of a program P.

A normal iteration leads to another iteration of the loop's body, so thread i_0 is at the same "control point" in T_{α_0} and $T_{\alpha_{n+1}}$. In our semantics, this means that $\pi_2(T_{\alpha_0}[i_0]) = \pi_2(T_{\alpha_{n+1}}[i_0])$.

Let I be the indices in σ of all transitions that are part of normal iterations of pure loops. Let σ' be the execution constructed from σ by deleting transitions in I. Deleting transitions involves adjusting the states as follows. The j^{th} state in σ' corresponds to the $f(j)^{th}$ state in σ, where $f(j) = max(\{m \mid m - |[0..m-1] \cap I| = j\})$, *i.e.*, $f(j)$ is the maximal m which satisfies that there are j transitions remaining from the 0^{th} to $(m-1)^{th}$ transitions after deleting transitions in I. [0..m-1] denotes the set of integers from 0 to $m-1$.

The $f(j)^{th}$ state in σ is denoted as $G_{f(j)}, H_{f(j)}, T_{f(j)}$. The j^{th} state in σ' is denoted as G'_j, H'_j, T'_j and is computed as follows. Let $p \in Addr$, $fd \in Field$, $x \in LVar \cup GVar$. The treatment of arrays is very similar to the treatment of records, so for brevity, we show only the latter.

$$
\begin{array}{rcl}
G,H,T.\langle L,E[x]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[v]\rangle.T', \\
 & & \text{if } v = val(x,G,L) \wedge v \neq \bot \\
G,H,T.\langle L,E[p.fd]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\pi_1(H(p)(fd))]\rangle.T', \\
 & & \text{if } p \in dom(H) \wedge fd \in dom(H(p)) \\
G,H,T.\langle L,E[p[idx]]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\pi_1(H(p)(idx))]\rangle.T', \\
 & & \text{if } p \in dom(H) \wedge idx \in dom(H(p)) \\
G,H,T.\langle L,E[new\ C]\rangle.T' & \rightarrow_i & G,H[p \mapsto d],T.\langle L,E[p]\rangle.T', \text{ where } p \notin dom(H), \\
 & & \text{Note: } d \text{ is a record of type C,} \\
 & & \text{and appropriately initialized} \\
G,H,T.\langle L,E[prim(\bar{v})]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[v_0]\rangle.T', \text{ where } v_0 = [\![prim]\!](\bar{v}) \\
 & & \text{Note: } prim \text{ operations have no side effect} \\
G,H,T.\langle L,E[LL(x)]\rangle.T' & \rightarrow_i & G[x \mapsto \langle v, Y \cup \{i\}\rangle],H,T.\langle L,E[v]\rangle.T', \\
 & & \text{if } x \in dom(G) \wedge \langle v,Y\rangle = G(x) \\
G,H,T.\langle L,E[LL(x.fd)]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[fd \mapsto \langle v, Y \cup \{i\}\rangle]],T.\langle L,E[v]\rangle.T' \\
 & & \text{if } p = val(x,G,L) \wedge p \in dom(H) \wedge fd \in dom(H(p)) \\
 & & \quad \wedge \langle v,Y\rangle = H(p)(fd) \\
 & & \text{Note: } p \in dom(H) \text{ implies } p \neq \bot \\
G,H,T.\langle L,E[LL(x[idx])]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[idx \mapsto \langle v, Y \cup \{i\}\rangle]],T.\langle L,E[v]\rangle.T' \\
 & & \text{if } p = val(x,G,L) \wedge p \in dom(H) \wedge idx \in dom(H(p)) \\
 & & \quad \wedge \langle v,Y\rangle = H(p)(idx) \\
 & & \text{Note: } p \in dom(H) \text{ implies } p \neq \bot \\
G,H,T.\langle L,E[VL(x)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{true}]\rangle.T', \text{ if } x \in dom(G) \wedge i \in \pi_2(G(x)) \\
G,H,T.\langle L,E[VL(x)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T', \text{ if } x \in dom(G) \wedge i \notin \pi_2(G(x)) \\
G,H,T.\langle L,E[VL(x.fd)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge fd \in dom(H(p)) \wedge i \in \pi_2(H(p)(fd)) \\
G,H,T.\langle L,E[VL(x.fd)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge fd \in dom(H(p)) \wedge i \notin \pi_2(H(p)(fd)) \\
G,H,T.\langle L,E[VL(x[idx])]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge idx \in dom(H(p)) \wedge i \in \pi_2(H(p)(idx)) \\
G,H,T.\langle L,E[VL(x[idx])]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge idx \in dom(H(p)) \wedge i \notin \pi_2(H(p)(idx)) \\
G,H,T.\langle L,E[SC(x,v)]\rangle.T' & \rightarrow_i & G[x \mapsto \langle v, \emptyset\rangle],H,T.\langle L,E[\texttt{true}]\rangle.T', \\
 & & \text{if } x \in dom(G) \wedge i \in \pi_2(G(x)) \\
G,H,T.\langle L,E[SC(x,v)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T', \text{ if } x \in dom(G) \wedge i \notin \pi_2(G(x)) \\
G,H,T.\langle L,E[SC(x.fd,v)]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[fd \mapsto \langle v, \emptyset\rangle]],T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge fd \in dom(H(p)) \wedge i \in \pi_2(H(p)(fd)) \\
G,H,T.\langle L,E[SC(x.fd,v)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge fd \in dom(H(p)) \\
 & & \quad \wedge i \notin \pi_2(H(p)(fd)) \\
G,H,T.\langle L,E[SC(x[idx],v)]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[idx \mapsto \langle v, \emptyset\rangle]],T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \neq \bot \wedge p \in dom(H) \\
 & & \quad \wedge idx \in dom(H(p)) \wedge i \in \pi_2(H(p)(idx)) \\
G,H,T.\langle L,E[SC(x[idx],v)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge idx \in dom(H(p)) \\
 & & \quad \wedge i \notin \pi_2(H(p)(idx))
\end{array}
$$

