Atom-Aid: Detecting and Surviving Atomicity Violations

Abstract

Writing shared-memory parallel programs is an error-prone process. Atomicity violations are especially challenging concurrency errors. They happen when programmers make incorrect atomicity assumptions and fail to enclose memory accesses that should occur atomically inside the same critical section. If these accesses happen to be interleaved with conflicting accesses from different threads, the program might behave incorrectly.

Recent architectural proposals arbitrarily group consecutive dynamic memory operations into atomic blocks. This provides what we call implicit atomicity, as the atomic blocks are not derived from explicit program annotations.

In this paper, we make the fundamental observation that implicit atomicity probabilistically hides atomicity violations by reducing the number of interleaving opportunities between memory operations. We then propose Atom-Aid, a new approach that creates implicit atomic blocks intelligently instead of arbitrarily, dramatically reducing the probability that atomicity violations will manifest themselves. Atom-Aid can infer when an atomicity violation is likely to happen before it actually happens and avoid dangerous interleavings. Atom-Aid does not change the memory semantics observed by the software and also applies to transactional memory systems. Atom-Aid is also able to report where likely atomicity violations are in the code, providing resilience and debugability. We evaluate Atom-Aid using buggy code from applications including Apache, MySQL and XMMS. Our evaluation shows that Atom-Aid hides from 98.7% to 100% of the atomicity violations, virtually eliminating them.

1 Introduction

Extracting performance from emerging multicore architectures requires parallel programs. However, writing such programs is considered difficult and largely inaccessible to most programmers. When writing explicitly parallel programs for shared memory multiprocessors, programmers need to pay special attention to keeping shared data consistent. This is usually done by specifying critical sections using locks. However, this is typically error-prone and often leads to synchronization defects, such as data-races, dead-locks and atomicity violations. Atomicity violations are very challenging concurrency errors. They happen when programmers have wrong atomicity assumptions and fail to enclose, inside the same critical section, memory accesses that should occur atomically. We cannot afford to assume bugs will not be present in the code, so it is important both to detect them and, more importantly, to survive them.

Recent architectural proposals arbitrarily group consecutive memory operations into atomic blocks (e.g. ASO [14], Implicit Transactions [12], and BulkSC [3]). Those systems provide what we call *implicit atomicity*, as they arbitrarily execute a sequence of dynamic memory operations of a program thread in an atomic block (or chunk), significantly reducing the amount of interleaving between memory operations of different threads —

interleavings only happen at a coarse granularity.

In this paper, we make the fundamental observation that if the memory operations in an atomicity violation that were supposed to be atomic happen to be inside the same chunk, then the operations can't possibly be interleaved by memory operations from other threads since a chunk is atomic. This provides a very interesting hiding effect of atomicity violations. To leverage this observation, we propose Atom-Aid, an architecture that divides the dynamic instruction stream into chunks intelligently instead of arbitrarily, further increasing the odds that an atomicity violation will be hidden. Atom-Aid is able to infer when an atomicity violation is likely to happen *before* it actually happens. Based on this information, it chooses chunk boundaries to avoid dangerous interleavings.

1.1 An Example

Figure 1(a) has a simple classic example: `counter` is a shared variable, and both the read and update of `counter` are inside distinct critical sections under the protection of lock *L*, which means that the code is free of data-races. However the code is still incorrect, as it can happen that a call to `increment()` from another thread could be interleaved right in between the read and update of `counter`, leading to incorrect behavior (e.g. two concurrent calls to increment could cause `counter` to be incremented only once). While this is a subtle error, it is not difficult to come by as it is easy to overlook the need for atomicity, or to have misconceptions about atomicity. In fact, even programs written for transactional memory (TM) [7, 9, 11] are subject to atomicity violations.

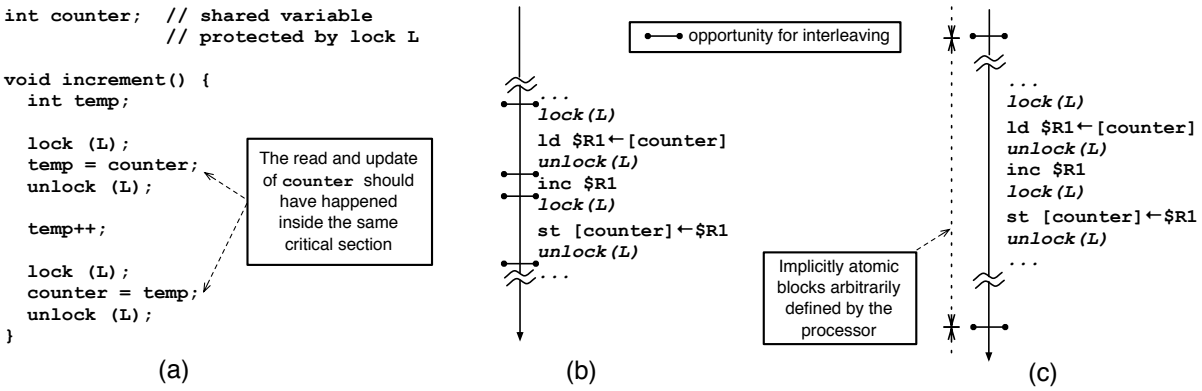


Figure 1: (a) shows a classic example of an atomicity violation. (b) shows where interleaving from other threads can happen in a traditional system, and (c) shows where such interleavings can happen in systems that provide implicit atomicity.

Atomicity violations lead to wrong program behavior if there is an unfortunate interleaving between memory

accesses of different threads that breaks the atomicity assumptions made by the programmer. The chances of an atomicity violation manifesting itself depend on the chances of such an unfortunate interleaving. In Figure 1(b), we show where interleavings can happen in traditional systems with fine-grain interleaving. There are four opportunities for interleaving. In contrast, Figure 1(c) shows where interleavings can happen in the case in which the memory operations of the atomicity violation happen to be inside the same chunk. In such case, the atomicity violation was *hidden*. In Section 3, we explain and analyze such observations in detail.

