AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection[*]

Abdullah Muzahid, Norimasa Otsuki[†], and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

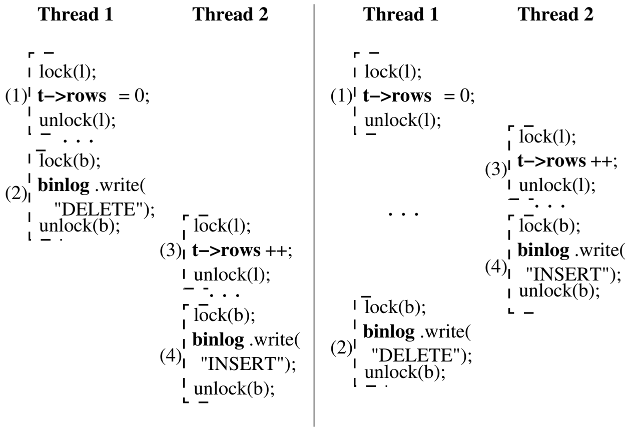
Abstract

A particularly insidious type of concurrency bug is atomicity violations. While there has been substantial work on automatic detection of atomicity violations, each existing technique has focused on a certain type of atomic region. To address this limitation, this paper presents AtomTracker, a comprehensive approach to atomic region inference and violation detection. AtomTracker is the *first* scheme to (1) automatically infer *generic* atomic regions (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of code construct the region is embedded in) and (2) automatically detect violations of them at runtime with negligible execution overhead. AtomTracker provides novel algorithms to infer generic atomic regions and to detect atomicity violations of them. Moreover, we present a hardware implementation of the violation detection algorithm that leverages cache coherence state transitions in a multiprocessor. In our evaluation, we take eight atomicity violation bugs from real-world codes like Apache, MySql, and Mozilla, and show that AtomTracker detects them all. In addition, AtomTracker automatically infers all of the atomic regions in a set of microbenchmarks accurately. Finally, we also show that the hardware implementation induces a negligible execution time overhead of 0.2–4.0% and, therefore, enables AtomTracker to find atomicity violations on-the-fly in production runs.

1. Introduction

Multicore machines have brought into broad light how difficult it is to debug concurrency bugs — bugs such as deadlocks, livelocks, data races or atomicity violations. Of these bugs, atomicity violations are particularly hard to isolate, and have received little attention compared to their importance [7]. An atomicity violation can occur when the programmer fails to enclose in the same critical section all of the memory accesses that should occur atomically. During execution, such accesses get interleaved with accesses from another thread that alter the program state, making it inconsistent.

Figure 1 shows an example of an atomicity violation bug in the MySql program. Variables *t->rows* and *binlog* need to be accessed together to generate the correct logging order of concurrent operations in the database (Figure 1(a)). However, the variables are protected by different critical sections. It is possible that, in between the accesses to the two variables by a thread, a second thread accesses them (Figure 1(b)). This is an atomicity violation, which results in a wrong logging order. Note that this program is incorrect even though there is no data race. In practice, atomicity violations are hard to debug.



```
     Thread 1                  Thread 2

     lock(l);
(1)  t->rows  = 0;
     unlock(l);
        . . .
     lock(b);
(2)  binlog .write(
       "DELETE");
     unlock(b);
                               lock(l);
                          (3)  t->rows ++;
                               unlock(l);
                                  . . .
                               lock(b);
                               binlog .write(
                          (4)    "INSERT");
                               unlock(b);
```

DELETE & INSERT recorded in correct order

(a)

```
     Thread 1                  Thread 2

     lock(l);
(1)  t->rows  = 0;
     unlock(l);
                               lock(l);
                          (3)  t->rows ++;
                               unlock(l);
                                  . . .
        . . .                  lock(b);
                               binlog .write(
                          (4)    "INSERT");
                               unlock(b);
     lock(b);
     binlog .write(
(2)    "DELETE");
     unlock(b);
```

DELETE & INSERT recorded in wrong order

(b)

Figure 1. Atomicity violation in MySql. The variables involved in the bug are in bold.

Existing approaches to find these bugs can be classified into two groups. The first one are techniques that require the programmer to annotate the Atomic Regions (AR) [3, 4, 5, 17]. Providing this information may be too tedious and error prone on the part of the programmer. The second group are techniques that attempt to detect atomicity violations automatically. They include, among others, SVD [18], AVIO [8], AtomFuzzer [15], and Atom-Aid [11]. These techniques are often effective. However, as we will discuss in detail, they are all constrained in the types of ARs that they can support — typically limited by the number of variables that they access, the number of instructions that they execute, or the type of code construct in which they are embedded (e.g., a function). For example, AVIO only finds ARs with two instructions and a single variable. A substantial improvement in the state of the art would be to devise an approach that identifies violations of *any* type of AR.

This paper proposes such an approach, which we call *AtomTracker*. AtomTracker is a comprehensive approach to AR inference and violation detection. It is the *first* scheme to (1) automatically infer *generic* non-nested ARs (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of code construct beyond avoiding AR nesting) and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed.

AtomTracker has two parts: one that automatically infers ARs (*AtomTracker-I*) and one that automatically detects violations of

[*] This work was supported in part by the National Science Foundation under grant CNS 07-20593, and by Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC).

[†] N. Otsuki is a visiting scholar from Renesas Electronics Corp, Japan.

their atomicity (*AtomTracker-D*). AtomTracker-I infers generic ARs by analyzing annotation-free memory traces of test runs of the program. AtomTracker-I's main contribution is a novel algorithm that works by greedily joining successive references of a thread into an AR if the other threads do not conflict. AtomTracker-I does not require any semantic knowledge of the program. It is the first algorithm of its kind.

AtomTracker-D takes the set of ARs and detects violations of their atomicity at runtime. AtomTracker-D's first contribution is an algorithm for atomicity violation detection. It checks if concurrently-executing ARs can be made to appear to execute in sequence by taking one reference at a time and reconsidering the relative order of the ARs. The second contribution is a *hardware implementation* of AtomTracker-D in a shared-memory multiprocessor that leverages cache coherence state transitions. It induces a negligible execution time overhead and, therefore, can be on during production runs.

We evaluate AtomTracker by using eight atomicity violation bugs from real-world applications such as Apache, MySql and Mozilla. The results show that AtomTracker correctly detects them all, which is not possible with any of the existing schemes. In addition, AtomTracker automatically infers all of the ARs in a set of microbenchmarks accurately. Finally, the hardware implementation induces an execution time overhead of 0.2–4.0% and, therefore, enables AtomTracker to find atomicity violations on-the-fly in production runs.

This paper is organized as follows: Section 2 gives a background; Sections 3 and 4 describe the AtomTracker-I and AtomTracker-D algorithms; Section 5 presents the hardware implementation; Section 6 shows our evaluation; Section 7 discusses our scheme's main limitation; and Section 8 concludes.

2. Background: Atomicity Violation Detection

The state of the art in atomicity violation detection without annotations is set by SVD [18], AVIO [8], MUVI [6], AtomFuzzer [15], PSet [19], Atom-Aid [11], and Bugaboo [9]. SVD [18] proposes the Computational Unit (CU) concept, which approximates a limited type of AR. The idea is that, after a thread has written to a shared variable, when it reads it again, it starts a new CU. As a program runs, SVD computes CUs based on the observed dependencies. SVD reports violations when CUs are interleaved with unserializable writes from other threads. While SVD is effective, it only looks for violations of the limited set of ARs considered.

AVIO [8] looks for ARs composed of only a single variable and two instructions. AVIO uses memory traces of correct executions of the program to train the algorithm. If two instructions in a thread that access the same shared variable are never found to be interleaved unserializably by an access from another thread while training, then AVIO assumes that these two instructions are intended to be atomic by the programmer. Consequently, AVIO reports violations when these instructions are interleaved unserializably in production runs.

MUVI [6] is a step toward handling multiple variables. It finds access correlations among multiple variables. Variables that are accessed together for some minimum number of times are likely to be related. These variables should be protected by the same lock. The MUVI paper shows how to use correlation information in a race detector. It claims that this information would be hard to use to detect multi-variable atomicity violations.