Figure B.1: Transition rules of SYNL, part 1.

$$\begin{array}{rcl} H'_j(p)(fd) & = & \texttt{let } k = max((WriteH(\sigma,p,fd)\backslash I) \cap [0..f(j)-1]) \\ & & \texttt{in } H_{k+1}(p)(fd) \\ G'_j(x) & = & \texttt{let } k = max((WriteG(\sigma,x)\backslash I) \cap [0..f(j)-1]) \\ & & \texttt{in } G_{k+1}(x) \end{array}$$

$WriteH(\sigma, p, fd)$ denotes the indices of transitions in σ that write the value or synchronization state of field fd of the structure at address p. Thus, $max((WriteH(\sigma, p, fd) \backslash I) \cap [0..f(j)-1])$ denotes the maximal transition that is not contained in I and writes the value or synchronization state of $p.fd$ before the $f(j)^{th}$ state. $WriteG(\sigma, x)$ denotes the indices of transitions in σ that write the value or synchronization state of global variable x.

Note that a LL is considered as a write, since it writes the synchronization state of x. Failed SC and CAS transitions are not considered as writes.

Let $Trans(\sigma, i)$ denote the indices of transitions of thread i in σ. $WriteL(\sigma, x, i)$ is analogous to $WriteG$, except it is for local variables of thread i.

$$\begin{array}{rcl} T'_j & = & \langle L'_j, S'_j \rangle \\ L'_j[i](x) & = & \texttt{let } k = max((WriteL(\sigma,x,i)\backslash I) \cap [0..f(j)-1] \\ & & \texttt{in } L_{k+1}(x) \\ S'_j[i] & = & \texttt{let } k = max((Trans(\sigma,i)\backslash I) \cap [0..f(j)-1]) \\ & & \texttt{in } S_{k+1}[i] \end{array}$$

The following formulas for σ express the fact that each variable contains the value most recently written to it.

$$\begin{array}{rcl} H_j(p)(fd) & = & \texttt{let } k = max((WriteH(\sigma,p,fd)) \cap [0..j-1]) \\ & & \texttt{in } H_{k+1}(p)(fd) \\ G_j(x) & = & \texttt{let } k = max((WriteG(\sigma,x)) \cap [0..j-1]) \\ & & \texttt{in } G_{k+1}(x) \end{array}$$

For each thread i, we have

$$\begin{array}{rcl} L_j[i](x) & = & \texttt{let } k = max((WriteL(\sigma,x,i)) \cap [0..j-1] \\ & & \texttt{in } L_{k+1}(x) \\ S_j[i] & = & \texttt{let } k = max((Trans(\sigma,i)) \cap [0..j-1]) \\ & & \texttt{in } S_{k+1}[i] \end{array}$$

A storage location can be a local variable, global variable, field, or array element. The sets of locations read or written by a transition are defined in a straightforward way.

