A System for Detecting, Preventing and Exposing Atomicity Violations in Multithreaded Programs

by

Lee Chew

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2009 by Lee Chew

A System for Detecting, Preventing and Exposing Atomicity Violations in Multithreaded Programs

Lee Chew

Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto

2009

Abstract

Multi-core machines have become common and have led to an increase in multithreaded software. In turn, the number of concurrency bugs has also increased. Such bugs are elusive and remain difficult to solve, despite existing research. Thus, this thesis proposes a system which detects, prevents and optionally helps expose concurrency bugs. Specifically, we focus on bugs caused by atomicity violations, which occur when thread interleaving violates the programmer's assumption that a code section executes atomically. At compile-time, our system performs static analysis to identify code sections where violations could occur. At run-time, we use debug registers to monitor these sections for interleaving thread accesses which would cause a violation. If detected, we undo their effects and thus prevent the violation. Optionally, we help expose atomicity violations by perturbing thread scheduling during execution. Our results demonstrate that the system is effective and imposes low overhead.

Acknowledgments

First, I would like to thank Professor David Lie for his guidance. His insight, knowledge and expertise have greatly improved this work.

Second, I would like to thank my fellow graduate students, Lionel Litty, Tom Hart, James Huang and Stan Kvasov, for their helpful comments.

I would like to also thank the Ontario Graduate Scholarship program and the University of Toronto for funding my studies.

Finally, I would to thank my family for their love and support. They have made this endeavour even more worthwhile.

Table of Contents

1 Introduction 1
1.1 Our Approach 1
1.2 Contributions 2
1.3 Structure 2
2 Related Work 4
2.1 Atomicity Violation Detectors 4
2.2 Atomicity Violation Prevention Systems 6
2.3 Data Race Detectors 6
2.4 Complementary Systems 8
3 System Overview 10
3.1 Problem Definition 10
3.2 Architecture 12
3.2.1 Static Component: ANNOTATOR 13
3.2.2 Dynamic Component: PREVENTION ENGINE 17
3.3 Limitations 22
4 Implementation 24
4.1 CIL Background 24
4.2 ANNOTATOR 24
4.2.1 CIL Analysis 24
4.2.2 CIL Transformation 25
4.3 PREVENTION ENGINE 27
4.4 Deficiencies 30
4.5 Optimizations 31

5 Evaluation 33
5.1 Experimental Set-up..33
5.2 Run-time Overhead..33
5.3 Detecting and Preventing Atomicity Violations..36
6 Future Work..38
7 Conclusion..39
Bibliography..40

List of Tables

Table 3.1: Access interleavings 11

Table 5.1: Benchmarks used 34

Table 5.2: Non-optimized overhead results 35

Table 5.3: Impact of different sources of overhead 35

Table 5.4: Effects of various optimizations 35

Table 5.5: Bug detection results 37

List of Figures

Figure 3.1: Serializable interleaving and an equivalent serial execution.................................... 10

Figure 3.2: Unserializable interleaving.. 10

Figure 3.3: System diagram 12

Figure 3.4: ANNOTATOR Diagram.. 13

Figure 3.5: Annotation example 14

Figure 3.6: Disallowed atomic regions 14

Figure 3.7: Disallowed atomic regions (longer example).. 15

Figure 3.8: Allowed atomic regions.. 16

Figure 3.9: Simplified NORMAL mode operation.. 18

Figure 3.10: Actual NORMAL mode operation 20

Figure 3.11: Why some atomicity violations are hard to expose.. 21

Figure 3.12: Making some atomicity violations easier to expose... 22

Figure 4.1: Deciding what remote access type to watch.. 25

List of Acronyms

Acronym	Definition
AR	Atomic Region
AVT	Atomicity Violation Triplet
CFG	Control-flow Graph
DFA	Data-flow Analysis
IR	Intermediate Representation
LSV	List of Shared Variables
PC	Program Counter

1 Introduction

Multi-core computers are, and will continue to be, prevalent. This is because simply increasing the clock frequency is no longer a viable means of gaining performance. In turn, programs will have to become multi-threaded in order to take advantage of this hardware and increase performance. Concurrent programs are notoriously difficult to write correctly, and give rise to numerous types of bugs. These bugs are difficult to fix because they often remain dormant, requiring a combination of two unlikely conditions. First, like regular non-threaded bugs, they require the right set of program inputs, which is exponential in number. Second, they also require the right thread interleaving, which is again exponential in number.

Both difficulties are outstanding problems, and the target of ongoing research [1,11,24,44]. Currently, most companies resort to running "stress" tests for several days, which involves running the program repeatedly with inputs from a massive test suite. The idea is that they will get the right combination of program inputs and thread interleaving on one of the runs through brute force. Not only is this very time-consuming, it is not particularly efficient at catching bugs [31].

1.1 Our Approach

Given the difficulties with finding concurrency bugs, we propose a system that protects the user from them. The system would detect the bugs, mitigate their effects and optionally help expose them. We envision two usage scenarios: 1) the developer wants to release the application with concurrency bugs in it, or 2) same as the first scenario, except the customer is willing to help improve the application – as might be the case during beta-testing. While there are numerous types of concurrency bugs, such as deadlocks and data races, we restrict ourselves to concurrency bugs caused by atomicity violations. These violations occur when the results of one thread's memory accesses are changed because they were interleaved with the memory accesses of another thread. Thus, such bugs occur because the atomicity of the accesses that was assumed by the programmer was violated. In a recent survey, approximately 70% of non-deadlock concurrency bugs were caused by atomicity violations [18].

In keeping with the two uses, our system operates in two modes: normal and bug-finding. Normal mode detects and prevents atomicity violations. First, our system conducts a conservative static analysis of the program source code, and annotates groups of memory accesses whose atomicity may be violated. Then our system executes the program, and checks for interleaving accesses made by other threads which would violate the atomicity of these groups. When such an access is about to occur, our system prevents the violation by forcing the access to execute serially with respect to the group of accesses.

Bug-finding mode does everything that normal mode does, but also tries to expose atomicity violations. We achieve this by exploiting the following insight. Atomicity violations are often hard to trigger because the memory accesses that should execute atomically usually occur temporally close together. Even if a thread could violate their atomicity, the chances of it making an access in this short interval are low. Therefore, we increase the probability by injecting delays between accesses in the group. When a violation is detected, our system reports the details of it to help the developer fix the violation.

1.2 Contributions

We are the first to implement an online system that can both detect and prevent atomicity violations. We developed a method that can do this automatically, efficiently, and using only commodity hardware. Others have focused on a low-overhead system for detecting and avoiding deadlocks [45]. Previous atomicity violation detectors have incurred high overhead [7], required user annotations [10] or were offline [31]. Previous systems that attempted to prevent violations required custom hardware [16,49]. We also extend this method to expose hidden atomicity violations and help developers fix them by providing following information: the threads involved and the memory accesses involved and their order. Finally, we measure the performance overheads of our approach, and our effectiveness at detecting bugs caused by atomicity violations.

1.3 Structure

The rest of this thesis is structured as follows. Chapter 2 discusses related work. Chapter 3 provides an overview of the system: the problem definition, our system architecture and the limitations of our approach. Chapter 4 presents our system implementation, existing deficiencies

and optimizations. Chapter 5 contains an evaluation of our system's performance overheads, and ability to detect bugs. Chapter 6 discusses future work, and we conclude in Chapter 7.

2 Related Work

Although multi-core computers have only become prevalent in the last few years, multi-threaded software, and its attendant bugs, has been around for a long time. As such, there has been a lot of related work. In this chapter, we begin by presenting research in the areas directly related to our system: detecting atomicity violations, and preventing atomicity violations. Then we cover research in the closely related area of detecting data races. Finally, we finish with research that complements our work.

2.1 Atomicity Violation Detectors

A class of detectors is based on the theory of left- and right-movers. A left/right-mover is an operation of one thread which can be moved behind or in front of, respectively, an adjacent operation (in execution order) of another thread without changing the system state. These systems start with a sequence of operations which consist of an atomic group of operations from one thread interleaved with operations from other threads. If, by commuting left- and right-movers with their adjacent instructions, it can be converted to an equivalent sequence where the atomic group of operations is not interleaved by other operations, then atomicity is preserved.

The detectors are implemented as type-based systems. Each expression in the program is assigned a *type*, which reflects its value, and an *atomicity*, which reflects its effect on atomicity. A violation is detected whenever an expression is assigned an *atomicity* of *error*. The first system was a static detector, and required the programmer to annotate all synchronization points and atomic functions [10]. A subsequent system, Atomizer [7], was a dynamic detector that could infer this information, but programs running under it suffered from slowdowns of 2.2X-50X. In contrast, our approach does not require programmer annotations, and imposes only modest overheads. Recent work extends the type system introduced above to increase expressiveness and thus decrease the number of false positives [35]. It is also able to infer type information and function atomicity without user annotations. It is aimed at complementing dynamic atomicity violation detectors by filtering out code that can be statically proven violation-free. This is because, despite its extensions, the system still suffers from a high false-positive rate.

Velodrome [9] takes a similar approach, but checks whether a sequence of atomic code sections can be serialized using the *happens-before* relation [15]. This relation states that certain events create an ordering between operations. For example, if an atomic code section A releases a lock L which another atomic code section B then acquires, then A is considered to *happens-before* B. A more precise explanation is provided in Section 2.3. As the program is running, Velodrome constructs a *happens-before* graph which orders the atomic code sections observed thus far. If a cycle is created in the graph, then an atomicity violation has occurred, because this means there is no equivalent serial execution of the atomic sections. Again, this approach requires that the programmer annotate which code sections are atomic, and suffers from slowdowns of 1.1X-72X.