1.2 Contributions

This paper makes the following contributions. We make the fundamental observation that systems with implicit atomicity can naturally hide some atomicity violations and back it up with a probability analysis. Furthermore, we propose Atom-Aid, an architecture that uses hardware signatures to detect likely atomicity violations and dynamically adjust chunk boundaries, making the system *detect and survive* atomicity violations. To the best of our knowledge, this is the first paper on surviving atomicity violations without requiring global checkpointing. Since we do not want the atomicity violations to go unnoticed by the programmer, Atom-Aid is also able to report where likely atomicity violations are in the code, providing resilience and debugability. Finally, we provide an evaluation using real applications that shows that Atom-Aid is able to reduce the chances that an atomicity violation will lead to wrong program behavior by *several orders of magnitude*.

This paper is structured as follows. Section 2 gives background information on implicitly atomic systems and atomicity violations. In Section 3, we explain our observation with a probability study. Section 4 presents the Atom-Aid algorithm and architectural components. We present our evaluation infrastructure and results in Sections 5 and 6, respectively. Section 7 points out work related to Atom-Aid. Finally, Section 8 concludes.

2 Background

2.1 Implicit Atomicity

Recent proposals such as Atomic Sequence Ordering (ASO) [14], BulkSC [3] and Implicit Transactions [12] arbitrarily group consecutive dynamic memory operations into atomic blocks in order to support consistency enforcement at a coarse grain, rather than supporting it at the granularity of individual instructions. We say these systems provide *implicit* atomicity because such groups of dynamic memory operations do not follow any annotation in the program, as opposed to explicit transactions in TM. In essence, systems with implicit

atomicity take periodic checkpoints (e.g. every 2000 dynamic instructions) to form arbitrary groups of dynamic instructions (hereafter called *chunks* of instructions) that appear to execute atomically and in isolation. It is important to note that chunks are *not* programming constructs as transactions are in TM. Other systems, such as the TCC prototype [6, **?**], use periodic transaction commits to support legacy code, showing that implicit atomicity can be implemented as a direct extension of hardware-based TM systems, which will soon be available commercially [**?**].

The goal of supporting consistency enforcement at a coarse grain is to bridge the performance gap between strict and relaxed memory models, while keeping hardware complexity low. Enforcing memory consistency at the granularity of chunks and executing chunks atomically allows the processor to fully reorder instructions within a chunk while preserving the ordering requirements of the memory consistency model, since the order of instructions within a chunk is not exposed to remote processors. As a result, ASO [14], Implicit Transactions [12] and BulkSC [3] can all offer sequential consistency with the performance of more relaxed memory models such as release consistency [5].

Supporting coarse-grain consistency enforcement with implicit atomicity has two interesting properties, both very relevant to the work presented in this paper. First, coarse-grain memory ordering reduces the amount of interleaving of memory operations from different processors — they can only interleave at the granularity of chunks. This means that the effect of remote threads are only visible at chunk boundaries. This is what Figures 2(a) and 2(b) show: on the left side, they show units that can be interleaved (2(a): instructions, 2(b): chunks); on the right side, they show the possible interleavings. There are far fewer possible coarse-grain interleavings. Second, since the granularity of consistency enforcement is oblivious to the software, the system can arbitrarily choose chunk boundaries, adjusting the size of chunks dynamically without affecting program correctness or the memory system semantics.

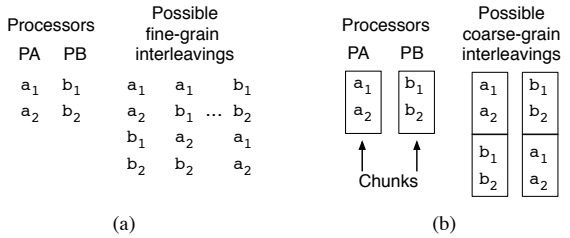


Figure 2: Fine (a) and coarse-grain (b) access interleaving. There are 6 possible interleavings for the fine-grained system and 2 possible interleavings for the coarse-grained system.

Bulk [4] is a set of hardware mechanisms that simplify the support of common operations in environments with multiple speculative tasks such as Transactional Memory (TM) and Thread-Level Speculation (TLS). A hardware module called the Bulk Disambiguation Module (BDM) dynamically summarizes the addresses that a task reads and writes into Read (*R*) and Write (*W*) signatures, respectively. A signature is an inexact encoding of addresses following the principles of Bloom Filters [1], which are subject to aliasing, representing a superset of the original address set. The BDM also includes units that perform signature operations such as union, intersection, expansion, etc.

The idea behind Atom-Aid can be implemented in any architecture that supports implicit atomicity. However, for the purpose of illustration, in this paper, we assume a system similar to BulkSC, where processors repeatedly execute chunks separated by checkpoints (no dynamic instruction is executed outside a chunk). BulkSC leverages a cache hierarchy with support for Bulk operations and a processor with efficient checkpointing. The memory subsystem is extended with an arbiter to guarantee a total order of commits. As a processor executes a chunk speculatively, it buffers the updates to memory in the cache and generates a *R* and a *W* signature. When chunk *i* completes, the processor sends the arbiter a request to commit, together with signatures R_i and W_i. The arbiter intersects R_i and W_i with the *W* signatures of all the currently-committing chunks. If all intersections are empty, W_i is saved in the arbiter and also forwarded to all interested caches for commit. Each cache uses W_i to perform bulk disambiguation and potentially abort local chunks in case a conflict exists. Chunks are periodic and boundaries are chosen arbitrarily (e.g. every 2000 instructions). Finally, forward progress is guaranteed by reducing chunk sizes on the presence of repeated chunk aborts.

2.2 Atomicity Violations

Data-races are the most widely known concurrency defects. There has been a significant amount of work on tools [**?**, **?**, **?**, **?**] and hardware support [10, **?**] for data-race detection. However, as pointed out by Flanagan *et. al.* [**?**], data-race freedom does not imply a concurrent program is correct, as the program can still have *atomicity violations*.