Let τ'_j denote the j^{th} transition of σ', *i.e.*, $G'_j, H'_j, T'_j \rightarrow_i G'_{j+1}, H'_{j+1}, T'_{j+1}$. Let $\tau_{f(j)}$ denote the corresponding transition in σ, *i.e.*, $G_{f(j)}, H_{f(j)}, T_{f(j)} \rightarrow_i G_{f(j+1)}, H_{f(j+1)}, T_{f(j+1)}$.

Lemma B.3.1. *For every transition τ'_j in σ',*
(i) τ'_j and $\tau_{f(j)}$ are transitions of the same thread, call it thread i, and
(ii) $S'_j[i] = S_{f(j)}[i]$, and
(iii) all locations read by τ'_j have the same value in state G'_j, H'_j, $L'_j[i]$ and state $G_{f(j)}$, $H_{f(j)}$, $L_{f(j)}[i]$.

Proof. Claims (i) and (ii) follow directly from the definitions of σ' and f. For claim (iii), we consider the different kinds of locations that τ'_j may read.

1. τ'_j reads a global variable x, *i.e.*, $x \in GVar$.

According to the definition of pure loop, I does not contain writes to global variables, so $WriteG(\sigma, x) \cap I = \emptyset$. Thus, in the definitions of $G'_j(x)$ and $G_{f(j)}(x)$, k is $max(WriteG(\sigma, x) \cap [0..f(j) - 1])$, therefore, $G'_j(x) = G_{f(j)}(x)$.

2. τ'_j reads a local variable x of thread i.

Let τ_k be thread i's last write to x before $\tau_{f(j)}$ in σ; note that this is the same value of k as in the definition of $G_{f(j)}(x)$. To conclude $L'_j[i](x) = L_{f(j)}[i](x)$, it suffices to show that $k \notin I$, which implies that the write τ_k also appears in σ'. We prove this by contradiction. Suppose $k \in I$. Note that $f(j) \notin I$, because $\tau_{f(j)}$ corresponds to τ'_j and hence is not a deleted transition. $k \in I$ implies τ_k is a pure action in a normal iteration of some loop. (*1.ii.a*) in the definition of pure action implies that there is another write to x by thread i between τ_k and $\tau_{f(j)}$, contradicting the definition of k.

3. τ'_j reads a field of an unshared object.

The proof is the same as in case 2 for local variables, except for differences in notation.

4. τ'_j reads a field $p.fd$ of a shared object.

Let τ_w be thread i's last write to $p.fd$ before $\tau_{f(j)}$ in σ. Similar as Case 2, it suffices to show that $w \notin I$. We prove this by contradiction. Suppose $w \in I$.

If τ_w and $\tau_{f(j)}$ are executed by the same thread i, by the same reason as in Case 2, there is another write to x by thread i between τ_w and $\tau_{f(j)}$, contradicting the definition of w.

Suppose τ_w and $\tau_{f(j)}$ are executed by different threads i to h, respectively. According to the definition of pure loop, τ_w is performed by dereferencing primary reference(s). For brevity, we talk only one primary reference, the proof for multiple primary references can be done in the same way. Thus, $\tau_{f(j)}$ must be performed by dereferencing a quasi-unique reference. Since $\tau_{f(j)}$ is not performed by dereferencing a secondary reference (otherwise, $\tau_{f(j)}$ cannot have a corresponding τ_j in σ), $\tau_{f(j)}$ must be performed by dereferencing a primary reference. Therefore, the ownerships of the quasi-unique reference is transferred from thread i to thread h. In SYNL, this transfer must be performed by assigning the reference to some global variable. Let $asgn$ denote the assignment. $asgn$ is not in normal iterations, because τ_w is performed by dereferencing primary references. Thus, $asgn$ happens after the normal iteration where τ_w occurs, and before $\tau_{f(j)}$. By the same reason as in Case 2, there is another write to $p.fd$ by thread i between τ_w and $asgn$, *i.e.*, τ_w and $\tau_{f(j)}$, contradicting the definition of w.