Some recent detectors do not require manual annotation. SVD [48] uses the idea of computational units (CU) to approximate atomic regions. A CU is an intra-thread group of instructions that starts with a read to a shared variable, and then consists of any following instructions that are read-write or control-dependent on that initial read. As such, it can only detect half of the violations our system can. Specifically, it cannot detect ones which begin with writes to shared variables. In addition, programs running under it suffer up to 65X slowdown, partially due to the complicated dependencies it must calculate in software for every instruction. AVIO [19] detects atomicity violations of memory access pairs. It first executes training runs on programs to differentiate between benign violations and malignant ones. Then it builds a database of malignant violations indexed by the second access of a memory access pair. The authors provide both a hardware and software implementation. The hardware solution imposes very little overhead (0.4-0.5%), but requires custom hardware. The software solution imposes very high overheads (programs running under it suffer up to 25X slowdown), as it must record every access to a shared variable, and for every access it must perform a lookup to see if a malignant violation has occurred. Finally, CTrigger [31] uses the same insight as our system with regards to exposing atomicity violations. It takes as input an execution trace and identifies sets of instructions which may participate in atomicity violations. Afterwards, it removes sets which cannot lead to atomicity violations due to synchronization such as locks or barriers. Then it re-executes the program with the same inputs, and injects delays between instructions in each set to increase the likelihood of a violation. Although our techniques are similar, our goals are different – CTrigger is primarily attempting to expose atomicity violations offline, while we are

primarily attempting to detect and mitigate them online. As a result, low performance overheads are not a concern for their system, whereas it is an important consideration for our system.

2.2 Atomicity Violation Prevention Systems

Atom-Aid [16] is a hardware system that attempts to decrease the probability of atomicity violations occurring. It builds upon recent architectural proposals, such as BulkSC [2] and ASO [46], that arbitrarily execute groups of instructions atomically. Atom-Aid extends this by actively grouping instructions. It looks for suspicious access patterns to shared variables that indicate an atomicity violation is possible. It then dynamically groups future accesses to those shared variables on-the-fly into *chunks*, which are guaranteed by the hardware to execute atomically. In contrast, our system deterministically prevents atomicity violations, and does not require custom hardware.

The system in [49] proposes to prevent atomicity violations at run-time by restricting the number of possible thread interleavings. The system requires an initial training phase, where the target program is executed several times and its memory accesses are recorded. Based on these traces, it generates *Predecessor Set* constraints. Each constraint records, for a particular memory instruction, the set of memory instructions made by other threads that must immediately precede it. At run-time, an atomicity violation occurs if a memory access not in this set executes immediately before the particular instruction. When the system detects a violation, it repairs the violation by either delaying the violating memory access or using a checkpoint-and-rollback mechanism. The system was implemented using the dynamic binary rewriter Pin [17], which incurred 100x-200x run-time overhead. In order to achieve reasonable overheads, the authors stated that caches would have to be modified to record last-access information for every memory location, and that cache-coherency protocols would have to be extended to transmit this information. Our system is able to achieve low overhead with existing commodity hardware.

2.3 Data Race Detectors

A data race occurs when two threads access a shared memory location without any synchronization, and at least one of the two accesses is a write. Although it is closely related to atomicity violations, they are not the same. Not all data races are atomicity violations. For example, a data race only requires two accesses – one from each thread, whereas an atomicity

violation requires three accesses – two by one thread interleaved with a third access by another thread. Conversely, not all atomicity violations are data races. For example, each of the three accesses in an atomicity violation could be individually protected by a lock (i.e., the lock is acquired immediately before the access and released immediately after), and thus would be data-race-free. There are three general approaches to detecting these types of bugs: those based on the lockset algorithm, those based on the *happens-before* relation, and those based on a hybrid of the two.

The basic lockset algorithm determines whether all accesses to shared variables are protected by some non-empty set of locks. Each shared variable V has a lockset L_v, which initially consists of all locks in the program. Whenever a thread accesses a shared variable, the variable's lockset is updated to the intersection of itself with the locks currently held by the thread. If a lockset of any variable becomes empty, then a warning is raised about a possible data race. The algorithm was first implemented as a dynamic detector in Eraser [36], and then as a dynamic detector with static optimizations such as filtering out statements which can never race in [3]. The algorithm has also been implemented as a static detector in RacerX [6], which ranks possible data races according to metrics derived from statistical analysis. These included whether the race occurred in a multithreaded section of code and whether the variable actually needed to be protected.

The *happens-before* relation is a partial order on events in a distributed memory system. It can trivially be adapted to provide a partial order on operations performed by a multi-threaded program. The basic concept is that certain events can order the operations made by different threads. For example, if it is observed during execution that thread A releases lock M and then thread B acquires lock M, then all memory accesses made by thread A prior to the release of M *happens-before* all memory accesses made by thread B after the acquisition of M. Other examples of ordering events include thread A creating thread B, or thread A calling *signal* on a semaphore S that thread B is *waiting* on. A data race is detected when a pair of accesses made by two threads, with at least one being a write, is not ordered by *happens-before*. CHESS [25] performs "stress" tests on small sections of code, each time choosing a different thread schedule, and uses the *happens-before* relation to detect data races. Recently, FastTrack [8] found that, for the majority of the time, the amount of data which had to be kept about past memory accesses could be made constant (as opposed to linear with the number of threads). This space

optimization in turn led to performance optimizations, as certain comparison operations now occurred in constant time as opposed to linear time.

Finally, some detectors are a combination of the two approaches. RaceTrack [50] and MultiRace [32] first use the lockset algorithm to generate a list of possible data races. Then they use the *happens-before* relation to filter out races which could not occur because the memory accesses involved were ordered. The system in [29] took a similar approach, except it used a simplified version of the *happens-before* relation which ignored certain ordering events, such as lock acquires/releases. It gained performance at the expense of missing some data races. RaceFuzzer [37] is an improvement on that system and only reports data races which lead to errors. It takes as input the set of possibly racing pairs of statements from the latter system. For each pair of statements, it runs the program, and whenever a thread is about to execute one of the statements in the pair, it is frozen. When both statements have a frozen thread about to execute it, then the system randomly lets one thread proceed. If an error or exception is generated afterwards, then this is a data race.

2.4 Complementary Systems

There has been research into the prevention of data races – either preventing them from occurring, or preventing them from damaging the system. Transactional memory [12] provides support for preventing data races from being visible to the system. The programmer annotates code regions that access shared data as transactions. Transactions speculatively execute, and commit their changes (i.e., make their changes visible to the system) only if there are no conflicts. If conflicts exist, the transaction is re-executed again and again, until it is able to commit without conflicts. Some recent examples include RSTM [38] and McRT-STM [34]. Autolocker [20] aims to prevent data races from ever occurring. Programmers annotate which locks must protect which shared variables, and which code sections are transactions (these are the same as in transactional memory). The system then converts the transactions into lock-based code which is guaranteed to be free of data races. Essentially, the tool converts optimistic concurrency into pessimistic concurrency.

Given the difficulty in detecting data races and atomicity violations, there has also been research into making debugging such problems easier. Kendo [30] proposes new lock and unlock functions that ensure threads acquire locks in a deterministic fashion. Although it makes

it easier to reproduce some results of multithreaded programs, it is not applicable to solving data races or atomicity violations as these are due to missing synchronization operations such as locks. DMP [4] is a hardware system which guarantees that a multithreaded program will behave in the same way given the same inputs. It allows multiple threads to execute in parallel, and only if determinism is compromised will threads be re-executed using transactional memory hardware. Race/replay systems allow programmers to record a program's execution into a log, and then later on replay the program from it. There are both hardware implementations such as Delorean [22], Strata [27] and Rerun [13], and software implementations such as ReVirt [5] and Flashback [39]. However, since cache coherency messages represent the only way to efficiently capture the ordering between thread operations, only hardware solutions have low enough overhead for multithreaded applications.

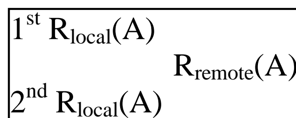
We conclude by presenting recent work which tries to make debugging easier, and can sometimes prevent data races from occurring. ReEnact [33] implements lightweight data race monitoring on top of thread-level speculation hardware. When a data race is detected, it can replay the system from a point before the bug occurred, with extra instrumentation to create a bug signature for the programmer. ReEnact would also theoretically be capable of fixing data races by replaying the system again from a point before the bug occurred, and delaying threads to prevent the buggy ordering. However, the authors only discussed, but did not implement, such a feature.

3 System Overview

This chapter aims to give a high-level overview of our system. We first define the problem. Then, we discuss our system architecture. Finally, we finish with a discussion of the limitations of our approach.

3.1 Problem Definition

Consider an execution of a multithreaded program P, where a thread T in P makes a sequence S of two or more accesses to shared variables. The accesses in S are interleaved with accesses to the same variables by other threads in P. The outcome of S is the set of values composed of the value read by every read in S and the final value of each shared variable after the last write to it in S. This interleaving is unserializable if there is no serial execution of S and the accesses by other threads such that the outcome of S is the same. We define this situation as an atomicity



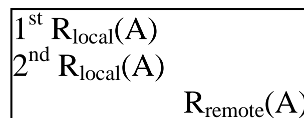


Figure 3.1: Serializable interleaving and an equivalent serial execution

1st $R_{local}(A)$	
	$W_{remote}(A)$
2nd $R_{local}(A)$	

Figure 3.2: Unserializable interleaving

violation with respect to S, or simply as an atomicity violation when the context is clear. In particular, our system addresses atomicity violations in the simplified case where S contains only two memory accesses to a shared variable, and they are interleaved by a single access to the same shared variable by another thread. We refer to T as the *local thread*, and its accesses to shared variables as *local accesses*. Any other thread is a *remote thread*, and its accesses to shared variables are *remote accesses*. A *remote access* that violates the atomicity of S is called a *violating access*. The two *local accesses* in S are called the *local access pair*. When combined with the *violating access*, they form the *atomicity violation triplet* (AVT) that characterizes an atomicity violation.

Table 3.1 lists the eight different possible interleavings of *local accesses* and *remote access*, including the four which are AVT's. In Table 3.1, A is a shared variable, $R_{local}(A)$/ $R_{remote}(A)$ denote a *local/remote read* of variable A, and $W_{local}(A)$/ $W_{remote}(A)$ denote a

local/remote write to variable A. Let us consider both an example of a serializable interleaving and an unserializable interleaving from Table 3.1 to illustrate the difference. In Figure 3.1, the left box contains a serializable interleaving and the right box contains one possible serial execution of the interleaving. The interleaving is serializable because the outcome of the *local accesses*, the two *local reads*, is the same in both cases. Figure 3.2 contains an unserializable interleaving, because there is no serial execution of the accesses such that the outcome of the *local accesses* is the same. To prove this, we note that there are only two alternative serial executions: either move the *remote write* before the first *local read* or after the second *local read*. This will cause the first and second *local read* to respectively read a different value.