The code snippet in Figure 1(a) does not have a data race (all accesses to `counter` are properly synchronized), however the code is still incorrect. In this example, what is missing is *atomicity*, since both the read and update of `counter` should have been atomic to avoid unwanted interleaving of accesses to `counter` from

other threads. Atomicity is a non-interference property stronger than freedom from data-races, as pointed out by Flanagan *et. al.* [**?**]. An *atomicity violation* exists in the code when the programmer has wrong assumptions about atomicity and fails to enclose accesses that should have been performed atomically inside the same critical section. Note that atomicity violations can also exist in programs that use transactional memory-based synchronization as opposed to locks — the programmer can fail to enclose memory accesses to shared variables that are supposed to be performed atomically inside the same transaction.

If there are no atomicity violations in a concurrent execution, it is said to be *serializable*. This means that there is an equivalent serial execution of the blocks assumed to be atomic by the programmer. Conversely, if there is an atomicity violation, the concurrent execution is not serializable — i.e. there are no equivalent serial executions. We can apply this concept directly to shared data accesses by determining whether the interleavings of memory accesses to a shared variable are serializable. Lu *et. al.* [**?**] used this analysis to determine same-variable unserializable access interleavings detected by their AVIO algorithm. Table 1 reproduces the analysis presented in [**?**]. The column "Interleaving" represents the interleaving in the format $AA \times B$, where AA is the pair of local memory accesses interleaved by B, the access from a remote thread. For example, $RR \times W$ corresponds to two local read accesses interleaved by a remote write access.

Interleaving	Serializability	Comment
RR × R	serializable	—
RR × W	unserializable	The interleaved write makes the two local reads inconsistent.
RW × R	serializable	—
RW × W	unserializable	The local write may depend on the result of the earlier read, which is overwritten by the remote write before the local write.
WR × R	serializable	—
WR × W	unserializable	The local read does not get the expected value.
WW × R	unserializable	The intermediate value written by the first write is made visible to other threads.
WW × W	serializable	—

Table 1: Memory interleaving serializability analysis cases (from [**?**]).

Terminology used in this paper. Throughout this paper, we use the following terminology. *Atomicity-lacking section* is the region of code between (and including) the memory operations that were supposed to be atomic (e.g. the code between the read and update to `counter` in Figure 1(a)). The *size* of an atomicity violation is the number of dynamic instructions in the atomicity-lacking section. An *opportunity for interleaving* is any point in the execution of a thread where the operation of remote threads can become visible. This corresponds to

chunk boundaries in systems with implicit atomicity or between any pair of memory operations in conventional systems. An atomicity violation is said to be *exposed* if there is at least one opportunity for interleaving between the two memory operations that were supposed to be atomic. Conversely, an atomicity violation is said to be *hidden* if it is not exposed. An *atomicity violation manifests itself* if and only if it is exposed and unserializable (according to Table 1), which will likely lead to wrong program behavior. We refer to a *likely atomicity violation* when an unserializable interleaving could *potentially* have happened but did not necessarily happen (e.g. the local accesses and remote access of Table 1 were nearby but were not actually interleaved). Finally, we refer to implicitly atomic systems and chunk granularity systems interchangeably.

3 Implicit Atomicity Hides Atomicity Violations

In this section, we contrast chunk granularity systems with instruction granularity systems, and show that the implicit atomicity in chunk granularity systems probabilistically reduces the chances of atomicity violations being exposed and, consequently, manifesting themselves. This can happen when an atomicity-lacking section is completely enclosed in a chunk. We call this phenomenon *natural hiding*.

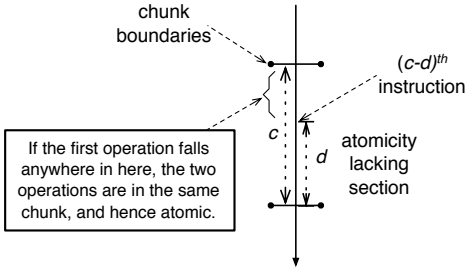
Achieving the same effect of implicit atomicity statically, by having a compiler automatically insert arbitrary transactions in a program, is very challenging because it could hurt performance, or even prevent forward progress [2]. On the other hand, systems that support implicit atomicity do provide forward progress guarantees [3].

3.1 Probability Study

We now show analytically that chunk granularity systems have a lower probability of exposing atomicity violations than instruction granularity systems. For the following analysis, let c be the default size of chunks in number of dynamic instructions; d be the size of the atomicity violation — the number of dynamic instructions in the atomicity-lacking section; and let P_{hide} be the probability of the atomicity-lacking section falling inside a single chunk — i.e., the probability of hiding the atomicity violation.

Figure 3 illustrates how we derive the probability that an atomicity-lacking section will fall inside a chunk. If the first operation of the atomicity-lacking section is within the first $(c - d)$ instructions of a chunk with a total of c instructions, the first and second atomicity-lacking operations will fall in the same chunk and will be committed atomically, hiding the atomicity violation. With this model, we can express the probability of hiding

an atomicity violation as shown in Figure 3.



$$P_{hide} = \begin{cases} 0 & \text{if } c < d \\ \frac{c-d+1}{c} & \text{if } c \geq d. \end{cases}$$

Figure 3: Atomicity-lacking section within chunk boundaries. P_{hide} is the probability that two atomicity-lacking operations will fall in the same chunk.

Note that we can assume an instruction granularity system as a system with chunk size equal to one instruction ($c = 1$), which will imply $P_{hide} = 0$, since the size of an atomicity violation is at least two instructions ($d \geq 2$). This is consistent with the intuition that an instruction granularity system cannot hide atomicity violations. Note that, in the worst case that atomicity-lacking sections are bigger than chunks in actual chunk-based systems, $P_{hide} = 0$ as well. This shows that chunk granularity systems can hide atomicity violations, but never increase the chances of them manifesting themselves.