AtomFuzzer [15] looks for the case when a lock is grabbed multiple times in the same function. It reckons that this pattern suggests an atomicity violation bug. This is because functions are likely to be atomic.

PSet [19] detects and avoids concurrency bugs by embedding in the binary legal interleavings obtained from training runs. This is done by specifying which memory operation depends upon which other remote memory operations. If, at runtime, an unexpected interleaving is observed, the system reports a potential bug. Bugaboo [9] extends this work by adding context information to the interleavings. A given interleaving is acceptable under an certain program context, while it is not under a different one. These two works do not specifically target atomicity violations, but can be used to detect unusual interleavings that uncover such violations.

Atom-Aid [11] detects and survives atomicity violations in hardware. It uses a special multiprocessor architecture that executes blocks of instructions atomically. Therefore, ARs are determined by the blocking hardware, and are subject to a certain size restriction. Because cache overflow disables block atomicity, the scheme may not be able to ensure the detection of atomicity violations in a given set of ARs.

A recent work that can detect multi-variable atomicity violations is ColorSafe [10]. This scheme requires *annotations* of which groups of variables are related (i.e., have the same color). Then, the system reports a violation when a thread's accesses to same-color variables are interleaved by another thread's access to a variable of that color.

While the work in this area has made great strides, it is still limited, considering the many dimensions of the problem. Our scheme, AtomTracker, differs from these proposals in that it is the first one to work with annotation-free *arbitrary* (non-nested) ARs. Such ARs are not constrained in the number of variables, number of instructions, or type of code construct (e.g., a function). Moreover, AtomTracker both automatically *infers* ARs in a program (AtomTracker-I) and automatically *detects* atomicity violations of these ARs at runtime (AtomTracker-D). All this makes AtomTracker the first of its kind.

3. AtomTracker-I: Automatic Inference of Atomic Regions

AtomTracker-I automatically infers ARs from a program by training on the memory traces of many correct executions of the program. Training on correct runs of programs is a feasible and commonly-used approach in software-development groups. It is also used in many proposed debugging tools, such as AVIO [8] or PSet [19].

AtomTracker-I does not require any manual annotation from the programmer. The key idea is to scan the dynamic memory reference trace of a thread, and *greedily try to join* successive references of the thread into a common AR — for as long as the references of the other threads do not conflict with this newly-formed AR. The operation is repeated for the references of each thread in turn, and then for each training file. The output of AtomTracker-I is a list of static AR entry and exit points in the program.

With this algorithm, the analysis of the first trace file typically generates a set of large ARs. Later, as we process another trace file, we may find evidence that an AR should be broken into two or more smaller ARs. As we process more files, ARs will tend to get more numerous and smaller. When the set of ARs does not change

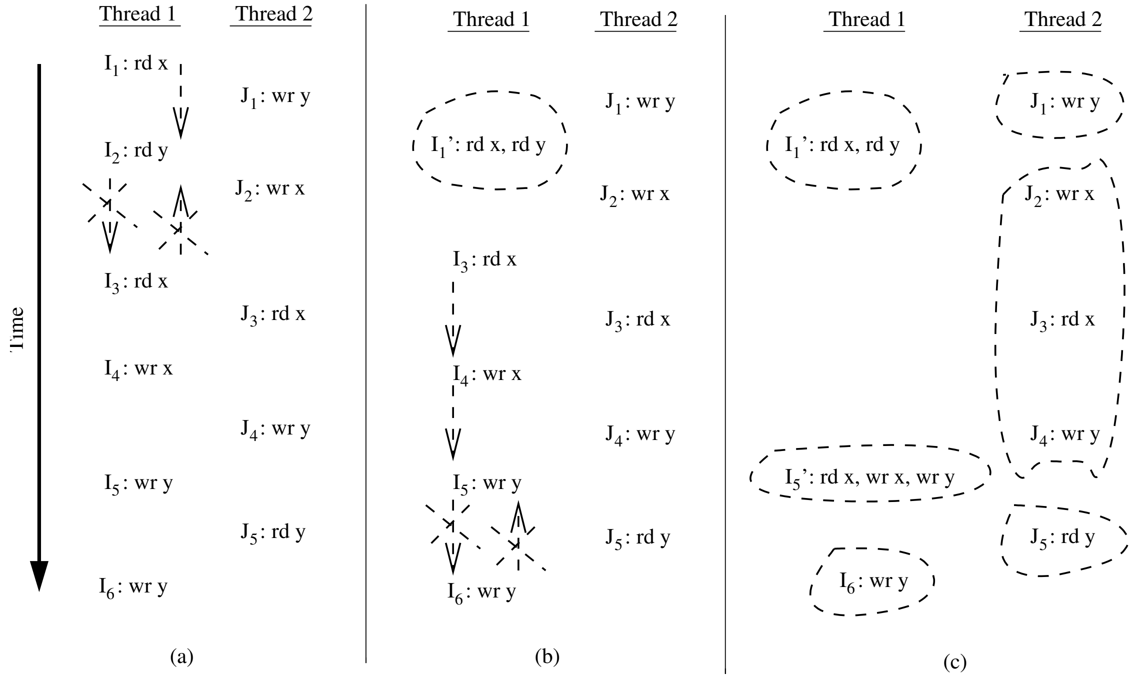


Figure 2. Basic AtomTracker-I algorithm for inferring ARs.

anymore, we assume that we have found the actual ARs. The rationale is that if a set of accesses by a thread are found to be atomic in all the correct runs, then the programmer likely intends them to be atomic. As we can see, this algorithm extracts any arbitrary AR, unconstrained in the number of variables, number of instructions, or type of code construct.

Next, we describe the basic algorithm, some design decisions, and two examples.

3.1. Basic AtomTracker-I Algorithm

AtomTracker-I processes multithreaded traces of a program's memory accesses. A trace file has an ordered list of records from several threads collected during an execution of the program. Each record contains the thread ID, PC, address accessed, and whether it is a load or a store. AtomTracker-I processes many trace files of correct runs.

The algorithm works by greedily trying to join successive references of a thread into an AR. The goal is to generate ARs that are as big as possible. To describe the algorithm, we use the two-threaded trace of Figure 2(a). In the figure, I_i:*rd x* means that the instruction at PC I_i reads address *x*.

The algorithm starts by processing all the references of one thread, then moving to a second thread, and so on. The order of thread processing may initially generate small variations, but the end result is the same.

In the example, we start with Thread 1. The thread reads *x* in I_1 and then *y* in I_2. We would like to group I_1 and I_2 in an AR. However, Thread 2 writes *y* in between (in J_1). We cannot assume that I_2 happened before J_1 (and, therefore, move I_2 above J_1) because, if we do, Thread 1 would read a wrong *y* value. However, we can assume that I_1 happened after J_1 without affecting the final execution outcome. Consequently, we move I_1 down after J_1 (hence, the arrow) and combine I_1 and I_2 into an AR called $I_{1'}$ that reads *x* and *y*. Conceptually we think of it as a giant instruction. Next, we consider combining $I_{1'}$ with I_3. We cannot move $I_{1'}$ below J_2 of Thread 2 because there is a conflict on variable *x*. We cannot move I_3 upward above J_2, either. Therefore, AR $I_{1'}$ cannot grow any bigger (Figure 2(b)).