Type	Interleaving	Equivalent Serial Execution
Serializable	$R_{local}(A)$ $R_{remote}(A)$ $R_{local}(A)$	$R_{remote}(A)$ $R_{local}(A)$ $R_{local}(A)$
	$R_{local}(A)$ $R_{remote}(A)$ $W_{local}(A)$	$R_{remote}(A)$ $R_{local}(A)$ $W_{local}(A)$
	$W_{local}(A)$ $R_{remote}(A)$ $R_{local}(A)$	$W_{local}(A)$ $R_{local}(A)$ $R_{remote}(A)$
	$W_{local}(A)$ $W_{remote}(A)$ $W_{local}(A)$	$W_{remote}(A)$ $W_{local}(A)$ $W_{local}(A)$
Unserializable	$R_{local}(A)$ $W_{remote}(A)$ $R_{local}(A)$	N/A
	$R_{local}(A)$ $W_{remote}(A)$ $W_{local}(A)$	N/A
	$W_{local}(A)$ $W_{remote}(A)$ $R_{local}(A)$	N/A
	$W_{local}(A)$ $R_{remote}(A)$ $W_{local}(A)$	N/A

Table 3.1: Access interleavings

3.2 Architecture

Our system has two goals: the primary one is to detect and prevent atomicity violations, and the secondary one is to expose atomicity violations. The system has two major components, which are shown in Figure 3.3 in the solid-border boxes. The ANNOTATOR is a static component which annotates all *local access pairs* in the source code – these are the potential victims of atomicity violations. The compiler, which is outside of our system, then compiles the annotated code normally. Finally, the PREVENTION ENGINE, which is a dynamic component, executes the compiled program. It can operate in two modes, depending on our goals. The first mode achieves the primary goal. For the period between the first *local access* and the corresponding

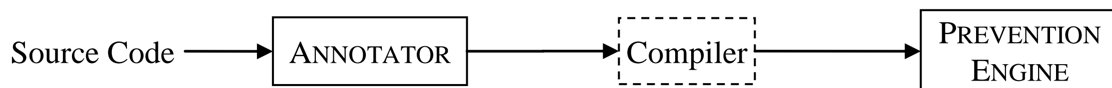


Figure 3.3: System diagram

second *local access*, it detects atomicity violations by monitoring the shared variable being accessed by the first *local access* for *violating accesses*. If it detects that such an access is about to occur, the PREVENTION ENGINE prevents the atomicity violation by converting what would have been an unserializable interleaving of the *local access pair* and *violating access* into a serial execution of the accesses. The second mode achieves both the primary and secondary goals. Thus in addition to the aforementioned functionality, it will also do the following. When it encounters the first *local access* of a *local access pair*, the PREVENTION ENGINE introduces a delay to increase the probability of exposing an atomicity violation. When an atomicity violation occurs, it will report the details to the developer.

3.2.1 Static Component: ANNOTATOR

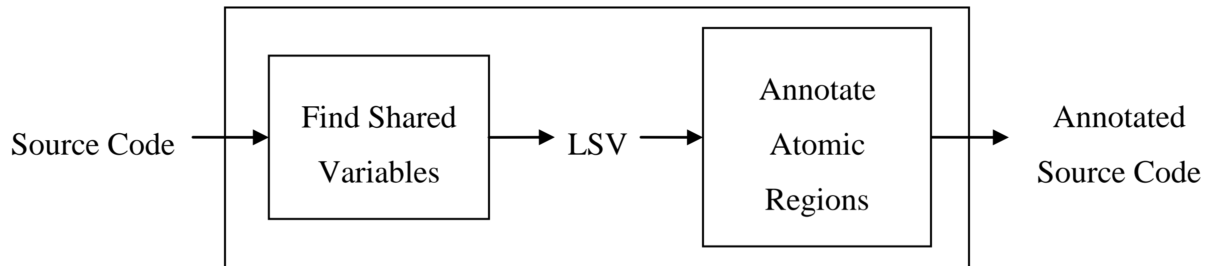


Figure 3.4: ANNOTATOR Diagram

The purpose of the ANNOTATOR is to annotate *local access pairs* in the source code, and is accomplished in the two steps shown in Figure 3.4. The first step is to generate a conservative list of shared variables (LSV); being conservative, some variables on this list may not actually be shared. Henceforth, we will refer to variables on this list, and in general variables which could be shared, as shared variables for brevity. In the second step, for each variable in the LSV, the ANNOTATOR searches for pairs of *local accesses* to it. For each pair it finds, it marks the code between the two accesses inclusive as an *atomic region*. As such, each *atomic region* is associated with a variable in the LSV. The beginning of the region is marked with a *begin_atomic()* function call such that if the function is called, then the first *local access* is guaranteed to occur afterwards. The end is marked with an *end_atomic()* function call such that if the second *local access* occurs, then the function is guaranteed to be called afterwards.

Figure 3.5 gives an example of how these annotations are placed. Here, it is assumed that *global_counter* is on the LSV. The ANNOTATOR has found a *local access pair*: the read of *global_counter* on line 2 and the write to *global_counter* on line 4. Thus, it marks all the code between the two accesses inclusive – the code on lines 2, 3 and 4 – as an *atomic region* using the *begin_atomic()* and *end_atomic()* functions. More detail about what these two functions do is presented in the next section. The parameters of these two functions are presented in the next chapter, and will be elided in all figures until then.

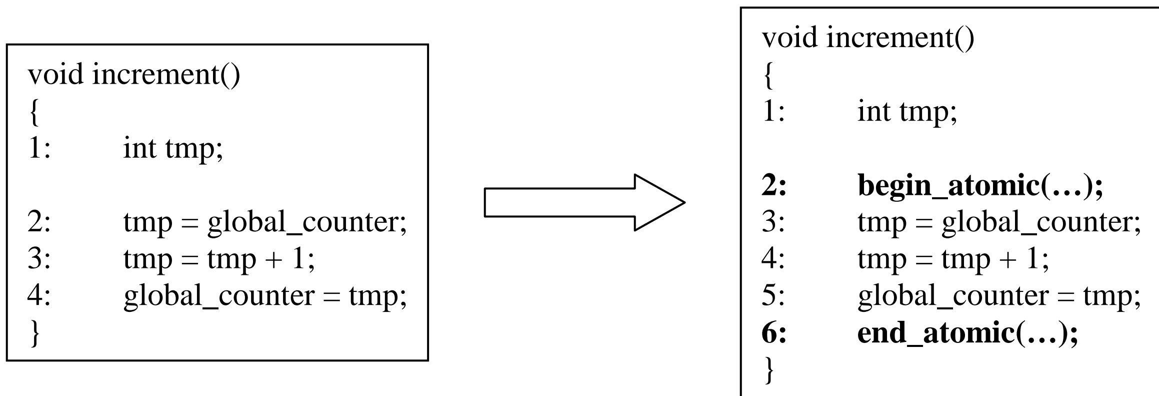


Figure 3.5: Annotation example

A *local access* can usually be the first *local access* of a *local access pair* at most once, because otherwise *atomic regions* could have more than two *local accesses*, and thus fall outside of the type of atomicity violation our system addresses. We do not extend our problem definition to include such cases because the number of *atomic regions* our system must monitor and manage at run-time is $O(N^2)$, where N is the number of *local accesses* in the *atomic region*. As an example of why this rule is applied, refer to the short code snippet in Figure 3.6, where A represents a shared variable and AR1, AR2 and AR3 are *atomic regions*. There should only be two *atomic regions*: AR1 and AR2. However, by allowing the read on line 3 to be the first *local access* more than once, there are three *atomic regions*: AR1, AR2 and AR3. Figure 3.7 shows a longer code snippet which demonstrates the quadratic growth of the number of *atomic regions*.