5. τ'_j reads an array element. The analysis is similar to the analysis for a field access.

6. τ'_j performs LL or CAS.

Reads performed by LL or CAS do not require separate analysis: the preceding analysis applies to these reads.

7. τ'_j performs SC or VL on some location x.

A SC or VL reads the synchronization state of thread i for that variable (*i.e.*, checks whether i is in the set of thread id's associated with x). Since this SC or VL is not in the deleted normal iteration I of the pure loop, condition (2) in the definition of pure loop implies that the matching LL, if any, is not in I, so deleting I does not affect the synchronization state read by this SC or VL.

8. τ'_j performs a `synchronized` transition, *i.e.*, acquires some lock.

The lock must be free or held by thread i in the state immediately before $\tau_{f(j)}$. Synchronized statements are block-structured, so if some transition in I performs an acquire transition, then (i) I also contains the matching release transition (which exits the corresponding synchronization block), and (ii) $\tau_{f(j)}$ does not occur between that matching pair of lock operations. Deleting an execution of a synchronized block does not affect the state of the lock before or after it, so the state of the lock before τ'_j is the same as it is before $\tau_{f(j)}$. □

Lemma B.3.2. *σ' is an execution of the program P.*

Proof. A straightforward property of the operational semantics is that, if some transition rule shows that $\varphi_1 \rightarrow \varphi_2$ is a transition of P, and φ'_1 and φ'_2 are obtained from φ_1 and φ_2, respectively, by a change to some parts of the state that are not accessed by the transition according to Lemma B.3.1, then the same transition rule shows that $\varphi'_1 \rightarrow \varphi'_2$ is a transition of P. We conclude that τ'_j is a transition of the program based on the same transition rule used to show that $\tau_{f(j)}$ is a transition of the program. Therefore, σ' is an execution of the program P. □

Lemma B.3.3. *σ and σ' contain the same states in which all threads are executing outside pure loops.*

Proof. All deleted transitions are in pure loops, so there is a one-to-one correspondence between states outside executions of pure loops in σ and states outside executions of pure loops in σ'. We show that the corresponding states are the same. By inspection of the formulas defining H', G', and T', deletion of a transition that updates a location x produces a difference between corresponding states in σ and σ' that propagates forward in σ' until it encounters either a transition that updates x or the end of x's scope. Conditions (*1.ii.a*) and (*1.ii.b*) in the definition of pure loop imply that, for each such location x, at least one of these two things happens before the end of the pure loop. □

Theorem 8.4.1 *Let σ be an execution of a program P. Let σ' be an execution obtained from σ by deleting all transitions in all normal iterations of all pure loops in P. Then σ' is also an execution of P, and σ and σ' contain the same states in which all threads are executing outside pure loops.*
Proof. The theorem follows directly from Lemmas B.3.2 and B.3.3. □