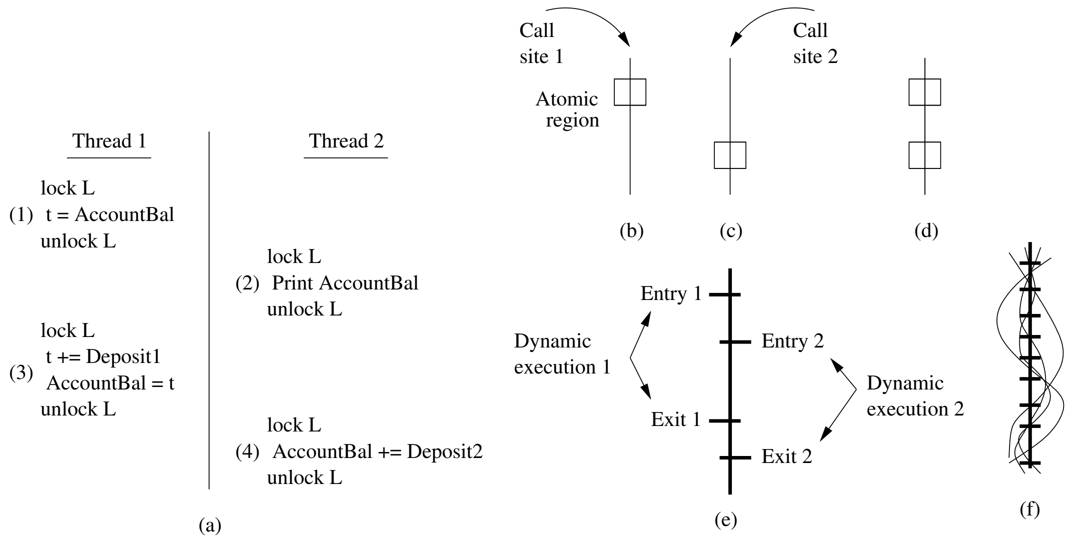
We start a new AR with I_3. Applying the same algorithm, we move I_3 down and combine it with I_4. Then, we move the $I_3 + I_4$ combination down, combine it with I_5 and get the bigger AR $I_3 + I_4 + I_5$. We call this AR $I_{5'}$ (Figure 2(c)). Now, we cannot move $I_{5'}$ down or move I_6 above J_5 because of the conflict with *y*. So, $I_{5'}$ does not grow any more.

If the trace contains more than two threads, every time we attempt to combine two accesses in Thread 1, we need to consider the intervening accesses from *all* the other threads. After processing Thread 1, we move to process Thread 2, then Thread 3, and so on. In the example, since there are only 2 threads, the ARs in Thread 2 are created as a side effect of processing Thread 1 (Figure 2(c)).

As AtomTracker-I processes a trace file, it may find that an AR that was previously inferred from the same file gets divided into multiple ARs. Therefore, after AtomTracker-I finishes the file, it goes back to re-process it from the beginning, and breaks the previous AR into the multiple smaller ones. This process continues iteratively until AtomTracker-I gets a stable set of ARs from this trace file.

After this, AtomTracker-I takes these ARs and moves to analyze the traces of another run. As AtomTracker-I processes the new file, it starts with the ARs obtained from previous files and likely breaks some of them into smaller ones. The process is repeated until the set of total ARs does not change by analyzing more runs. The fact that every iteration can only create smaller ARs ensures that AtomTracker-I always converges.

AtomTracker-I could also work even if some of the training runs were of incorrect executions. In this case, after the ARs are collected, we would apply statistical analysis to identify the good ARs. Specifically, if a certain AR appeared at least a threshold number of



```
        Thread 1                    Thread 2

     lock L
(1)  t = AccountBal
     unlock L
                              lock L
                          (2) Print AccountBal
                              unlock L
     lock L
     t += Deposit1
(3)  AccountBal = t
     unlock L
                              lock L
                          (4) AccountBal += Deposit2
                              unlock L
```

Figure 3. Design decisions in AtomTracker-I.

times in the test runs, we would consider it a good AR; otherwise, we would discard it.

3.2. Design Decisions

3.2.1. Handling Synchronization Variable Accesses

When AtomTracker-I attempts to move references up and down, it disregards synchronization accesses. This prevents synchronization bugs in the program from making AtomTracker-I infer incorrect ARs.

To see why, consider Figure 3(a). Thread 1 incorrectly uses two separate critical sections to add *Deposit1* to *AccountBal* — one to read the balance *(1)* and one to write it *(3)*. During training runs, a harmless critical section that prints the balance *(2)* interposes between the two. If we made the synchronization accesses visible to AtomTracker-I, it would detect conflicts on lock variable *L*, and incorrectly create separate ARs for *(1)* and *(3)*. Later, during production runs, if critical section *(4)* interposed between *(1)* and *(3)*, we would not detect an atomicity violation. However, by disregarding synchronization accesses, AtomTracker-I moves code *(1)* below *(2)*, and correctly merges *(1)* and *(3)* into the same AR.

3.2.2. Using Critical Section Information

Programmers mark as critical sections portions of the code that they want to be atomic. Typically, an atomicity violation occurs because the critical section that the programmer wrote is not large enough, although what is inside should indeed be atomic. Consequently, AtomTracker-I uses the lock/unlock statements in a program as a hint that they enclose code that should not be split into multiple ARs.

Specifically, the AtomTracker-I algorithm includes a preprocessing pass on the trace files. The pass identifies each lock/unlock pair that protects a critical section whose data is not being accessed concurrently by another thread. The sections protected by such lock/unlock pairs are recorded in a table. Later, when the AtomTracker-I algorithm runs, it uses this table. Specifically, it considers each of the sections in the table as an indivisible instruction. When it tries to expand an AR and finds one of these critical sections, it either takes-in the whole section or no instruction from it.

3.2.3. Using Loop Information

Typically, a programmer either makes a whole loop atomic or creates one or multiple ARs within an iteration of the loop. ARs very rarely cross an iteration boundary in a loop. To exploit this fact, in the same preprocessing pass described above, AtomTracker-I also identifies loops and checks whether they conflict with concurrent accesses from other threads. If none of the instances of a loop has conflicting accesses, then AtomTracker-I stores the loop boundaries in a database for later use. Later, when the AtomTracker-I algorithm runs, it will consider the whole loop as one giant indivisible instruction; when it tries to expand an AR, it either takes in the whole loop or no instruction from it.

On the other hand, if the loop has conflicting accesses in the trace, then the AtomTracker-I preprocessing pass records the iteration boundaries of the loop in the database for later use. Later, when the AtomTracker-I algorithm runs, it will always end an AR at the iteration boundary, so that no AR crosses the boundary. This, of course, does not disallow multiple ARs within an iteration.

To keep things simple, we flatten the inner loops and only consider outer loops. To make the preprocessing pass possible, loops are dynamically identified automatically using the algorithm of Moseley *et al* [13] during training runs, and this information is recorded in the trace files.

3.2.4. Handling Different Call Sites

Depending on what site a subroutine is called from, it may behave differently and, therefore, AtomTracker-I may create different ARs. For example, assume that when a subroutine is called from Site 1, AtomTracker-I creates the AR in Figure 3(b), while when it is called from Site 2, it creates the one in Figure 3(c). For simplicity, we make AtomTracker-I context insensitive. This means that, for each subroutine, we use the combination of all the ARs found in all of the calls. In the example, we use the ARs in Figure 3(d).

3.2.5. Atomic Region Representation

The goal of AtomTracker-I is to augment the *static* code of the program with AR entries and exits. This is not trivial because each training run may take different control paths.

For example, Figure 3(e) shows a segment of static code and two dynamic executions that took different control paths and found different AR entries and exits. Pictorially, Figure 3(f) shows many dynamic executions (shown as curved lines) that intercept the static code differently, inserting different AR entries and exits.

To ensure that the resulting information makes sense to AtomTracker-D, the AR exit markers inserted by AtomTracker-I in the code also include the PC of the corresponding AR entry point. In this way, if dynamic execution *2* in Figure 3(e) finds Exit *1*, it ignores it because it is not paired with Entry *2*.

3.3. Putting It all Together

To summarize, this is how the complete AtomTracker-I algorithm works. When first presented with a trace file, AtomTracker-I performs a preprocessing pass to record lock/unlock pairs as per Section 3.2.2 and loop boundaries and iteration boundaries as per Section 3.2.3. Then, AtomTracker-I runs the algorithm of Section 3.1, as many times as it needs to converge (typically 2–4 times).