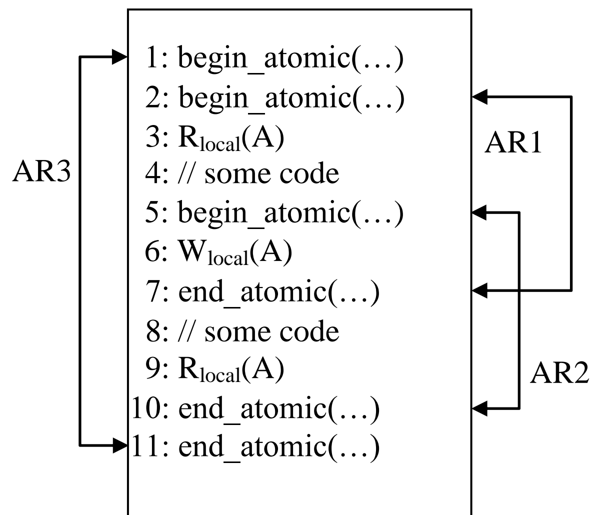


Figure 3.6: Disallowed atomic regions

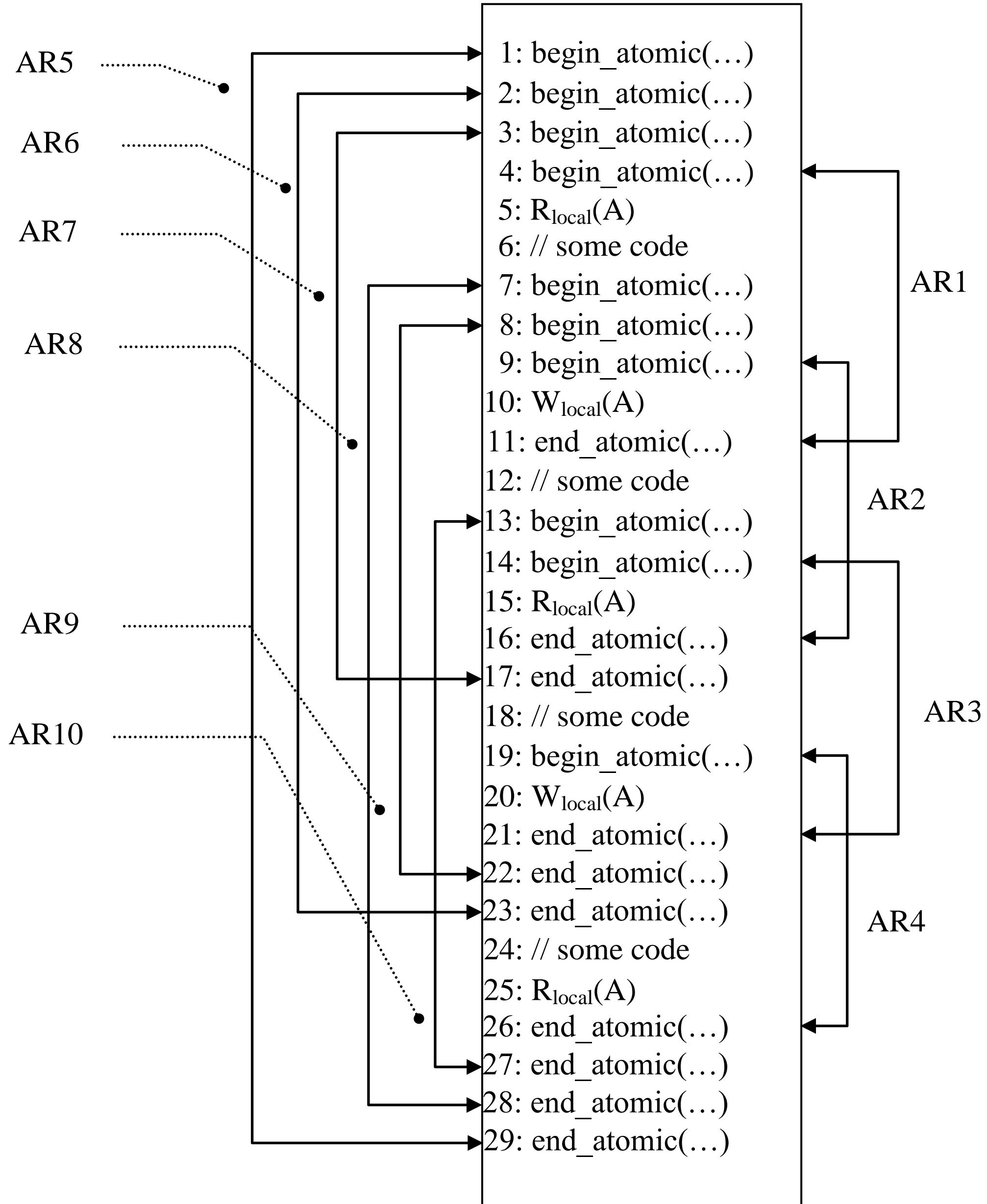


Figure 3.7: Disallowed atomic regions (longer example)

The only exception to this rule is if a *local access* can form a *local access pair* with multiple other *local accesses*. This occurs when: 1) a *local access* is in a basic block BB which has multiple immediate successors, 2) more than one of BB's successors contain a *local access* to the same shared variable and 3) there are no intervening *local accesses* to the same shared variable. Figure 3.8 presents such an example – notice that AR1 and AR2 only have 2 *local accesses*, as required. By the same logic, a *local access* can usually be the second *local access* of a *local access pair* at most once. There is a similar exception to this rule when multiple other *local accesses* can form a *local access pair* with this *local access*. Note that, because of these rules and our algorithm, all *atomic regions* have exactly two *local accesses* to the same shared variable in them.

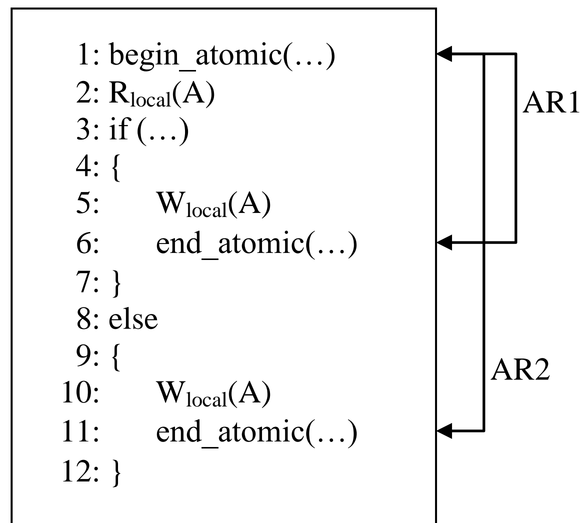


Figure 3.8: Allowed atomic regions

Our system does not use on the ANNOTATOR to annotate *remote accesses* to shared variables, as the task would be difficult using static analysis. Instead, we use the PREVENTION ENGINE to detect them and determine if they are *violating accesses* at run-time. The PREVENTION ENGINE is presented in the next section.

As this component is based on static analysis, in the general case, it cannot determine with certainty whether a particular variable is shared or whether a particular *local access* will occur at run-time. The result is that the annotations are conservatively placed. This does not affect the correctness of our approach, since we detect atomicity violations at run-time. It only hurts the performance of the PREVENTION ENGINE due to unnecessary annotations.

3.2.2 Dynamic Component: PREVENTION ENGINE

The PREVENTION ENGINE operates in one of two modes: NORMAL, which is the default, or BUGFINDING. In NORMAL mode, the PREVENTION ENGINE detects and prevents atomicity violations from occurring. We initially present its operation under the simplifying assumption that when a *begin_atomic()* is called, its corresponding *end_atomic()* will also eventually be called and vice-versa, as shown in Figure 3.9. First, it runs the program. When a thread in the program calls *begin_atomic()*, the PREVENTION ENGINE starts monitoring accesses to the shared variable associated with this *atomic region*. This is done through hardware registers for efficiency reasons - the exact implementation is discussed in the next chapter. If a *remote thread* is about to make a *violating access*, the access is delayed until after the *atomic region*. This is done by first freezing the *remote thread*. When the *local thread* calls *end_atomic()*, we stop monitoring the shared variable and the *remote thread* that was frozen is awoken. It is allowed to make its access, which now cannot violate the atomicity of the *atomic region*. Then the program proceeds to execute normally. If during the *atomic region* no *remote thread* is ever about to make a *violating access*, then the only action taken when the *local thread* calls *end_atomic()* is to stop monitoring the shared variable.

Although not shown in Figure 3.9 for simplicity, the PREVENTION ENGINE supports nested *atomic regions*. If an *atomic region* AR2 is started by the *local thread* during an *atomic region* AR1 that was also started by the *local thread*, then the operation of the PREVENTION ENGINE for AR2 follows the diagram in Figure 3.9. If an *atomic region* AR2 is started by a *remote thread* during an *atomic region* AR1 started by the *local thread*, and AR2 is not associated with the same shared variable, then the operation of the PREVENTION ENGINE for AR2 again follows the diagram in Figure 3.9. If AR2 is associated with the same variable, then it is shifted to occur after AR1. This is done by treating the particular *begin_atomic()* call as a *violating access*. It is acceptable because the ANNOTATOR inserts the call to *begin_atomic()* such that if it is called, then the first *local access* is also guaranteed to occur afterwards. Thus, the thread to which AR2 belongs will be frozen until after AR1's *end_atomic()*.

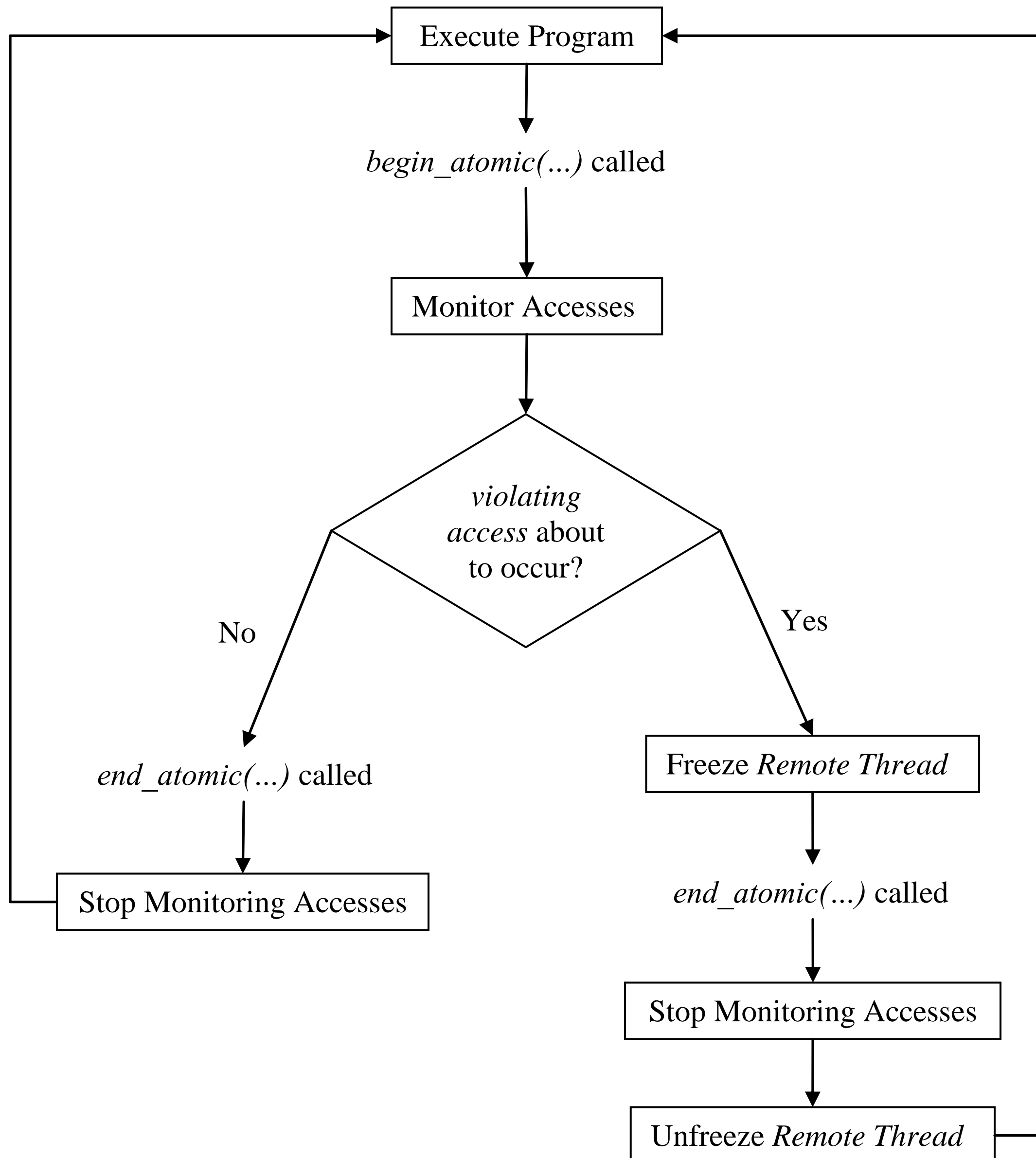


Figure 3.9: Simplified NORMAL mode operation

Unfortunately, our simplifying assumption is unrealistic. When the ANNOTATOR annotates a *local access pair* with *begin_atomic()* and *end_atomic()*, it is not possible in the general case to statically determine whether either function call will occur at run-time. There are two cases to consider:

1. ***begin_atomic*() is not called** – This is not a problem because, as shown in Figure 3.9, the PREVENTION ENGINE only begins detection and prevention after *begin_atomic()* is called. It is irrelevant whether *end_atomic()* is called in this case, since it will do nothing.