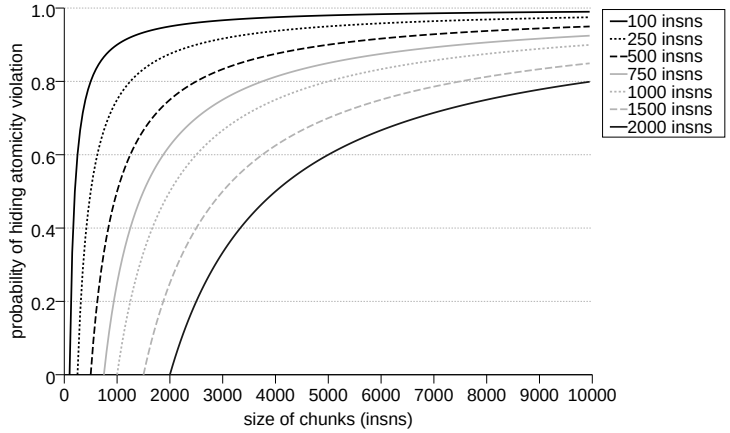


Figure 4: Probability of hiding an atomicity violation as a function of chunk size for various violation sizes.

Figure 4 shows the probability of hiding an atomicity violation for various violation sizes as chunk sizes increase. As expected, we observe that increasing chunk size increases the probability of hiding an atomicity violation, but this is subject to diminishing returns.

4 Actively Hiding Atomicity Violations with Smart Chunking

In Section 3, we showed that implicit atomicity can naturally hide some atomicity violations. We also point out in Section 2.1 that the choice of where to place chunk boundaries is arbitrary and does not affect the memory

semantics. The idea behind Atom-Aid's *smart chunking* is to automatically determine where to place chunk boundaries in order to further reduce the probability that atomicity violations are exposed. Atom-Aid does this by detecting potential atomicity violations before they happen and inserting a chunk boundary right before the first memory access of subsequent occurences of these potential violations, in the hope of enclosing both accesses in the same chunk. In essence, Atom-Aid infers where critical sections should be in the dynamic instruction stream, and inserts chunk boundaries accordingly. Note that this process is transparent to software and it is oblivious to synchronization constructs that might be present in the code.

Atom-Aid detects potential atomicity violations by observing the memory accesses of each chunk and the interleaving of memory accesses from other committing chunks in the system. When Atom-Aid detects at least two nearby accesses to the same variable a by the local thread and at least one recent access to a by another thread, it looks at the involved accesses types and determines if these accesses are *potentially* unserializable. If so, Atom-Aid starts monitoring accesses to a. When the local thread accesses a again, Atom-Aid decides if a chunk boundary should be inserted. Atom-Aid keeps a history of memory accesses by recording the read and write sets of the most recent local chunks and recently commited remote chunks.

Figure 5 shows how the idea is applied to the counter increment example, assuming BulkSC provides implicit atomicity. Atom-Aid maintains the read and write sets of the previously committed chunk, which are called R_P and W_P, respectively. Recall that, in BulkSC, processors committing chunks send their write sets to other processors in the system, allowing Atom-Aid to learn what was written recently by remote committing chunks (W_{RP}). In Figure 5, processors P1 and P2 are both executing `increment()`. While there is a chance that the read and update of `counter` will be atomic due to natural hiding, in Figure 5(a) that did not happen. In the example, the read of the counter is inside the previously committed chunk (P), while the update is part of the chunk currently being built (C). When `counter` is updated in C, Atom-Aid determines that `counter` was read by the previous local chunk (`counter` $\in R_P$) and recently updated by a remote processor (`counter` $\in W_{RP}$). This characterizes a potential violation, making `counter` a member of the set of all variables possibly involved in an atomicity violation — the *chunkBreakSet*. Later (Figure 5(b)), when P1 accesses `counter` again, Atom-Aid detects that `counter` $\in$ *chunkBreakSet* and therefore a chunk boundary should be inserted *before* the read from `counter` is executed. This increases the chances that both accesses to `counter` will be enclosed in the

same chunk, making them atomic. While in this example the atomicity violation is exposed, it does not mean it has manifested itself. Also, even if the atomicity violation is naturally hidden, Atom-Aid is still able to detect it. This means that Atom-Aid is able to detect atomicity violations before they manifest themselves.

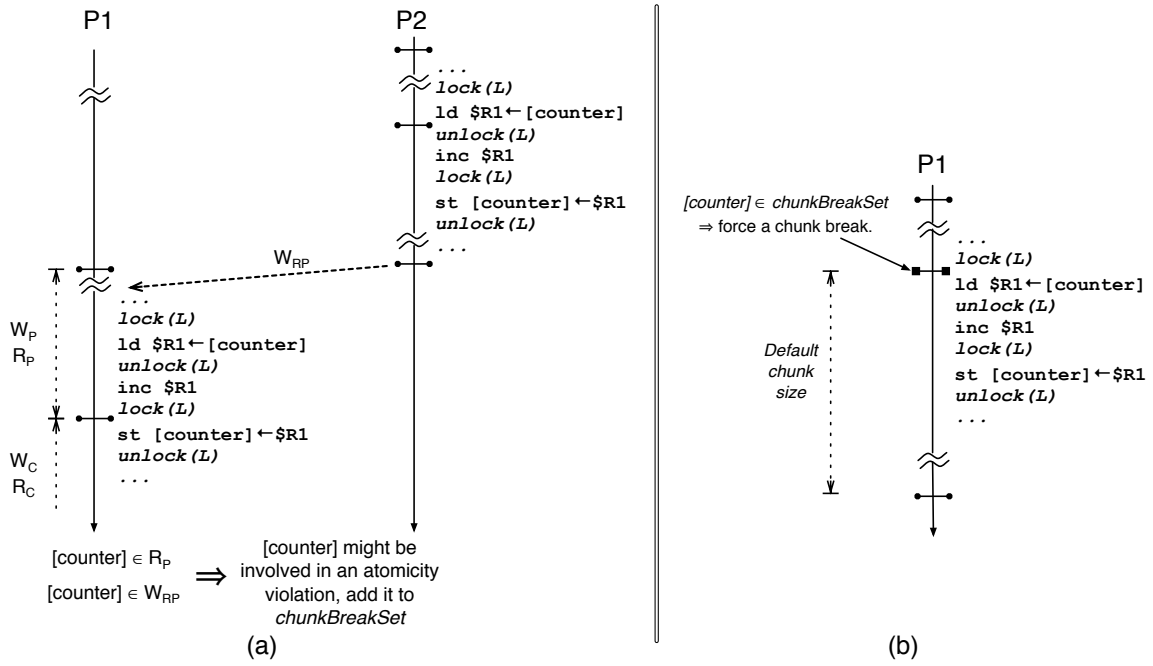


Figure 5: Example on actively hiding an atomicity violation. (a) shows how Atom-Aid discovers that `counter` might be involved in an atomicity violation and adds it to the *chunkBreakSet*. (b) shows that when `counter` is accessed, a chunk is automatically broken because it belongs to the *chunkBreakSet*

In the following sections, we explain in detail how the detection algorithm works and how Atom-Aid decides where to place chunk boundaries. We also describe an architecture built around signature operations that implements the mechanisms used by the algorithm.