The reason why the algorithm may need to run multiple times over the same trace is because key information for AR inference may only appear toward the end of the trace. This is seen in the trace of Figure 4. Suppose that the correct ARs for Thread 1 are those in Figure 4(a). However, as AtomTracker-I eagerly builds the first AR, such AR will gobble-up the I_3 read to *x* — since Thread 2 only reads *x*. AtomTracker-I will not include the I_4 write to *y* in the same AR because J_2 from Thread 2 conflicts. Consequently, AtomTracker-I will record that I_3 is the end of the AR starting at I_1 (Figure 4(b)).

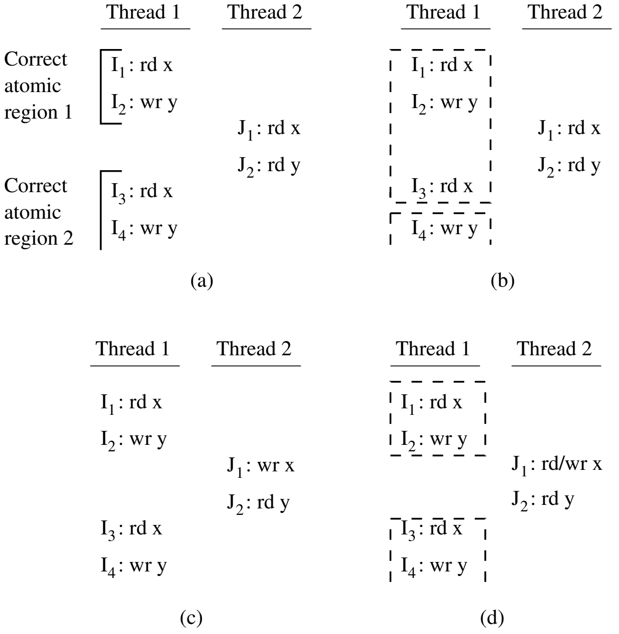


Figure 4. ARs correct themselves thanks to the multiple iterations of AtomTracker-I.

As AtomTracker-I runs, it can fix itself. Indeed, if later-on in the trace file, Thread 2 writes to *x* in between the two correct ARs as in Figure 4(c), AtomTracker-I will record that I_2 is the end of the AR starting at I_1. Then, in a second iteration of AtomTracker-I on the trace file (as per Section 3.1), as AtomTracker-I reaches I_1 in Figure 4(a), it will pick up the two ARs from its database — the smaller one that terminates at I_2 and the larger one that finishes at I_3. Since AtomTracker-I reaches I_2 next, it will confirm the smaller AR and discard the larger AR. The result will be the ARs of Figure 4(d), which are correct. This is an example of how multiple iterations help AtomTracker-I converge to the correct ARs.

After this, AtomTracker-I moves to process another trace file. The same convergence described above may occur across trace files. When the processing of several new trace files does not result in changes in the inferred ARs, AtomTracker-I completes.

3.4. Examples

In this section, we show the outcome of running the complete AtomTracker-I on two applications: Apache and LU-contiguous from SPLASH-2. To generate the ARs, we follow the training experiments outlined in Section 6.1. We need about 25 and 10 runs of Apache and LU-contiguous, respectively. For Apache, we focus on bug Apache#2 from our evaluation. Figures 5(a) and (b) show a dynamic execution of worker threads from the applications, showing where the ARs terminate. In the figure, dashed boxes are the ARs, which are labeled by their first instruction (shown as file name and line number). The figure also shows where the critical sections (CS) of the code are.

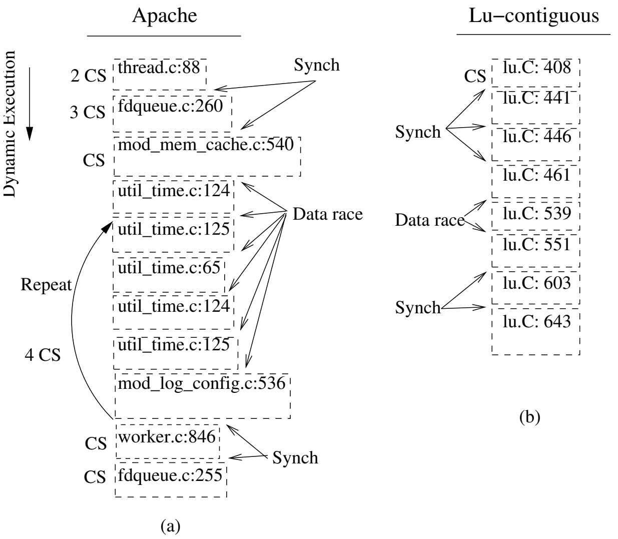


Figure 5. ARs found in Apache (a) and LU-contiguous (b).

The figure shows that the boundaries of ARs automatically converge to synchronizations and data races. Recall from Section 3.2.2 that AtomTracker-I considers a critical section without data races as a single instruction. In Apache, the synchronizations are critical sections. In LU-contiguous, all synchronizations but one are barriers. Conflicts between references before and after barriers cause AR boundaries to converge at barriers. Also, both programs have data races, which create AR boundaries. Finally, ARs are not equivalent to CSs: some ARs in Apache contain multiple CSs.

4. AtomTracker-D: Automatic Detection of Atomicity Violations

AtomTracker-D takes program annotations in the binary like those inserted by AtomTracker-I and automatically detects violations of ARs at *runtime*. Beyond this, AtomTracker-D is independent of AtomTracker-I.

The idea behind AtomTracker-D is as follows. As two ARs execute concurrently, AtomTracker-D checks whether they *can be made to appear to execute in sequence*, one after the other, in any

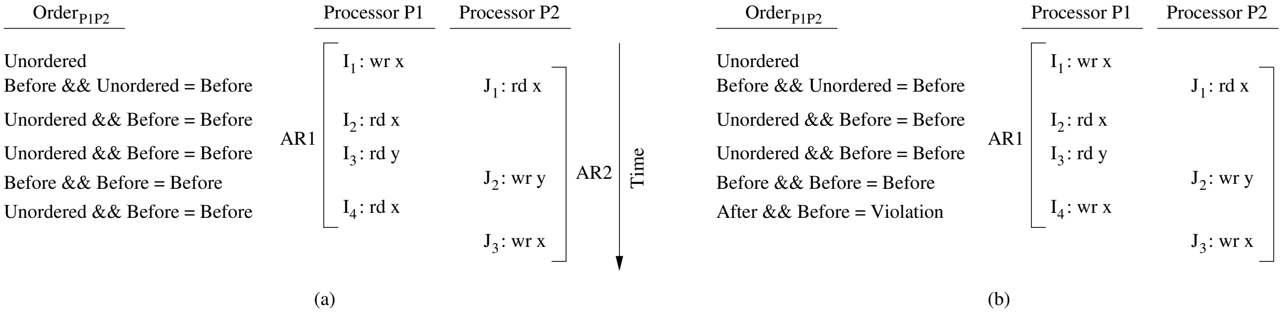


Figure 6. Two examples of the AtomTracker-D algorithm.

of the two possible orders. Only if they cannot, AtomTracker-D declares an atomicity violation.

To do its checks, AtomTracker-D uses a novel algorithm. The algorithm considers each access of the two (or more) concurrent ARs in sequential order and checks for conflicts. The algorithm can be efficiently implemented in hardware by leveraging the cache coherence protocol messages. In this section, we describe the algorithm as it could be implemented in a tool like Pin [12], while in Section 5, we describe a hardware implementation.

4.1. Description of the Algorithm

In this description, we assume that we have two processors executing two ARs concurrently; the algorithm will be later generalized to any number of concurrent ARs. Let us call the processors P_R and P_S, and the ARs AR_R and AR_S, respectively. In the AtomTracker-D algorithm, P_R keeps a local flag called *Order*$_{RS}$ that tells, at any time, whether AR_R appears (so far) to execute *before* or *after* AR_S. P_S keeps a symmetrical flag called Order$_{SR}$.