2. ***begin_atomic()* is called and *end_atomic*() is not called** – This is a problem for two reasons. The first reason is that we can only prevent the atomicity violation when the *remote access* is about to occur, but we only know if the *remote access* is a *violating access* when the second *local access* occurs, and by extension the *end_atomic*() is called. Thus, the PREVENTION ENGINE is augmented as shown in Figure 3.10. Instead of freezing the *remote thread* when it makes a *remote access* until *end_atomic()* is called, it is frozen for a set period of time. If *end_atomic*() is called within this period, then everything precedes as before. If *end_atomic*() is not called within this time, the *remote thread* unfreezes. If the *end_atomic*() eventually gets called, the PREVENTION ENGINE will record that an atomicity violation occurred but it could not be prevented. If the *end_atomic*() never gets called, then we do nothing. The second reason is that we monitor the shared variable until *end_atomic()* is called. If *end_atomic()* is never called, the unnecessary monitoring introduces performance and resource overheads with no benefit. For now, we assume there is a way to stop monitoring regardless of whether *end_atomic()* is called. The exact method is detailed in the next chapter. In Figure 3.10, this is why we can stop monitoring when no *violating access* is about to occur.

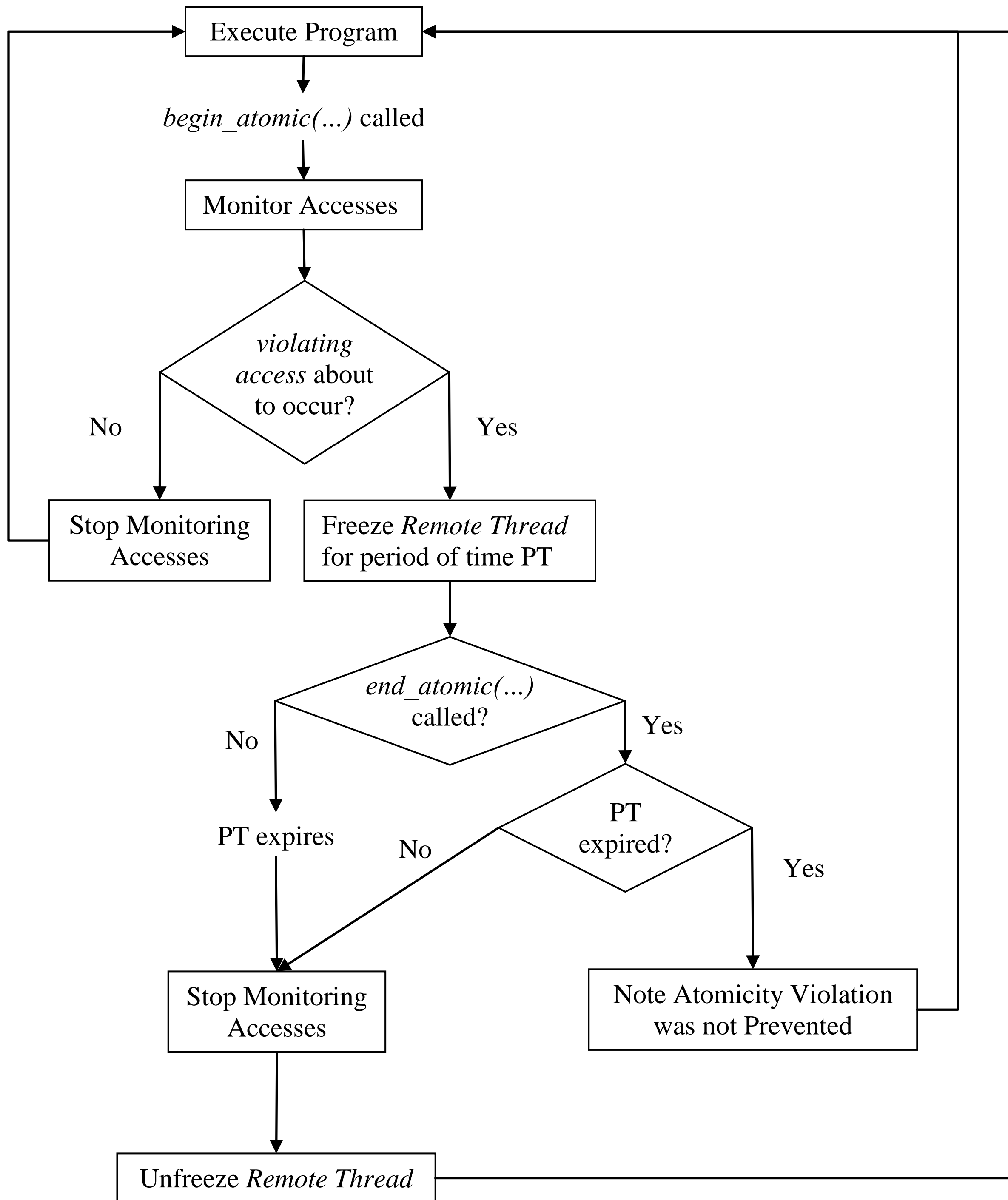


Figure 3.10: Actual NORMAL mode operation

In BUGFINDING mode, the PREVENTION ENGINE tries to expose atomicity violations. As in NORMAL mode, it still detects and prevents atomicity violations. The only differences are that the PREVENTION ENGINE increases the chances of an atomicity violation occurring in the program and reports debugging information to the developer. To achieve this, we use the intuition from Chapter 1: atomicity violations are hard to expose because the *local accesses* in an AVT usually execute together – that is, no thread is scheduled to run in between the accesses. We illustrate this in Figure 3.11. There is an atomicity violation involving the first and second *local access*, but the gap between the accesses is small. For most executions, during testing for example, the

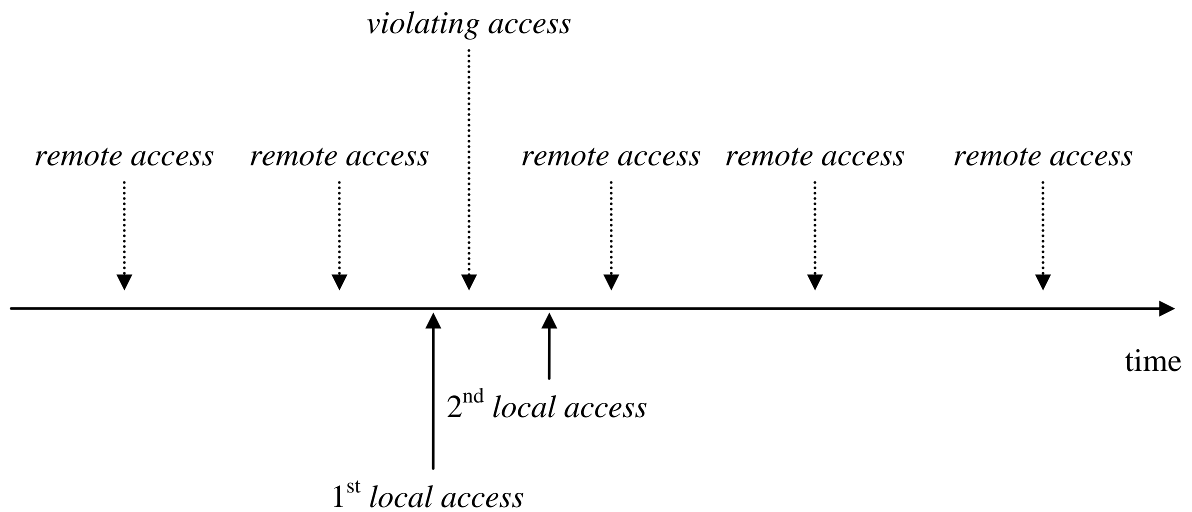


Figure 3.11: Why some atomicity violations are hard to expose

accesses made by *remote threads* fall outside this gap and remain *remote accesses* and not *violating accesses*. Thus, we inject an artificial delay between the first and second *local access*, leading to the situation shown in Figure 3.12. This is done by freezing the *local thread* for a set period of time when *begin_atomic()* is called, but letting all the *remote threads* execute normally. Although this injects a delay before the first *local access,* which is different from our conceptual idea of injecting a delay between the accesses, it accomplishes the same goal. This is because if we detect a *remote access* during this freeze period, then it would have been possible for that *remote access* to occur between the two accesses in the *local access pair*. In order to report debugging information, after we've detected an atomicity violation we record the accesses which make up the AVT and the ID's of the *local* and *remote threads* that participated in the violation.

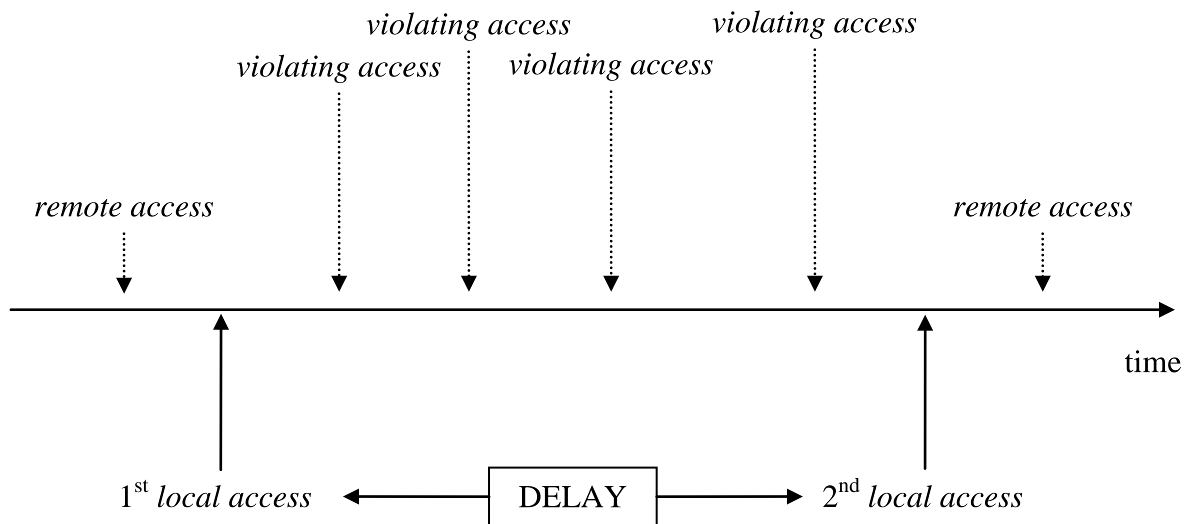


Figure 3.12: Making some atomicity violations easier to expose

3.3 Limitations

In BUGFINDING mode our system is unable distinguish between benign atomicity violations – those that do not cause bugs or that the programmer intended to occur – and malignant ones. Thus, we report all atomicity violations that our system detects at run-time. However, we may miss violations, and in turn, any bugs caused by them, under various conditions:

1. **The right inputs are not provided** - Our system does not influence the programs' control flow. As such, if the program does not execute a section of code that contains an atomicity violation because the required inputs were not provided, then our system will not report the violation.