$$
\begin{array}{rcl}
G,H,T.\langle L,E[CAS(x,v_1,v_2)]\rangle.T' & \rightarrow_i & G[x \mapsto \langle v_2,\pi_2(G(x))\rangle],H,T.\langle L,E[\texttt{true}]\rangle.T', \\
 & & \text{if } x \in dom(G) \wedge \pi_1(G(x)) = v_1 \\
G,H,T.\langle L,E[CAS(x,v_1,v_2)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T', \\
 & & \text{if } x \in dom(G) \wedge \pi_1(G(x)) \neq v \\
G,H,T.\langle L,E[CAS(x.fd,v_1,v_2)]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[fd \mapsto \langle v_2,\pi_2(H(p)(fd))\rangle]], \\
 & & T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge fd \in dom(H(p)) \\
 & & \wedge \pi_1(H(p)(fd)) = v_1 \\
G,H,T.\langle L,E[CAS(x.fd,v_1,v_2)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge fd \in dom(H(p)) \\
 & & \wedge \pi_1(H(p)(fd)) \neq v_1 \\
G,H,T.\langle L,E[CAS(x[idx],v_1,v_2)]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[idx \mapsto \langle v_2,\pi_2(H(p)(fd))\rangle]], \\
 & & T.\langle L,E[\texttt{true}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge idx \in dom(H(p)) \\
 & & \wedge \pi_1(H(p)(idx)) = v_1 \\
G,H,T.\langle L,E[CAS(x[idx],v_1,v_2)]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{false}]\rangle.T' \\
 & & \text{if } p \in val(x,G,L) \wedge p \in dom(H) \wedge idx \in dom(H(p)) \\
 & & \wedge \pi_1((H(p)(idx)) \neq v_1 \\
G,H,T.\langle L,E[x := v]\rangle.T' & \rightarrow_i & G',H,T.\langle L',E[done]\rangle.T' \\
 & & \text{if } (x \in dom(G) \wedge G' = G[x \mapsto \langle v,\pi_2(G(x))\rangle] \wedge L' = L) \\
 & & \vee (x \in dom(L) \wedge L' = L[x \mapsto v] \wedge G' = G) \\
G,H,T.\langle L,E[x.fd := v]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[fd \mapsto \langle v,\pi_2(H(p)(fd))\rangle]], \\
 & & T.\langle L,E[done]\rangle.T' \\
 & & \text{if } p = val(x,G,L) \wedge p \in dom(H) \wedge fd \in dom(H(p)) \\
G,H,T.\langle L,E[x[idx] := v]\rangle.T' & \rightarrow_i & G,H[p \mapsto H(p)[idx \mapsto \langle v,\pi_2(H(p)(idx))\rangle]], \\
 & & T.\langle L,E[done]\rangle.T' \\
 & & \text{if } p = val(x,G,L) \wedge p \in dom(H) \wedge idx \in dom(H(p)) \\
G,H,T.\langle L,E[\texttt{if true } s_1\ s_2]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[s_1]\rangle.T' \\
G,H,T.\langle L,E[\texttt{if false } s_1\ s_2]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[s_2]\rangle.T' \\
G,H,T.\langle L,E[\texttt{loop } s]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{inloop } s\ s]\rangle.T' \\
G,H,T.\langle L,E[\texttt{inloop } s\ E'[\texttt{break}]]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{done}]\rangle.T', \\
 & & \text{if } E' \text{ does not contain } \texttt{inloop} \\
G,H,T.\langle L,E[\texttt{inloop } s\ \texttt{done}]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{inloop } s\ s]\rangle.T' \\
G,H,T.\langle L,E[\texttt{synchronized } p\ s]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{insync } p\ s]\rangle.T', \\
 & & \text{if } T \text{ and } T' \text{ do not contain } \texttt{insync } p \\
G,H,T.\langle L,E[\texttt{insync } p\ \texttt{done}]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[\texttt{done}]\rangle.T' \\
G,H,T.\langle L,E[\texttt{done}; s]\rangle.T' & \rightarrow_i & G,H,T.\langle L,E[s]\rangle.T' \\
G,H,T.\langle L,E[\texttt{local } x = v \texttt{ in } s]\rangle.T' & \rightarrow_i & G,H,T.\langle L[x \mapsto v],E[\texttt{inlet } x\ s]\rangle.T', \text{ if } x \notin dom(L) \\
G,H,T.\langle L,E[\texttt{inlet } x\ \texttt{done}]\rangle.T' & \rightarrow_i & G,H,T.\langle L-x,E[\texttt{done}]\rangle.T' \\
G,H,T.\langle L,E[\texttt{return}]\rangle.T' & \rightarrow_i & G,H,T.\langle L,\texttt{done}\rangle.T' \\
G,H,T.\langle L,E[\texttt{return } v]\rangle.T' & \rightarrow_i & G,H,T.\langle L,\texttt{done}\rangle.T' \\
G,H,T & \rightarrow_i & G,H,T.\langle L,s\rangle, \text{ where the program declares a procedure} \\
 & & \quad p(\bar{x})\ \{s\}, \text{ and } dom(L) = \bar{x}. \\
G,H,T.\langle L,\texttt{done}\rangle.T' & \rightarrow_i & G,H,T.\langle L',s\rangle.T', \text{ where the program declares a procedure} \\
 & & \quad p(\bar{x})\ \{s\}, \text{ and } dom(L) = \bar{x}.
\end{array}
$$

Figure B.2: Transition rules of SYNL, part 2.