Let us focus on Order$_{RS}$. Order$_{RS}$ is updated in three cases, as shown in Figure 7. Case $P_R \rightarrow P_S$ is when a reference by P_S accesses a variable that has already been referenced by P_R in AR_R. In this case, we check the following references *to the variable*: (1) the current one by P_S and (2) all of the previous ones by P_R in AR_R. If at least one is a write, then P_S is dependent on P_R and, therefore, AtomTracker-D tries to set Order$_{RS}$ to *Before*. If, instead, all of them are reads, then P_S is not dependent on P_R and AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

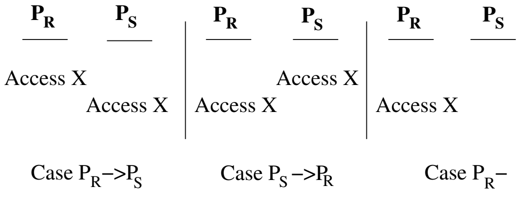


Figure 7. Different cases for AtomTracker-D.

The second case (Case $P_S \rightarrow P_R$) is the dual case: a reference by P_R accesses a variable that has already been referenced by P_S in AR_S. As before, we check if at least one of the relevant references to the variable is a write. If so, P_R is dependent on P_S and AtomTracker-D tries to set Order$_{RS}$ to *After*. Otherwise (i.e., the references to the variable are all reads), P_R is not dependent on P_S and AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

The final case (Case P_R-) is when P_R accesses a variable that P_S has not accessed in AR_S yet. In this case, AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

We say that the algorithm *tries* to set Order$_{RS}$ to a value because what it really does is to set Order$_{RS}$ to the logical AND of the value and the old contents of Order$_{RS}$. This is done to detect any ordering inconsistency: the AND of *Unordered* with any other value is that other value, while the AND of *Before* and *After* signals an ordering inconsistency. In this latter case, the two ARs cannot be serialized, and we have an atomicity violation.

4.2. Illustrative Examples

Figure 6 shows two examples where processors P1 and P2 execute atomic regions AR1 and AR2, respectively. It also shows the updates to Order$_{P1P2}$. We start with Figure 6(a). The first access (I_1) in the figure is a write to *x* by P1. This access falls into Case *P1-* because P2 has not accessed *x* in AR2 yet. Therefore, Order$_{P1P2}$ is set to *Unordered*. In access J_1, P2 reads *x*. This is Case $P_1 \rightarrow P_2$ with a write, and AtomTracker-D logically-ANDs *Before* to Order$_{P1P2}$. The result is *Before*. In access I_2, P1 reads *x*, which is Case $P_2 \rightarrow P_1$ with all reads, and AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. The result is *Before*. In access I_3, P1 reads *y*, which is Case *P1-*, and AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. The result is *Before*. Next, in access J_2, P2 writes *y*, which is Case $P_1 \rightarrow P_2$ with a write, and AtomTracker-D logically-ANDs *Before* to Order$_{P1P2}$. The result is *Before*. Next, in access I_4, P1 reads *x*, which is Case $P_2 \rightarrow P_1$ with all reads. AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. This was the last access in AR1 and the algorithm concludes that there is no atomicity violation because AR1 can appear to execute *before* AR2.

Consider now Figure 6(b), which changes I_4 from a read to a write. Access I_4 is Case $P_2 \rightarrow P_1$ with a write. Consequently, AtomTracker-D logically-ANDs *After* to Order$_{P1P2}$, which triggers a violation. Effectively, this access requires AR1 to be *after* AR2, which is incompatible with the other accesses.

In these examples, as soon as the first processor completes its AR, the algorithm can declare the presence or absence of a violation. However, this is not always the case. Specifically, if the Order flag of the processor that finishes first has the value *After*, AtomTracker-D cannot declare the outcome until the other processor also finishes its AR.

This case is shown in Figure 8, which shows both Order$_{P1P2}$ and Order$_{P2P1}$. The example shows a dependence from P1 to P2, and one in the opposite direction. Consequently, there is an atomicity violation. We focus first on P1 and its flag Order$_{P1P2}$. Reference I_1 sets the flag to *Unordered*, and J_1 ANDs *Unordered* to it. Reference I_2 is a read to *y* by P1, which is Case $P_2 \rightarrow P_1$ with a write. Consequently, AtomTracker-D logically-ANDs *After* to Order$_{P1P2}$. At this point, AR1 completes. However, AtomTracker-D cannot strictly declare an outcome because the AR that appears to execute first (AR2) is not complete yet.

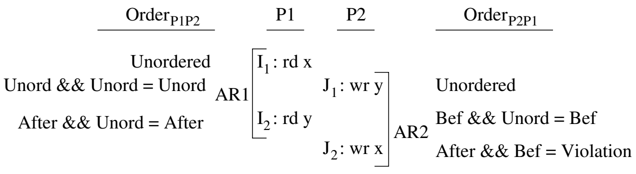


Figure 8. Both processors need to complete their ARs for AtomTracker-D to declare the outcome.

We now examine P2 and its flag Order$_{P2P1}$. Reference J_1 sets Order$_{P2P1}$ to *Unordered*. Reference I_2 ANDs *Before* to Order$_{P2P1}$. Finally, reference J_2 ANDs *After* to Order$_{P2P1}$. At this point, AtomTracker-D correctly declares a violation. Note, however, that the information that P1 had read *x* in AR1 *has to be kept around* until AR2 completes.

4.3. Generalization to More Atomic Regions

In the most general case, all *N* processors in a multicore may be executing atomic regions AR_0, $AR_1 ... AR_{N-1}$ concurrently, and AtomTracker-D has to detect atomicity violations between any two ARs. Consequently, each processor *i* keeps N-1 Order$_{i*}$ flags, where * takes the values from *0* to *N-1* except *i*.

Given a processor P_i, its Order$_{i*}$ flags are updated following the above description. Specifically, when another processor P_j accesses a variable that has been accessed by P_i, Order$_{ij}$ is ANDed with *Before* or *Unordered*. When P_i accesses a variable that has been accessed by P_j, Order$_{ij}$ is ANDed with *After* or *Unordered*. Finally, when P_i accesses a variable that P_j has not accessed yet, Order$_{ij}$ is ANDed with *Unordered*.

5. Hardware Implementation

Both AtomTracker-I and AtomTracker-D can be easily implemented in software. However, if we want to run AtomTracker-D on-the-fly in production runs, a software implementation is too slow. Consequently, we propose an efficient hardware implementation of AtomTracker-D.

A key insight is that the AtomTracker-D algorithm does not really need to observe every single access in the two (or more) concurrent ARs. Instead, it only needs to observe those accesses that induce cache coherence transactions in the network — with some exceptions that we will handle. Consequently, we propose to add a hardware module called *Atomicity Violation Detection Module* (AVM) attached to the on-chip network of the multicore. The AVM sees all the relevant accesses and runs the AtomTracker-D algorithm in hardware. It supports atomicity violation detection without slowing down execution.

For simplicity, our AVM design is centralized. It is possible to distribute the design to make it scalable.

5.1. Leveraging Cache Coherence Transactions

Many of the references processed in the AtomTracker-D algorithm of Section 4.1 are guaranteed not to change the value of the Order flag. For example, when a processor writes the same variable multiple times without any intervening access from other processors, then, after the first write, the value of the Order flag will not change. Consequently, it suffices that we capture only the references that *can* change the value of Order.

One group of accesses that can change Order are those that introduce RAW, WAW, or WAR dependences on a variable between two processors. These are cases $P_R \rightarrow P_S$ and $P_S \rightarrow P_R$ with a write in Figure 7. In particular, in a sequence of such dependences on a given variable between a given source and a given destination processor, AtomTracker-D only needs to observe one dependence. Fortunately, by construction (and ignoring false sharing for now) the first one of these dependences induces a change in the variable's cache coherence state, and a resulting coherence transaction in the network. Consequently, it can be observed easily.