2. **The right thread interleaving does not occur** – Our system does not influence the program's thread schedule outside of freezing threads when they call *begin_atomic()*, and when they make *violating accesses*. If an atomicity violation exists, but no remote access was made during the atomic region (e.g., the remote thread made the *violating access* before the atomic region started), then our system will not report a violation.

3. **Pointer aliasing** – In the general case, it is not possible to statically determine whether two pointers point to the same memory location. As such, our system misses AVT's which

involve aliased pointers. For example, if the first *local access* was to *ptr,* and the second *local access* was to *possible_alias_of_ptr*, the ANNOTATOR will not consider them a *local access pair*.

4. **Multiple executions of a statement** – The ANNOTATOR only considers pairs of accesses where the first access is not the same program statement as the second access. For example, in a *while*-loop, it will not form a pair of *local accesses* from the loop's condition statement with itself. Thus, our system misses any atomicity violations involving such AVT's.

4 Implementation

This chapter presents our system implementation. First, we give background on a text replacement tool called C Intermediate Language (CIL), and then describe how we implemented the ANNOTATOR using it. Next, we present the implementation of the PREVENTION ENGINE as a modified Linux kernel. Following that, we discuss the existing deficiencies of our implementation, and finally conclude with optimizations.

4.1 CIL Background

CIL [28] is a tool that simplifies analysis of C programs and can perform source-to-source transformations on them. It works in three stages. First, it parses the source code of a program and constructs an intermediate representation (IR) of it. The IR is similar to an abstract syntax tree. Second, it applies analyses/transformations – OCaml programs written by the user – on the IR. Lastly, it converts the IR back into C source code, which can then be passed to a compiler. It supports almost all GCC extensions – enough to successfully process the Linux kernel.

4.2 ANNOTATOR

As mentioned in Chapter 3, this component is responsible for annotating *atomic regions*. It has two sub-components: a CIL analysis which constructs the LSV, and then a CIL transformation that takes the LSV and inserts annotations around *atomic regions*.

4.2.1 CIL Analysis

This OCaml program constructs the LSV in three steps:

1. **Global Variables –** CIL automatically identifies these, so they are simply copied from CIL's built-in list.

2. **Pointer arguments** – CIL automatically builds function signatures for every function, so pointer arguments are added to the LSV by simply reading from these signatures.

3. **Others** - For each function in the program, our analysis constructs a control flow graph (CFG). Using the CFG, it performs data-flow analysis (DFA) and tracks which variables are potentially shared. We are conservative, and treat any variable as shared unless we can

show otherwise. These potentially shared variables include pointers to pointers (and pointers to pointers to pointers, etc.) local to the function that are passed to another function, or pointers assigned the return value of another function, pointers assigned a pointer already in the LSV, and variables accessed by dereferencing pointers already in the LSV (e.g., for *shared_pointer→data*, *data* is considered to be shared)

4.2.2 CIL Transformation

This OCaml program takes the LSV, and applies the following algorithm for each function in the program source code. First, it constructs the CFG. Then, it uses the CFG and performs path-insensitive DFA, tracking where shared variables are accessed (i.e., the program statement) and the type of access. At the end of each DFA-iteration, it tries to form intra-procedural *local access pairs* by matching each shared variable accessed in the current statement with accesses to that variable in a statement previously visited in the DFA. The decision to restrict our search to intra-procedural pairs was based on empirical evidence. Our system was able to detect all the bugs in our handcrafted corpus of bugs from large, mature and widely-used applications, without having to rely on the complexity of inter-procedural analysis. The corpus is presented in the next chapter.

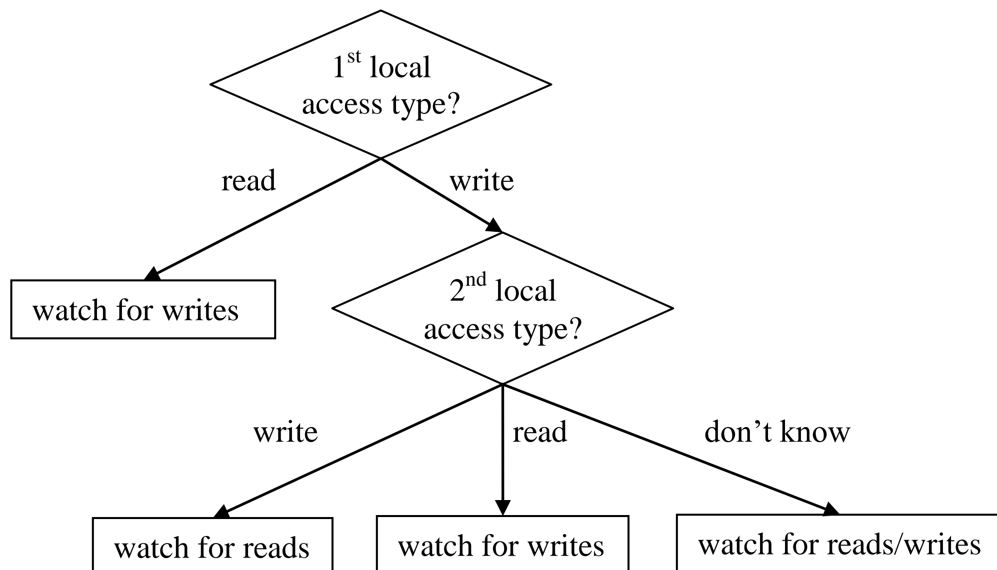


Figure 4.1: Deciding what remote access type to watch

For each such pair, the transformation inserts a *begin_atomic*() function call immediately before the previous access (i.e., first *local access*). The function *begin_atomic*() requires six parameters: the address of the shared variable, the size of the shared variable, the first *local access*' access type, the type of *remote access* to watch for, a globally unique *atomic region* ID, and a globally unique function ID. The first three parameters are obtained from the information CIL generates by default about the source code. The type of remote access to watch is determined by the decision tree in Figure 4.1. The decisions are driven by the question of what atomicity violations are possible. For example, if the first *local access* is a read, then the only possible *violating access* in the two applicable AVT's ([*local read*, *remote write*, *local write*] and [*local read*, *remote write*, *local read*]) is a *remote write*, and thus we only need to watch for writes. The *atomic region* ID is unique for each *local access pair* identified by the ANNOTATOR. The function ID is unique for each function in the source code. The reasons for these parameters will be presented in the next section. Then it inserts an *end_atomic*() function call immediately after the current access (i.e., second *local access*) such that the former will execute if the latter does. The *end_atomic()* function requires two parameters: the second *local access*' access type and an *atomic region* ID. The arguments are obtained in the same way as they were for the *begin_atomic()* function. Again, the reason for these parameters will be presented in the next section.

As mentioned in the previous chapter, a *begin_atomic()*'s corresponding *end_atomic()* may not execute at run-time, and thus we need an alternative method to stop monitoring shared variables in such a case. Since our implementation restricts our search to intra-procedural points, we can simply stop monitoring when a function exits. The ANNOTATOR inserts a call to *clear_ar_data()* at all function exit points. This function takes one parameter: a globally unique function ID, which is generated in the same way as the one provided to *begin_atomic()*. More detail about this function is presented in the next section.

It should be emphasized that imprecision in the analysis of either sub-component of the ANNOTATOR does not affect correctness, since we only detect, and optionally report, violations that occur at run-time.

4.3 PREVENTION ENGINE

This portion of our system is a modified Linux kernel. The functions *begin_atomic()* and *end_atomic()* are implemented as system calls, and the freezing of a thread is achieved by modifying the kernel scheduler to remove the thread from the runqueue. Crucial to the success of our system was a low-overhead method of detecting accesses to a memory location. Our initial implementation had used page permissions to detect *remote accesses*. For example, to detect writes, our system would mark the page containing the shared variable as read-only. When a page fault occurred, the faulting address was compared to the address of the shared variable to determine if a *remote access* had occurred. It was able to support the simultaneous monitoring of any number of shared variables. However, the coarse granularity of detection meant there were many unnecessary page faults that occurred from accesses to variables that were on the same page, but were not the shared variables we were monitoring. This imposed high performance overheads and was incompatible with our goal of a low-overhead online system. Thus, our current implementation uses the four data debug registers present on modern Intel/AMD processors to detect *remote accesses*. These registers allow the kernel to monitor a memory location for reads, writes or both. A promising alternative, which we may use when it is implemented in production hardware, is Mondrian Memory Protection [47]. It combines the benefits of both the page permission and debug register approach by allowing for low-overhead hardware monitoring of any number of memory locations.

The PREVENTION ENGINE is responsible for detecting, preventing and optionally exposing atomicity violations. We will present its implementation in the context of a hypothetical program running on top of it. Steps that only occur in BUGFINDING mode are labelled as such. When a thread in the program calls *begin_atomic*(), the following happens:

1. Check if we've used up all the data debug address registers, because we allow for nested *atomic regions*. If not, set one of them to the provided address of the shared variable; otherwise, return.

2. Set the relevant bits in the debug control register to reflect the provided remote access type and the provided shared variable size. Then enable exact breakpoints, which cause the processor to trap immediately after the instruction which accessed the memory location

specified in the debug address register (i.e., after a *remote thread* has performed a *remote access*). In some cases, this is not possible [14]. We discuss this situation in the next section.