4.1 Detecting Likely Atomicity Violations

The goal of Atom-Aid is to detect potential atomicity violations *before* they happen. When it finds two nearby accesses by the local thread to the same address and one recent access by another thread to that same address, Atom-Aid examines the types of accesses to determine whether they are potentially unserializable. If they are, Atom-Aid treats them as potential atomicity violations. Atom-Aid needs to keep track of three pieces of information: (i) the type t (read or write) and address a of the memory operation currently executing; (ii) the read and write sets of the the current and previously committed chunk, referred to as R_C, W_C, R_P and W_P, respectively; and (iii) the read and write sets of chunks committed by remote processors while the previously

committed chunk was executing (referred to as R_{RP} and W_{RP}), together with read and write sets of chunks committed by other processors while the current chunk is executing (referred to as R_{RC} and W_{RC}).

Table 2 shows how this information is used to determine whether these accesses constitute potential atomicity violations. The first column shows the type of a given local memory access, the second column shows which interleavings Atom-Aid tries to identify when it observes this local memory access, and the third column shows how Atom-Aid identifies them. For example, consider the first two cases: when the local memory access is a read, the two possible non-serializable interleavings are $RR \times W$ and $WR \times W$. To detect if any of them has happened, Atom-Aid uses the corresponding set expressions in the third column. Specifically, to identify a potential $RR \times W$ interleaving, Atom-Aid first checks whether a can be found in any of the local read sets ($a \in R_C \vee a \in R_P$). If it is, Atom-Aid then checks whether a can also be found in any of the remote write sets of a chunk committed by another processor either while the previous local chunk was executing ($a \in W_{RP}$) or since the beginning of the current local chunk ($a \in W_{RC}$). If the condition is satisfied, Atom-Aid identifies address a as potentially involved in an atomicity violation and adds it to the processor's *chunkBreakSet*. Note that this case is not necessarily an atomicity violation because the remote write might not have actually interleaved between the two reads. Since Atom-Aid keeps only two chunks' worth of history, it is only capable of detecting atomicity violations that are shorter than the size of two chunks, which is not a problem, since Atom-Aid does not hide atomicity violations larger than a chunk.

Local operation	Interleaving case	Expression
read	RR × W	$(a \in R_C \vee a \in R_P) \wedge (a \in W_{RP} \vee a \in W_{RC})$
	WR × W	$(a \in W_C \vee a \in W_P) \wedge (a \in W_{RP} \vee a \in W_{RC})$
write	RW × W	$(a \in R_C \vee a \in R_P) \wedge (a \in W_{RP} \vee a \in W_{RC})$
	WW × R	$(a \in W_C \vee a \in R_P) \wedge (a \in W_{RP} \vee a \in R_{RC})$

Table 2: Cases when an address is added to the *chunkBreakSet*.

4.2 Adjusting Chunk Boundaries

After an address is added to the processor's *chunkBreakSet*, every access to this address by the local thread triggers Atom-Aid. If Atom-Aid placed a chunk boundary right before all accesses that trigger it, Atom-Aid would not actually prevent any atomicity violation from being exposed. To see why, consider Figure 5(b) again. Suppose the address of variable `counter` has been previously placed in the *chunkBreakSet*. When the load from `counter` executes, it triggers Atom-Aid, which can then place a chunk boundary right before this access.

When the store to `counter` executes, it triggers Atom-Aid again. If it placed another chunk boundary at that point, Atom-Aid would actually expose the atomicity violation, instead of hiding it as intended.

There are other situations in which breaking a chunk by placing a new chunk boundary is undesirable. For example, atomicity violations involving several variables might cause Atom-Aid to be invoked several times. We actually want Atom-Aid to place a chunk boundary before the first access, but not before every single access. Another example is the case when an address has just been added to the *chunkBreakSet*. Chances are that the local thread is still manipulating the corresponding variable, in which case avoiding a chunk break can actually be beneficial.

To intelligently determine whether to place a chunk boundary when it is triggered, Atom-Aid uses a simple policy consisting of two conditions. Figure 6 shows this policy in a flowchart. The first condition (1) determines that Atom-Aid never breaks a chunk more than once — after a forced break, the newly created chunk will be as large as the default chunk size. The second condition (2) determines that, if Atom-Aid adds any address to the *chunkBreakSet* during the execution of a given chunk, it cannot break that chunk.

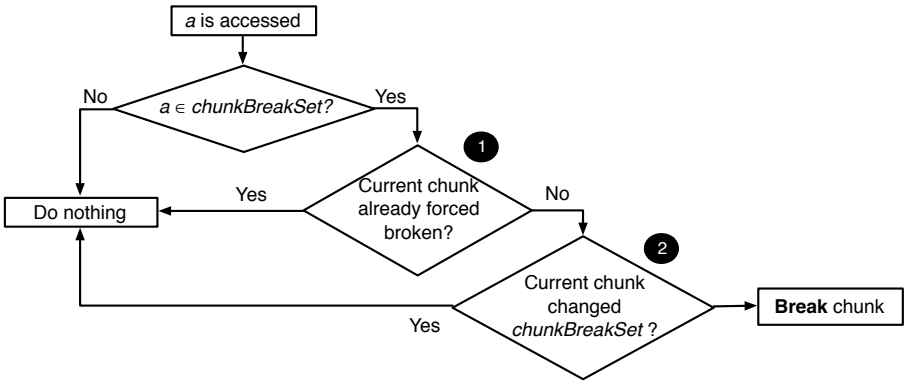


Figure 6: Chunk breaking policy in a flowchart.