The other group of accesses that can change Order are those that introduce RAR dependences on a variable between two processors (cases $P_R \rightarrow P_S$ and $P_S \rightarrow P_R$ with only reads in Figure 7) or accesses to variables that have not been accessed in the other AR (case P_R- in Figure 7). In a sequence of such accesses to a given variable for a given source and destination processor, AtomTracker-D only needs to observe one. Again, we choose the first one. This access may miss in the cache. If so, the AVM attached to the on-chip network will see the address and process the reference. However, this access will not miss if the variable is in the cache is a certain state when the processor enters the AR. Specifically, a read that finds the variable in a state equivalent to Shared or Dirty in the cache, or a write that finds it Dirty, will not miss.

To ensure that these accesses are visible to the AVM, we modify the cache to operate slightly differently when executing an AR. We add two *FirstAccess* bits per cache line — one for reads and one for writes. Every reference in the AR sets the corresponding *FirstAccess* bit in the line touched. If the reference hits in the cache and the corresponding bit was not set, the line address (and whether the access was a read or a write) is sent to the network, so that the AVM captures it; if the reference misses in the cache, the AVM sees it by default. *FirstAccess* bits are cleared when an AR finishes.

Finally, caches naturally replace lines and references that should otherwise not miss may miss. This means that the AVM will see more accesses than the strict minimum necessary for running AtomTracker-D. This does not affect correctness in any way.

5.2. An Atomicity Violation Detection Module (AVM) Based on Address Signatures

A naive AVM design would have, for each processor, a buffer with the list of references since the current AR started. The references would be processed using the AtomTracker-D algorithm. However, we propose a more efficient approach based on hardware address signatures. These are registers of about 2048 bits that accumulate the result of hashing addresses of references using Bloom filters [1]. Conceptually, a signature acts like a compressed buffer to store memory references. Per processor, the AVM has one signature (*RSig*) that hash-accumulates the addresses read in the AR and one (*WSig*) for the addresses written. Every network transaction by a processor executing an AR is captured by the AVM and the address is encoded in the correct signature. The AVM is shown in Figure 9. Each pair of signatures has N-1 associated Order flags.

AtomTracker-D is implemented as follows. Assume that the AVM observes a transaction by processor P_i to address *Addr*. The address is hashed and accumulated into the appropriate signature (RSig$_i$ or WSig$_i$). Assume it is a write. In this case, the AVM hardware uses membership signature operations [2] to see if *Addr* is present in RSig$_n$ or WSig$_n$, for all processors $P_n \neq P_i$. For each processor P_j where the answer is "yes", we have found a WAR or WAW. The AVM hardware does the following. Since this is Case $P_j \rightarrow P_i$ with write access, the AVM ANDs Order$_{ij}$ with *After* and, in addition, ANDs Order$_{ji}$ with *Before*. Finally, for each pro-

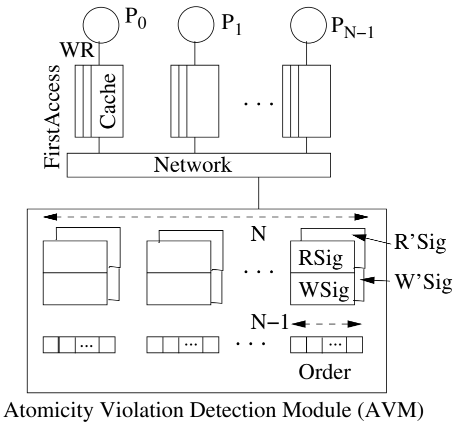


Figure 9. Signature-based hardware implementation of AtomTracker-D in a multicore.

cessor P_k where the answer is "no", since this is Case P_i-, the AVM ANDs Order$_{ik}$ with *Unordered*.

If the access is a read, the procedure is similar. AtomTracker-D first checks if *Addr* is present in WSig$_n$. For each processor P_j where the answer is "yes", we have found a RAW, and the AVM hardware ANDs Order$_{ij}$ with *After* and Order$_{ji}$ with *Before*. For the other processors, AtomTracker-D checks if *Addr* is present in RSig$_n$. For each processor P_j where the answer is "yes", we have found Case $P_j \rightarrow P_i$ with RAR only, and the hardware ANDs Order$_{ij}$ and Order$_{ji}$ with *Unordered*. Finally, for the other processors P_k, this is Case P_i-, and the hardware ANDs Order$_{ik}$ with *Unordered*. These operations are done in parallel.

Signatures do not keep precise information and, as they contain more addresses, they appear to include a larger number of other addresses as well — also called aliases. This may cause false positives, an issue we address in Section 5.4. To minimize such events, the AVM keeps several (four in our design) physical signatures for each logical signature. Each hashed incoming address is accumulated into one of the four signatures of the logical one, *depending on its address*. The number of operations on signatures does not change. Overall, at the cost of more hardware, this design reduces address aliasing.

As soon as the AVM detects a violation, it records it. When a processor ends its AR, its signatures are cleared. However, recall from Figure 8 that if the Order flag is set to *After* when a processor finishes its AR, and the concurrent AR is not yet finished, we cannot discard the state until the concurrent AR ends. Consequently, in this case, the AVM saves the processor's *RSig* and *WSig* into Shadow Signatures (*R'Sig* and *W'Sig* in Figure 9). These shadow signatures are checked by references from the concurrent AR until the latter finishes.

5.3. Software Interface

The AVM is driven by two instructions that are inserted in the program by either AtomTracker-I or another software tool. They are *atomic_enter* and *atomic_exit* (Table 1). *Atomic_enter* marks the entry to an AR. It triggers the allocation of signatures and Order flags in the AVM. It saves the PC of *atomic_enter* in a processor register that we call *AtomTrackerEntry*. This register will be used to identify the matching *atomic_exit* instruction. In addition, *atomic_enter* sets the cache operation to AR mode. In this mode, cache accesses set the *FirstAccess* bits of the lines touched. If the cache intercepts the access and the corresponding *FirstAccess* bit is clear, the hardware sends the line address (plus whether this is a read or a write) to the network, so that the AVM captures it. As indicated in Section 3.2.5, AR entries and exits may be unpaired and, therefore, execution may already be in an AR. In this case, *atomic_enter* has no effect.

Instruction	Description
atomic_enter	If (found outside an atomic region) Allocate signatures/flags in AVM *AtomTrackerEntry* = PC Set cache to AR mode. Per mem. access: If [(cache intercepts access) and (*FirstAccess* bit = 0)] Send line address + R/W to network *FirstAccess* bit = 1
atomic_exit PC	If [(PC = *AtomTrackerEntry*) or (NumMismatches = MAXMISMATCH)] Set cache to plain mode: *FirstAccess* bits = 0 for all cache lines Disable *FirstAccess* bit use *AtomTrackerEntry* = 0 Deallocate own structures in AVM

Table 1. Instructions added to manage the AVM.

Atomic_exit marks the exit of an AR. It takes the PC of the matching *atomic_enter* instruction. This instruction only has an effect if its PC argument is equal to the *AtomTrackerEntry* contents, or if we found a threshold number of mismatching *atomic_exit* instructions in a row. This is done to ensure that the current AR eventually finishes. In these cases, *atomic_exit* exits the AR by setting the cache operation to plain mode (no *FirstAccess* use), clearing *AtomTrackerEntry*, and deallocating its AVM structures.

5.4. Design Issues

Address aliasing in the signatures could result in false positives (FPs), namely claims of atomicity violations when there are none. The actual signature implementation affects the number of FPs. Consequently, we have chosen the design described in [14], which has few FPs. On the other hand, signatures cannot lead to false negatives, namely false claims of no atomicity violations.

Network accesses only expose *line* addresses. Therefore, all AtomTracker-D operations are done at cache-line granularity. This makes the implementation subject to false sharing, which can also result in false positive claims, but not false negatives.

The *atomic_enter* and *atomic_exit* instructions use fences, as if they were synchronizations. Therefore, the AtomTracker-D hardware also works with relaxed memory consistency model machines.

6. Evaluation

In this section, we evaluate AtomTracker-I and AtomTracker-D. We implement the AtomTracker-I algorithm in C++, and run it on traces of parallel applications generated by a Pin [12] tool. After AtomTracker-I determines the ARs, we use AtomTracker-D to find violations. We evaluate two implementions of AtomTracker-D, namely a software one using Pin and a hardware one using a whole-system simulator of the multicore architecture of Section 5 based on Simics [16]. The parameters of the multicore architecture simulated are shown in Table 3.