3. If we are detecting remote writes, save the value of the shared variable as OLD_VALUE. Otherwise, do nothing.

4. Our system has a per-thread table that records the *atomic regions* which have not yet ended. This table is indexed by the pair [*atomic region* ID, function ID]. Register our *atomic region* with the provided *atomic region* ID and function ID by inserting a new entry into this table.

5. [BUGFINDING mode] Set the state of the current thread (i.e., the *local thread*) to TASK_INTERRUPTIBLE, which makes the thread non-schedulable (i.e., frozen). We also set a freeze period counter $FPCount_{local}$ to the length of the freeze period. Now return.

6. [BUGFINDING mode] All other threads in the program are allowed to run as normal. The kernel scheduler keeps track of how long threads have been running by default, and we modified this portion to also keep track of how long the *local thread* has been frozen. After the freeze period has expired, the state of the *local thread* is set to TASK_RUNNING and it is now allowed to execute. It should be noted that the shared memory location is still being monitored, and only stops being monitored when *end_atomic()* or *clear_ar_data()* is called.

If a *remote access* is detected, we do the following:

1. Record its access type.

2. Unfortunately, accesses to the monitored memory location generate a trap, and not a fault. This means that the access has already occurred, and the program counter (PC) has been updated to the next instruction. Thus, we cannot prevent the access from occurring, and can only undo its effects. At this point, we cannot determine whether the *remote access* is a *violating access*, because it is known only when the second *local access* occurs. However, by that time, the results of the access could have been used by the program.

Thus, we must pre-emptively undo the effects of the access now. If the *remote access* was a write, we can undo its effects by simply saving the current value of the shared variable as NEW_VALUE and then writing OLD_VALUE to the shared variable. Otherwise, if the *remote access* was a read, then it becomes more complicated. We defer discussion of this until the next section.

3. Set the state of the current thread (i.e., the *remote thread*) to TASK_INTERRUPTIBLE, which makes the thread non-schedulable. We also set a freeze period counter $FPCount_{remote}$ to the length of the freeze period: ($FPCount_{local}$'s current value + 1) quanta. If *end_atomic()* does not awaken it, the *remote thread* is awoken in the same way as the *local thread*, as described above.

If a thread calls *end_atomic()*, the following occurs:

1. First, check whether the corresponding *begin_atomic()* was called by comparing the provided *atomic region* ID with all stored *atomic region* ID's which belong to the thread. If there is a match, proceed to the next step. Otherwise, return.

2. We check if a *violating access* occurred by checking whether the right type of *remote access* was made. If the *remote thread* which made this access is no longer frozen (i.e., $FPCount_{remote}$ has already expired), then we note that an atomicity violation occurred which was not prevented. Otherwise, if the *violating access* was a write, write NEW_VALUE into the shared memory location. In effect, this shifts the *violating write* after the *atomic region*. Note that when the *violating write* occurred, we only undid its effects. In the case of multiple *violating writes*, NEW_VALUE will contain the value of the most recent *violating write*.

3. Clear the appropriate fields of the data debug address and control registers for this *atomic region*. Then wake-up all frozen *remote threads* by setting their thread states to TASK_RUNNING.

4. [BUGFINDING mode] For each violation detected, report the thread ID's of the threads involved and the PC of each instruction involved and their order.

If a thread calls *clear_ar_data()*, we clear the appropriate fields of the data debug address and control registers for all *atomic regions* belonging to the calling thread that have the same function ID as the provided function ID.

4.4 Deficiencies

As mentioned before, if the *remote access* was a read, it becomes complicated to undo its effects because we do not know the original value stored at the read's destination, or even the destination of the read. The latter problem is due to x86 instructions being variable-length and so it is not possible in the general case to dynamically determine the previous instruction. Although we have not implemented this yet, we envision the following solution. We can disassemble the program and store a table that holds the address of each instruction. In this way, we can obtain the old program counter (PC) by indexing into the table's entry for the PC's current value, and then use the value in the previous entry. The one exception is if the instruction at the current PC is a jump target. The only way for a jump to read a memory location is in an indirect jump as a result of calling a function via a function pointer. Thus, if we are at the beginning of a function, we index into the table's entry for the return address of the current function, and then use the value in the previous entry as the old PC.

However, we still do not know the original value of the read's destination, and this prevents us from undoing the read's effects. Thus, we must prevent the system from using the value that was read until after the second *local access* (i.e., the second *local write*), when we can re-execute the read to read the proper value. To do this, we decode the previous instruction and check whether the destination of the read was a register or a memory location. If it was the former, then we freeze the *remote thread* which made the access because only it can use the value. If it was the latter, then we monitor the destination of the read and freeze the next thread to read it. This forms a dependency chain of threads, where each thread depends on the thread immediately before it to read the correct value. When the associated *end_atomic()* is called, we set the PC of the first thread in the chain to its old value, then single-step the read. This is repeated until all threads in the chain have re-executed their respective reads. The effect is to shift the reads after the *atomic region* in the order in which they occurred. If the situation arises that we need to monitor the destination of a read but no debug registers are available for use, we

unfreeze all threads in the dependency chain, and report that we were unable to prevent an atomicity violation.

A second problem alluded to before was that exact breakpoints are not always possible. The P6 family of processors, which include the Pentium I Pro, II and III processors, do not report data breakpoints for the REP MOVS and REP STOS instructions until after the completion of the iteration after the iteration in which the memory location was accessed. These two instructions perform repeated array reads and writes respectively. The consequences of such an event are serious because our system is dependent on being notified immediately of a *remote access*. There are two situations to consider. The first situation is that a *violating access* involving the REP MOVS/STOS instruction occurs, and the second *local access* occurs between when the *violating access* is made and the associated data breakpoint is triggered. In this case, our system would not be able to detect or prevent the atomicity violation. The second situation is that a *violating access* involving either the REP MOVS/STOS instruction occurs, and the second *local access* occurs after the associated data breakpoint is triggered. In this case, our system would still detect the atomicity violation, but our efforts to prevent the atomicity violation could introduce errors into the program. This is because prevention requires we undo the effects of the access such that they are not observed by the program. If there is a delay between the time of the access and the notification of the access, then reversing the effects could cause the program to enter an inconsistent state. Although we have not implemented it, we could choose not to prevent atomicity violations in the case that the current or previous instruction was REP MOVS/STOS and we were running on a machine that is using a processor from the P6 family. We would use the previously mentioned method for determining the previous instruction.

Additionally, all Pentium processors do not report data breakpoints for repeated INS and OUTS instructions until after the iteration in which the memory was accessed. These two are I/O instructions, and thus should not have any impact on our system as they would not appear in user-level programs.

4.5 Optimizations

As the next chapter shows, most of our overhead is due to the number of system calls made, and that the majority of our system call cost comes from dropping into the kernel. To reduce this

overhead, we target both aspects of the problem. First, the ANNOTATOR supports a whitelist for variables. This can either be done with a list of functions, where all function arguments are added to the whitelist, or with a list of object types, where all variables of those types are added to the whitelist. Currently, we manually generate a whitelist of synchronization object types, such as lock types, for the programs used in our evaluation. We assume that synchronization functions have been implemented correctly and therefore all atomicity violations with respect to synchronization variables are benign. This lowers the overhead of our system by reducing the number of *atomic regions*, and in turn, the number of system calls programs running on our system must make.

Second, instead of *end_atomic*() dropping into the kernel, it sets a flag and some variables with the relevant information (e.g., what registers to clear) in user space. Then, on the next *begin_atomic*() call or *remote access*, our system checks the flag and does what the *end_atomic*() would have done. Intuitively, we are lazily executing *end_atomic()*'s functionality by delaying it until we have to drop into the kernel for some other reason, thus avoiding a boundary crossing between user and kernel space If the functionality is delayed to a *remote access*, it is never considered a *violating access* because the *end_atomic()* has already occurred. This optimization also applies to *clear_ar_data()*.

Third, we employ minor optimizations to avoid dropping into the kernel when not required, such as when there are no available data debug address registers. These are straightforward and we will not elaborate on them further. Lastly, we implemented a simple but powerful optimization early in the development of our system. We disable the data debug registers during context switches when the next thread to run is the *local thread* that had set the registers. This eliminates the traps that would have otherwise occurred for each *atomic region* due to *local accesses*. However, the optimization introduces a problem for the AVT [*local write*, *remote write*, *local read*]. After the *local write*, OLD_VALUE still contains the value of the shared variable before that write. Therefore, when the *remote write* occurs, we will revert to the wrong value. Thus, the optimization requires that the ANNOTATOR split the *local write* into two statements. The first statement saves the value to be written for the PREVENTION ENGINE, and the second statement performs the original write.

5 Evaluation

We evaluate our system according to two metrics. We first measure run-time performance overhead because our system is an online one, and thus low overhead is an important criterion. Then, we test our system's ability to achieve its primary goal: detect and prevent atomicity violations. We do not measure the overhead introduced by the ANNOTATOR during the compilation process because that component of our system is used only once each time the program is released. It is not used any other time the program is compiled. As such, its overhead is negligible.

5.1 Experimental Set-up

All evaluation was conducted on a machine with a 2.13 GHz Core 2 Duo processor, 2GB of RAM, a 7200 RPM Serial-ATA disk and a Gigabit Ethernet network card. The operating system is Ubuntu 8.10, with a Linux 2.6.27 kernel modified to implement the PREVENTION ENGINE.

5.2 Run-time Overhead

To measure performance overhead, we ran several large multithreaded applications under our system. These include the NSS module in the Mozilla Firefox web browser [23], VLC media player [43], Apache web server [41], MySQL database application [26] and the SPEC2001 Open MP (OMP) benchmark [40]. Table 5.1 lists the benchmark used for each application. These were chosen to provide good code coverage, in order to get an accurate indication of our overheads. Our system was run in BUGFINDING mode to obtain the worst-case results.

We first ran the applications with a version of our system which was largely non-optimized – it contains only the last optimization in Section 4.5 – and which we will refer to as the non-optimized version. The overheads are presented in Table 5.2. The overheads for the Webstone [21] and TPC-W [42] benchmarks are with respect to throughput. The overheads for all other benchmarks are with respect to execution time. These initial results were unsatisfactory and we then considered two optimization strategies. The first strategy was to reduce the freeze period. To test the possible benefits, we ran our non-optimized system with all freeze periods set to zero. The second strategy was to reduce the cost of our system calls. To test the possible benefits, we ran our non-optimized system with *begin_atomic()* and *end_atomic()* replaced with

null (i.e., empty) system calls. Each modification was applied in isolation, and the overhead results are presented in Table 5.3. They show that the first strategy would not yield significant benefits. Although they would also seem to suggest the same thing about the second strategy, the reality is more nuanced. Table 5.3 actually demonstrates that the cost of *begin_atomic()/end_atomic()* is dominated by the boundary crossing between user and kernel space. Thus, the strategy with the greatest potential is to reduce the total number of system calls made by the instrumented programs.