4.3 An Architecture Based on Signature Operations

We based our implementation around BulkSC because its signatures offer a convenient way of storing chunk read and write sets. To collect the information required by the algorithm described in Section 4.1, we add three pairs of signatures to the original BulkSC design. Figure 7 shows all required signatures. Signatures R_C and W_C, which hold the read and write sets of the currently executing chunk, are used by BulkSC for chunk disambiguation and memory versioning. Signatures R_P and W_P hold the read and write signatures of the previously committed chunk. When a chunk commits, R_C and W_C are copied into R_P and W_P, respectively.

Signature R_{RC} encodes all local downgrades due to remote read requests received while the current chunk executes. Likewise, signature W_{RC} holds all signatures received from remote processors while the current chunk executes. When the current chunk commits, signatures R_{RC} and W_{RC} are copied into signatures R_{RP} and W_{RP}, respectively, which thus encode the remote operations that happened during the execution of the previous chunk. If a chunk is aborted, only signatures R_C and W_C are discarded, keeping the rest of the memory access history intact.

As mentioned earlier, we want Atom-Aid to be useful as a debugging resource as well. We envision doing this by making the *chunkBreakSet* visible to software and providing a fast user-level trapping mechanism that is triggered at every memory access to addresses in *chunkBreakSet*.

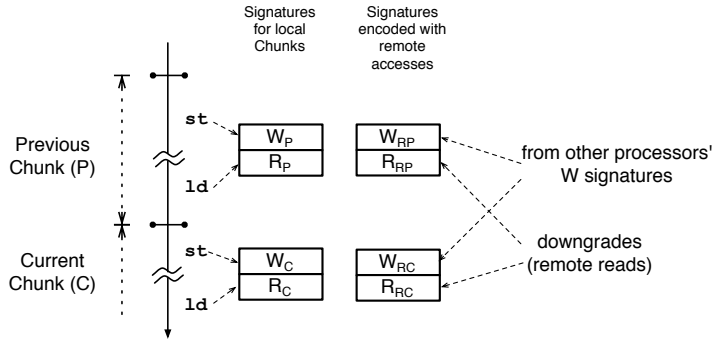


Figure 7: Signatures used by Atom-Aid to detect likely atomicity violations.

The *chunkBreakSet* itself can be implemented by using yet another signature. In this case, checking if an address is in the *chunkBreakSet* is done with a simple membership operation on the signature. Alternatively, the *chunkBreakSet* can be implemented with extensions to the cache tag. An extra bit per cache line indicates whether or not the corresponding line address is part of the *chunkBreakSet*. In this case, determining if a referenced address is in the *chunkBreakSet* is done when the cache line is accessed by checking if the corresponding bit is set.

Both the signature and cache-based implementations can support word-granularity addresses. With the signature-based implementation, this can be done by simply encoding word addresses as opposed to line addresses. With the cache-based implementation, this can be done by having one bit per word in a cache line to indicate whether the corresponding word is present in the *chunkBreakSet*.

The trade-off between these two implementations is one between complexity and effectiveness. While a

signature-based implementation is simpler and does not require the address to be present in the cache, it suffers from aliasing (false positives), especially if the *chunkBreakSet* involves many data addresses. With the cache-based implementation there is no aliasing, but the implementation of this approach is more complex. In addition, the *chunkBreakSet* information of a particular cache line is lost on cache displacements.

4.3.1 Implementation Issues

While we have assumed a BulkSC-like system in this paper, other systems that support implicit atomicity can similarly benefit from Atom-Aid. Take, for example, a TCC-like [6] system with a mechanism for automatic hardware-defined (implicit) transactions. In TCC, disambiguation is not done with signatures, but write sets are still sent between processors during commit. It is possible to record the information Atom-Aid needs by augmenting TCC with structures to hold the incoming write sets when remote processors commit. It is also possible to use similar structures to hold the read and write sets for previously executed chunks.

The performance impact of the architectural structures required by Atom-Aid is negligible. The membership operation with signatures is very fast because it does not involve any associative search and only requires simple logic [4, **?**]. As a result, accessing signatures is likely to be much faster than accessing the cache to read or modify data. Also, all accesses to signatures required by both the detection algorithm and the chunk breaking policy can be done in parallel. In case the *chunkBreakSet* is implemented as an extension to the cache tags, it is also unlikely that it will affect performance, since both the data and the bit indicating that the corresponding address is part of the *chunkBreakSet* can be fetched simultaneously. One final implementation detail is that only *chunkBreakSet* must be preserved through context switches.

5 Experimental Setup

5.1 Simulation Infrastructure

We model an implicitly atomic system resembling BulkSC [3] using PIN [8], a dynamic binary instrumentation infrastructure. Our simulator runs parallel workloads and can be configured to accurately build, record and communicate read and write sets, using either inexact signatures or exact sets. Unless otherwise noted, we use inexact signatures in our experiments. Since the simulator runs workloads in a real multiprocessor environment, it is subject to non-determinism. For this reason, we present results averaged across a number of runs, with error bars showing the 95% confidence interval for the average.

In our simulations, we want to be able to check whether a particular atomicity-lacking section is fully enclosed in a chunk. In order to do that, we explicitly mark the code with the beginning and end of each atomicity-lacking section. It is important to note that the sole purpose of collecting this information is to evaluate the techniques we propose; Atom-Aid does not use this information in any way.

5.2 Simulated Workloads

For our experiments, we use two types of workloads: sandboxed bugs and entire applications. The goal of using sandboxed bugs is to generate extreme conditions in which atomicity-lacking sections are likely executed more often than in real applications. We can then use them to stress-test Atom-Aid in a short amount of time. We also include entire applications (MySQL, Apache, XMMS) for a more complete evaluation. For MySQL runs, we used the SQL-bench test-insert workload, which performs insert and select queries. For Apache runs, we used the ApacheBench workload. For XMMS, we played a media file with the visualizer on.