We use three representative applications (Apache, MySql, and Mozilla) and focus on eight documented atomicity violation bugs. These bugs have been used in the evaluation of past works like

Bug	Version	Files Involved	# Variables	Description
Apache#1	2.0.48	mod_log_config.c	Single	Unprotected read and write of buffer length can corrupt log file.
Apache#2	2.0.46	mod_mem_cache.c	Single	Unprotected read and write of reference counter can cause null pointer dereference.
Mozilla#1	0.8	jsstr.h, jsstr.c	Multiple	Non-atomic update of total number of strings and total string length permit them to be inconsistent.
Mozilla#2	0.8	jsinterp.h, jsinterp.c	Multiple	Non-atomic access of cache structure can cause cache and empty flag to be inconsistent.
Mozilla#3	0.9	jsdhash.c	Multiple	Concurrent access of *entryCount* and *removedCount* can cause the table to incorrectly shrink.
MySql#1	4.0.12	log.cc, sql_insert.cc	Single	Unprotected close and open of database bin log can cause some actions not to be logged.
MySql#2	3.23.56	sql_insert.cc, sql_delete.cc	Multiple	Non-atomic update of rows and bin log can cause wrong order of logging. Shown in Figure 1
MySql#3	4.0.16	slave.cc	Multiple	Non-atomic read of *log_file_name* and *log_file* can cause slave sql thread to fail.

Table 2. Bug descriptions.

Multicore	4 cores at 4 GHz
Core	4 issue out-of-order
L1 cache (private)	32 KB, 4 way, 2 cycle lat.
L2 cache (private)	512 KB, 8 way, 12 cycle lat.
Cache line	64B
Memory	80 cycle round trip lat.
Network	Bus
Bus bandwidth	128B/cycle
Coherence protocol	MESI
Signatures	2Kbit each like in [14]

Table 3. Multicore architecture evaluated.

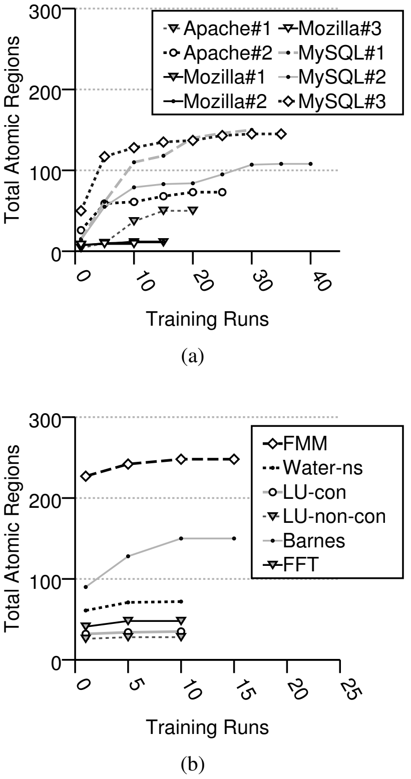
AVIO [8], MUVI [6], and PSet [19]. They are described in Table 2. Of these bugs, the three Mozilla bugs, MySql#2, and MySql#3 are multi-variable atomicity bugs; the rest are single-variable atomicity bugs. We also use six SPLASH-2 codes to characterize AtomTracker: three kernels (FFT, LU-con, and LU-non-con) and three applications (Barnes, FMM and Water-ns). Lastly, we also use three synthetic microbenchmarks to evaluate AtomTracker.

Our evaluation aims to (i) demonstrate the training methodology, (ii) show the bug detection ability, (iii) characterize the false positives, (iv) quantify the execution overhead, and (v) show the completeness of our design.

6.1. Training Sensitivity

To make AtomTracker effective, we need enough training runs to obtain a good set of ARs. To obtain many training runs, we change the program inputs and also get different interleavings for the same input. For MySql, we change the number of concurrent requests to the server. For Apache#2, we use *httperf* to send different numbers of concurrent requests, while for Apache#1, we use different numbers of calls to *wget* to fetch different numbers of web pages concurrently. For Mozilla, we wrote a driver that calls the buggy library with different parameters and different numbers of iterations. In all tests, we check that we do not trigger the bugs. This is easily done because all the bugs manifest themselves with either a program failure or a wrong output. Fortunately, bugs are hard to exercise because they require special interleavings. For all the applications, we stop the training as soon as we get 5 consecutive training runs that generate no new ARs.

Figure 10 shows the convergence of ARs in the commercial (a) and the SPLASH-2 (b) codes as we execute training runs. We perform from 10 to 40 training runs. The SPLASH-2 codes converge faster than the commercial codes because of their smaller size. Table 4 shows the average size of the resulting ARs, in number of source code lines that access shared variables (the lines that the programmer will check) and in number of shared variables. On average, ARs in the commercial codes have 52 lines and access 114 shared variables, whereas ARs in the SPLASH-2 codes have 5 lines and access 122 variables.



(a)

(b)

Figure 10. Convergence of ARs for the commercial (a) and the SPLASH-2 (b) codes as we execute training runs.

6.2. Bug Detection Ability

After AtomTracker-I infers the set of ARs, we provide the bug-triggering input and run AtomTracker-D. Our AtomTracker-D algorithm detects the resulting atomicity violation in every case. Table 5 compares the effectiveness of AtomTracker to that of AVIO [8], MUVI [6], and PSet [19] — based on data on the same bugs reported in the papers of those schemes.

App	# Lines of Code	# Shared Variables
Apache#1	44.3	62.4
Apache#2	76.4	125.6
Mozilla#1	62.2	82.9
Mozilla#2	83.8	197.1
Mozilla#3	46.4	102.1
MySql#1	43.9	230.4
MySql#2	44.3	82.4
MySql#3	17.0	31.8
Avg	52.3	114.3
SPLASH-2 kernels	4.6	215.5
SPLASH-2 apps	4.7	28.2
Avg	4.7	121.8

Table 4. Average AR size.

Bug	Atomicity Violation Detected?			
	AtomTracker	AVIO	MUVI	PSet
Apache#1	Yes	Yes	No	Yes
Apache#2	Yes	Yes	No	Yes
Mozilla#1	Yes	No	Yes	No
Mozilla#2	Yes	No	Yes	No
Mozilla#3	Yes	No	Yes	No
MySql#1	Yes	Yes	No	Yes
MySql#2	**Yes**	**No**	**No**	**No**
MySql#3	Yes	No	Yes	No

Table 5. Comparison of bug detection ability. Recall that the *Mozilla* bugs, *MySql#2*, and *MySql#3* are multi-variable atomicity violations.

AVIO and PSet are reported to catch the three single-variable atomicity bugs. However, since, by construction, they cannot catch multi-variable bugs, they cannot catch the three *Mozilla* bugs, *MySql#2*, or *MySql#3*. MUVI focuses on catching multi-variable data races. Consequently, it does not handle the bugs with a single variable, namely *Apache #1*, *Apache #2*, and *MySql #1*. The other five bugs are both multi-variable data races and multi-variable atomicity violations. The MUVI paper reports that MUVI catches the *Mozilla* bugs and *MySql#3*, but not *MySql#2*. The reason why *MySql #2* (shown in Figure 1) is undetected by MUVI but is detected by AtomTracker is as follows. MUVI fails because the correlation between *t->rows* and *binlog* is conditional: *t->rows* and *binlog* do not always need to be accessed together; only when *t->rows* is modified at the end, *binlog* needs to be modified atomically with it. This atomicity relation is easily extracted by AtomTracker-I by examining execution traces. MUVI does not extract this relation.

6.3. False Positives

AtomTracker is a heuristic-based approach and, therefore, subject to False Positives (FPs). To evaluate this issue, we apply AtomTracker-D to five bug-free runs per program — obtained by changing the inputs. Table 6 shows the average number of FPs observed per run, for three different scenarios: (i) software implementation, (ii) hardware implementation where, rather than signatures, we use an unbounded buffer to store addresses in the AVM, and (iii) hardware implementation with signatures as in Section 5.