Application	**Benchmark**
NSS	Running the included test suite
VLC media player	Transcoding a video using the x264 codec
Apache	Webstone 2.5
MySQL	TPC-W with 10 browsers
SPEC2001 OMP	Running itself

Table 5.1: Benchmarks used

We implemented optimizations which tried to first reduce the number of system calls made, and second, reduce the number of times we dropped into the kernel. These optimizations were discussed in Section 4.5. The results of these optimizations are presented in Table 5.4. In the last column, “Minor optimizations” refers to simple checks that ensure we drop into the kernel only when required. For example, as mentioned previously, these include checking for the existence of available data debug registers. The results show that the best improvement was obtained by not having *end_atomic()* drop into the kernel, while the use of a whitelist afforded approximately half the improvement. This is because the former reduces the overhead of every full *local access pair* (i.e., both *begin_atomic()* and *end_atomic()* execute at run-time) by half. The latter is only able to remove from consideration *local access pairs* involving synchronization variables, which are a small fraction of all *local access pairs* that we check.

Application	**Overhead (%)**
NSS	27.6
VLC media player	15.4
Apache	23.7
MySQL	24.5
SPEC2001 OMP	26.2

Table 5.2: Non-optimized overhead results

	Overhead (%)		
Application	**Original**	**Replace with null system call**	**Reduce freeze periods to zero**
NSS	27.6	25.0	26.4
VLC	15.4	14.1	14.7
Apache	23.7	21.9	22.5
MySQL	24.5	22.4	23.0
SPEC2001 OMP	26.2	24.3	25.2

Table 5.3: Impact of different sources of overhead

	Overhead (%)			
Application	**Original**	**Whitelist**	**Whitelist + *end_atomic()* does not drop into kernel**	**Whitelist + *end_atomic()* does not drop into kernel + Minor optimizations**
NSS	27.6	24.2	16.4	16.1
VLC	15.4	13.6	10.6	10.4
Apache	23.7	20.3	12.8	12.7
MySQL	24.5	21.4	13.9	13.7
SPEC2001 OMP	26.2	23.6	15.7	15.5

Table 5.4: Effects of various optimizations

5.3 Detecting and Preventing Atomicity Violations

In order to evaluate the ability of our system to achieve its primary goal of detecting and preventing atomicity violations, we tested it against a collection of known bugs. We created a corpus of concurrency bugs caused by atomicity violations by searching the public bug trackers of various large open-source multithreaded programs. We first collected fixed bugs whose reports contained the following keywords: “data race”, “race condition”, “concurrency” or “synchronization”. Afterwards, we removed those bugs that were not caused by atomicity violations. Finally, we filtered out bugs in non-C code and bugs whose reports either did not mention the triggering inputs or did not mention the specific code section and interleaving which caused the bug. Table 5.5 shows the distribution of the bugs by application and the type of interleaving that caused it.

For each bug, we ran the associated program with the inputs specified in the bug report. Unfortunately, not all bugs cause crashes, and we do not possess oracles which can determine whether the application is behaving incorrectly. Thus, for each bug, we obtain the *local access pair* from the bug report. Then, we manually analyze the source code and note the ID of the *atomic region* containing the aforementioned *local access pair*. We consider a bug to be detected if an atomicity violation involving the same *atomic region* occurs. Our system was able to detect all of the bugs.

For those bugs which caused crashes, we verified that once our system detected the bug, the application no longer crashed. For all other bugs, we lacked a means of determining whether the program was behaving correctly and thus whether we had successfully prevented the atomicity violation. Therefore, we augmented the PREVENTION ENGINE to provide timestamps, as well as the values read and written, for all accesses in the AVT of the relevant atomicity violation. We manually inspected the reported information and verified that our prevention routines were behaving as expected. Note that, due to the deficiency mentioned in Section 4.4, we were unable to prevent the NSS bug in the last row (i.e., the one caused by the *local write*, *remote read*, *local write* unserializable interleaving).

Unserializable Interleaving	Application			Total
	Apache	NSS	MySQL	
$R_{local}(A)$ $W_{remote}(A)$ $R_{local}(A)$	2	2	2	6
$R_{local}(A)$ $W_{remote}(A)$ $W_{local}(A)$	1	3	0	4
$W_{local}(A)$ $W_{remote}(A)$ $R_{local}(A)$	0	0	0	0
$W_{local}(A)$ $R_{remote}(A)$ $W_{local}(A)$	0	1	0	1

Table 5.5: Bug detection results

6 Future Work

There are a few directions we would like to explore with our work. The most straightforward one is supporting more languages. Currently, our system only works with programs written in C because the ANNOTATOR can only parse C. In order to support more languages, we would need a parser for each one. For interpreted languages (e.g., Java), we may also need to modify the PREVENTION ENGINE, depending on the threading model used.

The main source of overhead for our system is the number of system calls, which makes it a good target for future work. We can tackle it in two ways: 1) reduce the cost of our function calls, or 2) reduce the number of function calls. In order to reduce the cost, we could add dynamic checks to see if the *atomic region*'s shared variable is actually shared. If it is not, we avoid dropping into the kernel. In order to reduce the number of function calls, we could apply the static lockset algorithm to filter out places where we know atomicity violations cannot occur, or escape analysis to eliminate variables from the LSV that we know are not shared.

Finally, an area of our work which has largely been unexplored is the use of different scheduling algorithms to expose hidden atomicity violations during BUGFINDING mode. This could be applied to all threads prior to encountering an *atomic region*, or applied to *remote threads* when the *local thread* is frozen inside an *atomic region*.

7 Conclusion

Concurrency bugs are a difficult class of bugs to solve. Their numbers will only increase as more programs become multithreaded to take advantage of multi-core machines. We present a system which can detect, prevent and expose concurrency bugs caused by atomicity violations. It is an effective online system that imposes low overhead. We were able to achieve this for three reasons. The first is that the dynamic component of our system was implemented in the kernel. This allowed us to control the thread scheduling of user-level programs more efficiently than with dynamic binary instrumentation tools such as Pin. The second is that we used hardware registers present in modern Intel/AMD processors. This allowed us to efficiently monitor accesses to shared variables and be immediately notified when they occurred. Lastly, we implemented optimizations that greatly reduced the number of times we had to drop into the kernel. The end result is that users are protected from latent atomicity violation bugs in applications.

Bibliography

[1] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322-335, 2006. ACM.

[2] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 278-289, 2007. ACM.

[3] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258-269, 2002. ACM.

[4] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85-96, 2009. ACM.

[5] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211-224, 2002. ACM.

[6] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237-252, 2003. ACM.

[7] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256-267, 2004. ACM.

[8] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121-133, 2009. ACM.

[9] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293-303, 2008. ACM.

[10] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 338-349, 2003. ACM.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen, "DART: directed automated random testing," *SIGPLAN Not.*, vol. 40, pp. 213-223, 2005.

[12] Maurice Herlihy and J. Eliot B., "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289-300, 1993.

[13] Derek R. Hower and Mark D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," *SIGARCH Comput. Archit. News*, vol. 36, pp. 265-276, 2008.

[14] Intel Corporation. (2009, June) Intel® 64 and IA-32 Architectures Software Developer's Manual. [Online]. http://www.intel.com/Assets/PDF/manual/253669.pdf

[15] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558-565, 1978.

[16] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 277-288, 2008. IEEE Computer Society.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190-200, 2005. ACM.

[18] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *SIGARCH Comput. Archit. News*, vol. 36, pp. 329-339, 2008.

[19] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37-48, 2006. ACM.

[20] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346-358, 2006. ACM.

[21] Mindcraft. (2009) Mindcraft - WebStone Benchmark Information. [Online]. http://www.mindcraft.com/webstone/

[22] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 289-300, 2008. IEEE Computer Society.

[23] Mozilla Corporation. (2009) Network Security Services (NSS). [Online]. http://www.mozilla.org/projects/security/pki/nss/

[24] Madanlal Musuvathi and Shaz Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *SIGPLAN Not.*, vol. 42, pp. 446-455, 2007.

[25] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, pages 267-280, 2008.

[26] MySQL AB. (2009) MySQL : MySQL 5.1 GA. [Online]. http://dev.mysql.com/downloads/mysql/5.1.html

[27] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229-240, 2006. ACM.

[28] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213-228, 2002. Springer-Verlag.

[29] Robert O'Callahan and Jong-Deok Choi, "Hybrid dynamic data race detection," *SIGPLAN Not.*, vol. 38, pp. 167-178, 2003.

[30] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97-108, 2009. ACM.

[31] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25-36, 2009. ACM.

[32] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179-190, 2003. ACM.

[33] Milos Prvulovic and Josep Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110-121, 2003. ACM.

[34] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187-197, 2006. ACM.

[35] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83-94, 2005. ACM.

[36] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391-411, 1997.

[37] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11-21, 2008. ACM.

[38] Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. RSTM: A Relaxed Consistency Software Transactional Memory for Multicores. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 428, 2007. IEEE Computer Society.

[39] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 3-3, 2004. USENIX Association.

[40] Standard Performance Evaluation Corporation. (2009) SPEC OMP V3.2. [Online]. http://www.spec.org/omp2001/

[41] The Apache Software Foundation. (2009) The Apache HTTP Server Project. [Online]. http://httpd.apache.org/

[42] Transaction Processing Performance Council. (2009) TPC-W. [Online]. http://www.tpc.org/tpcw/default.asp

[43] VideoLAN. (2009) VLC media player - Open Source Multimedia Framework and Player. [Online]. http://www.videolan.org/vlc/

[44] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *ASE '00: Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 3, 2000. IEEE Computer Society.

[45] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252-263, 2009. ACM.

[46] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos, "Mechanisms for store-wait-free multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, pp. 266-277, 2007.

[47] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304-316, 2002. ACM.

[48] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-14, 2005. ACM.

[49] Jie Yu and Satish Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," *SIGARCH Comput. Archit. News*, vol. 37, pp. 325-336, 2009.

[50] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 221-234, 2005. ACM.