We created sandboxed bugs from real applications based on previous literature on atomicity violations [**?**, 15, **?**]. We made sure the atomicity violation in the original application remains intact in the sandboxed version. Wherever possible, we also included program elements which especially affect timing, such as I/O, to mimic realistic interleaving behavior in the sandboxed workloads.

Table 3 lists the workloads we use in our evaluation, dividing them into two categories: sandboxed bugs and real applications. We provide the average atomicity violation size for these applications (column 3) and the number of threads each workload uses (column 4), along with a brief description of each (column 5). Note that we have a reasonably wide range of violation sizes, from 90 to 3.5k dynamic instructions, but that all of them, including the ones found in real applications, are only as large as a few thousand instructions. Note that for Apache and MySQL, the violation sizes in the full application and the sandboxed versions are different. This is because in Apache-extract there was additional work in generating random log entries, and MySQL-extract does not use the custom implementation of `memcpy` MySQL does.

For the experiments we present in Section 6, we simulate each of the sandboxed bugs 40 times for each chunk size, and we vary chunk size from 750 to 8000 instructions. Due to restrictions in simulation time, we run the real applications 5 times, with a chunk size of 4000 instructions.

Workload Type	Workload Name	Violation Size (Avg)	# Threads	Description
real	Apache	464	25	Logging bug in Apache httpd-2.0.48. Two threads access same log entry without holding locks and change entry length. This leads to missing data or crash.
	MySQL	730	28	Security backdoor in mysql-4.0.12. While one thread closes file pointer and sets log status variable to closed, other thread tries to write log. Logging thread sees closed log, and discards entries.
	XMMS	586	6	Visualization system bug in xmms-1.2.10, a popular media player. Visualizer is accessing PCM stream data, and in certain interleavings data in PCM can be changed, or freed during partial access, causing corruption, or crash.
sandboxed	Apache-extract	979	2	Sandboxed version of above Apache log system bug.
	BankAccount	88	2	Shared bank account data structure bug. Simultaneous withdrawal and deposit with incorrectly synchronized program may lead to inconsistent final balance.
	BankAccount2	2406	2	Same as previous, with larger atomicity violation.
	CircularList	588	2	Shared work list data ordering bug. Removing, processing and adding work units to list non-atomically may reorder work units in list.
	CircularList2	3592	2	Same as previous, with larger atomicity violation.
	LogProc&Sweep	280	5	Shared data structure NULL dereference bug. Threads inconsistently manipulate shared log. One thread sets log pointer to NULL, another reads it and crashes.
	LogProc&Sweep2	2494	5	Same as previous, with larger atomicity violation.
	MySQL-extract	244	2	Sandboxed version of above MySQL log system bug.
	StringBuffer	556	2	java.lang.StringBuffer overflow bug [**?**]. On append, not all required locks are held. Another thread may change buffer during append. State becomes inconsistent.

Table 3: Bug benchmarks used to evaluate Atom-Aid.

6 Evaluation

Our evaluation is divided into four parts. Section 6.1 shows an experimental validation of our probability study presented in Section 3.1. Section 6.2 shows that Atom-Aid completely hides most instances of atomicity violations. Section 6.3 presents data on the dynamic behavior of Atom-Aid's architectural structures, including a comparison with an implementation that does not use hardware signatures. Finally, Section 6.4 discusses how the information derived by Atom-Aid can be used to help locate bugs in large software packages.

6.1 Natural Hiding

We ran the workloads from Table 3 on our simulator with Atom-Aid turned off to determine how often atomicity violation instances naturally fall within the same chunk. These numbers should follow the probability derived in Section 3.1.

Figure 8 plots the percentage of atomicity violation instances naturally hidden as the chunk size increases from 750 to 8000 dynamic instructions. Table 4 shows the same information for the full applications and chunk size of 4000 dynamic instructions.

With the information of violation sizes from Table 3, we can verify that the numbers plotted in Figure 8, as well as the numbers in Table 4, match the expected probability shown in Figure 4. For example, take MySQL-extract, which has an average violation size of 244 dynamic instructions. Its curve in Figure 8 matches almost

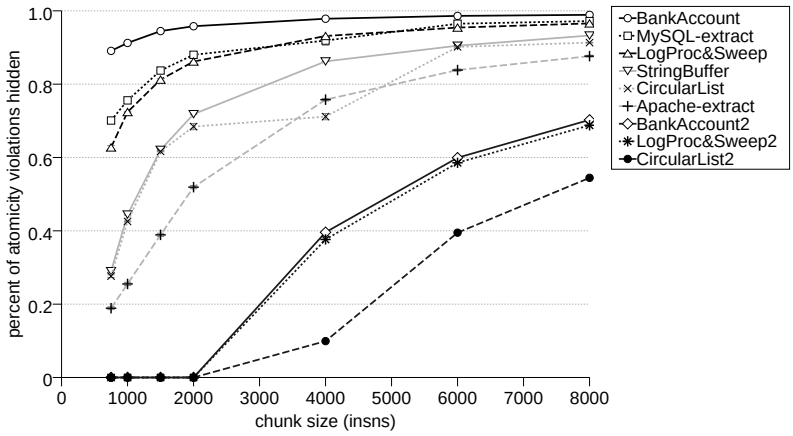


Figure 8: Experimental data on the natural hiding of atomicity violations with implicit atomicity for various chunk sizes.

exactly the curve for the violation size of 250 instructions in Figure 4. Overall, note that implicit atomicity with chunk sizes as low as 4000 instructions is able to hide 70% or more of the atomicity violation instances for 9 out of 12 workloads. Note that for BankAccount2, CircularList2 and LogProc&Sweep2, the percentage of atomicity violations hidden is zero for chunk sizes 2000 and lower, because their atomicity violation size is larger than 2000 instructions.