The software implementation of AtomTracker-D has an average of only 0.8 and 1.6 FPs per run for the commercial and SPLASH-2 codes, respectively. These few FPs are due to undertraining. The more we train, the better the AR accuracy will be, and hence the fewer the FPs will be. The hardware implementation has two additional sources of FPs: false sharing due to using cache line addresses and aliasing due to signatures. Column 3 of Table 6 includes only the impact of false sharing, since signatures are replaced by an unbounded address buffer. This leads to an average of 6.8 and 13.7 FPs per run for the commercial and SPLASH-2 codes, respectively. When we use signatures and, therefore, include the aliasing effect as well, the average FPs per run is 8.4 and 16.4.

App	FP in SW impl	FP in HW impl	
		Unbounded buffer	Signatures
Apache#1	1.8	2.0	2.0
Apache#2	2.6	10.4	14.4
Mozilla#1	0.4	1.8	1.8
Mozilla#2	0.0	3.0	6.0
Mozilla#3	0.0	0.0	0.0
MySql#1	0.6	12.2	15.0
MySql#2	0.8	7.6	9.6
MySql#3	0.2	17.2	18.0
Avg	0.8	6.8	8.4
SPL2 kernels	0.0	14.7	15.3
SPL2 apps	3.3	12.7	17.6
Avg	1.6	13.7	16.4

Table 6. False positives in different scenarios.

Overall, the number of FPs in AtomTracker-D is comparable to other schemes. For example, for similar commercial codes, AVIO has 7 FPs per code (compared to 8.4 in AtomTracker-D), and SVD has 3.2 FPs per M instructions (compared to 0.16 FPs per M instructions in AtomTracker-D).

6.4. Execution Time Overhead

Table 7 shows the overhead of the complete hardware and software implementations of AtomTracker-D. The overhead of the hardware one is measured in increase in execution time and in network traffic. The execution time increases by an average of only 0.2% and 4.0% for the commercial and SPLASH-2 codes, respectively. This makes AtomTracker-D suitable for *production runs*. This low overhead results from running the algorithm in hardware in the AVM. The main source of overhead is the additional network traffic caused by first-time accesses in an AR that are intercepted by the cache. This causes on average 3.3% and 9.2% more traffic for the commercial and SPLASH-2 codes, respectively. The SPLASH-2 codes induce higher traffic because they have relatively more data sharing.

App	HW impl		SW impl	
	Execution time increase (%)	Traffic increase in bytes (%)	Slowdown (x)	
			Due to Pin	Total
Apache#1	0.1	1.3	8.7	80.4
Apache#2	0.1	1.0	6.9	74.1
Mozilla#1	0.1	5.5	2.9	14.6
Mozilla#2	0.5	6.3	1.3	2.1
Mozilla#3	0.1	2.7	1.5	1.9
MySql#1	0.1	4.2	6.2	15.9
MySql#2	0.1	1.5	7.5	13.9
MySql#3	0.3	3.8	1.8	2.8
Avg	0.2	3.3	4.6	25.7
SPL2 kernels	2.2	9.0	106.4	573.1
SPL2 apps	5.8	9.4	19.3	256.1
Avg	4.0	9.2	62.9	414.6

Table 7. Execution overhead of AtomTracker-D.

The software implementation slows down, on average, 26x and 415x the commercial and SPLASH-2 codes, respectively. Of this, Pin accounts for a 5x and 63x slowdown, respectively. Since our software implementation is highly unoptimized, these slowdowns should only be considered an upper bound. They can easily be reduced with a better implementation. Still, they are acceptable for in-house testing, especially those for the commercial codes.

6.5. Components of AtomTracker-I

As described in Section 3.2, AtomTracker-I uses a preprocessing pass that collects Critical Section (CS) and Loop (LP) information. To evaluate the contribution of this pass in finding ARs, we use three microbenchmarks for which we know the actual ARs. We cannot use our main applications because we do not know the correct ARs there. The three microbenchmarks, shown in Table 8, implement a linked list, a producer-consumer pattern, and an FFT. They have 17, 4, and 14 ARs, respectively. Table 8 shows the fraction of the correct ARs that are inferred by our algorithms. We consider four cases: the complete AtomTracker-I (ATI), ATI without the critical section information (ATI-CS), ATI without the loop information (ATI-LP), and ATI without either (ATI-CS-LP). From the average numbers, we see that AtomTracker-I identifies all the ARs. Without CS or LP information, AtomTracker-I identifies only 79% or 90% of them. So, both types of information are needed.

Micro-benchmark (# of ARs)	ARs Inferred by AtomTracker-I (ATI) versions			
	ATI (% of correct)	ATI - CS (% of correct)	ATI - LP (% of correct)	ATI - CS - LP (% of correct)
LinkedList (17)	100.0	58.8	94.1	52.9
ProdCons (4)	100.0	100.0	75.0	75.0
FFT (14)	100.0	78.6	100.0	78.6
Avg (11.7)	100.0	79.1	89.7	68.8

Table 8. Impact of the AtomTracker-I preprocessing pass.

7. Limitation: Nested Atomic Region

The current AtomTracker algorithm cannot infer nested atomic regions. Figure 11(a) shows a correct nested atomic region in thread 1. The outer region accesses *y* and the inner ones access *x*. If we run AtomTracker-I, it detects a conflict on *x* and incorrectly infers the two ARs of Figure 11(b).

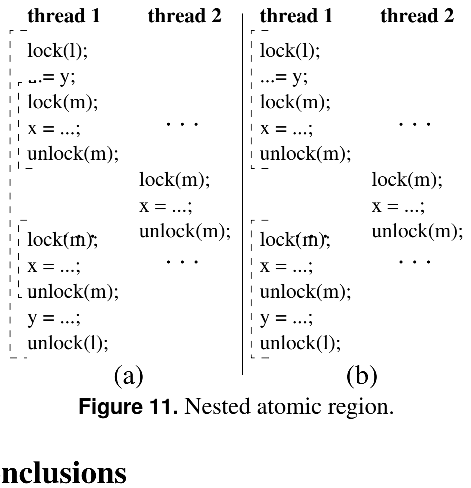


Figure 11. Nested atomic region.

8. Conclusions

This paper presented AtomTracker, the *first* scheme to (1) automatically infer *generic* non-nested ARs (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of code construct the region is embedded in) and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed. AtomTracker provides novel algorithms to infer generic ARs and to detect atomicity violations of them. Moreover, we presented a hardware implementation of the violation detection algorithm that leverages cache coherence state transitions in a multiprocessor. To evaluate AtomTracker, we took eight atomicity violation bugs from real-world codes like Apache, MySql, and Mozilla, and showed that AtomTracker detects them all — which is not the case with any of the existing approaches. In addition, AtomTracker automatically inferred all of the ARs in a set of microbenchmarks accurately. Finally, we also showed that the hardware implementation induces a negligible execution time overhead of 0.2–4.0% and, therefore, enables AtomTracker to find atomicity violations on-the-fly in production runs.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[3] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages*, January 2004.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation*, June 2003.

[5] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering*, May 2008.

[6] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles*, October 2007.

[7] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[8] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[9] B. Lucia, L. Ceze, and K. Strauss. Finding concurrency bugs with context-aware communication graphs. In *International Symposium on Computer Architecture*, December 2009.

[10] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *International Symposium on Computer Architecture*, June 2010.

[11] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *International Symposium on Computer Architecture*, June 2008.

[12] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, June 2005.

[13] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic techniques for loop detection and profiling. In *Workshop on Binary Instrumentation and Applications*, October 2006.

[14] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *International Symposium on Computer Architecture*, June 2009.

[15] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *International Symposium on Foundations of Software Engineering*, November 2008.

[16] Virtutech. Simics. `http://www.simics.net/`.

[17] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 2006.

[18] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Conference on Programming Language Design and Implementation*, June 2005.

[19] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, June 2009.