6.2 Active Hiding with Smart Chunking

We now assess how Atom-Aid improves the hiding capabilities of implicit atomicity with smart chunking. Figure 9 and Table 4 show the percentage of atomicity violations hidden by Atom-Aid. The data shows that for chunk sizes of 4000 dynamic instructions or more, Atom-Aid is able to hide virtually 100% of atomicity violation instances present in our bug-benchmarks, including the real applications. This is a remarkable result. For smaller chunk sizes, the results also show that, for most benchmarks, the majority of atomicity violation instances were also hidden, with the exception of Apache-extract and the three sandboxed bugs with larger atomicity violation sizes pointed out in Section 6.1. Apache-extract suffers from some early chunk breaks, which decreases the chances of hiding atomicity violations with smaller chunk sizes, but is not a problem for chunk sizes 4000 and larger. Still, for Apache-extract the percentage of hidden atomicity violations is much higher with active hiding than with natural hiding.

Application	% Hidden Natural	% Hidden Smart
Apache	86.25	98.96
MySQL	81.53	99.92
XMMS	85.40	98.76

Table 4: Full application experiments with chunk size set to 4000.

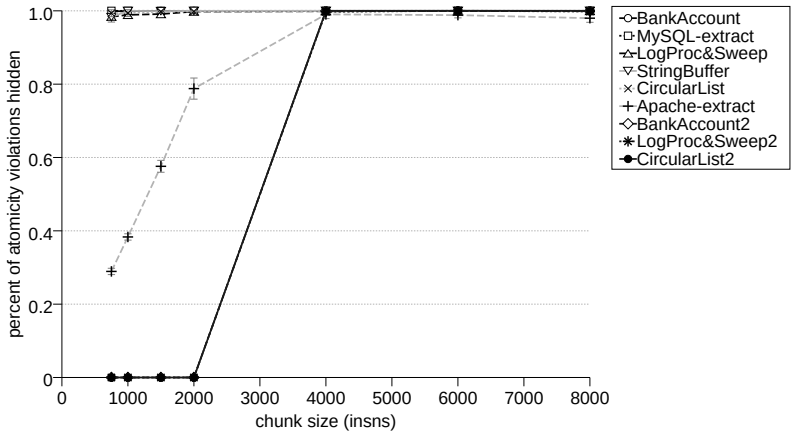


Figure 9: Percentage of atomicity violations hidden by Atom-Aid.

6.3 Characterization and Sensitivity

Table 5 characterizes Atom-Aid's behavior. To make it easier to understand the effects of smart chunking, we reproduce some data from Figure 8 and Figure 9, showing the percentage of atomicity violations hidden for natural and smart chunking with a chunk size of 4000 instructions (Columns 2 and 3). On average, while 67% of atomicity violations are hidden naturally, smart chunking was able to hide virtually 100% of them. Column 4 (*% Smart Chunks*) shows what fraction of chunks in the program were created by the smart chunking algorithm, while Column 5 (*% Unnecessary Breaks*) shows what percentage of chunk breaks did not help in hiding atomicity violations. *% Unnecessary Breaks* can be large for some workloads, which shows that Atom-Aid may unnecessarily create chunks. However, *% Smart Chunks* shows that, even with the unnecessarily created chunks, Atom-Aid still creates only a small fraction of all chunks, and is unlikely to have noticeable impact on performance [3].

Columns 6 (*chunkBreakSet Size*) and 7 (*# Break PCs*) tell, respectively, how many distinct data addresses (line granularity) were identified as involved in a potential violation, and how many distinct static memory operations in the code caused a chunk to be broken. This illustrates the behavior of Atom-Aid's atomicity violation detection algorithm. It selects, on average, only 4 data items, which leads to only 3 places in the program where chunks are broken. These numbers show that Atom-Aid is quite selective in identifying potential atomicity violations but is still able to hide almost all violations that show up during the program execution (Column 3).

So far, we have discussed data on the implementation of Atom-Aid that exclusively uses hardware signatures

Bug Benchmark	Natural	Signature-Based Atom-Aid					Exact Atom-Aid		
	% Hidden	% Hidden	% Smart Chunks	% Unnecessary Breaks	*chunkBreakSet* Size	# Break PCs	% Hidden	% Smart Chunks	% Unnecessary Breaks
Apache-extract	75.77	99.03	4.4	79.4	5	3	99.94	1.1	16.7
BankAccount	97.84	99.99	12.5	75.2	4	3	99.99	6.4	50.4
BankAccount2	39.6	100.00	11.7	74.8	4	3	100.00	6.1	49.6
CircularList	71.14	99.95	12.5	0.0	2	2	99.95	12.5	0.0
CircularList2	9.92	99.90	11.1	0.1	2	2	99.90	11.1	0.1
LogProc&Sweep	93.14	99.88	12.4	0.2	11	4	99.89	12.4	0.4
LogProc&Sweep2	37.64	99.73	11.0	0.1	2	2	99.78	11.0	0.1
MySQL-extract	91.89	100.00	18.8	46.1	3	6	100.00	18.7	45.6
StringBuffer	86.21	100.00	6.2	0.0	3	2	100.00	6.2	0.0
Average	67.02	99.83	11.2	30.6	4	3	99.94	9.5	18.1

Table 5: Characterization of Atom-Aid for both the signature and cache-based implementations.

for chunk disambiguation, for detecting chunk interleaving and for maintaining the *chunkBreakSet*. The *Exact Atom-Aid* numbers allude to the behavior of a non-signature based implementation of Atom-Aid, possibly on a traditional TM system [6, 11, 9]. For that, all signatures in the design presented in Section 4.3 are simulated ideally — there is no aliasing when detecting potential violations or when determining if a memory address is in the *chunkBreakSet* and a chunk should be broken. That data is shown in the group of columns entitled *Exact Atom-Aid* in Table 5. As expected, *% Smart Chunks* (Columns 4 and 9) is, on average, higher for the signature-based Atom-Aid, since aliasing causes chunks to be broken more frequently. However, the difference is small. This phenomenon is also reflected in the percentage of unnecessary breaks (Columns 5 and 10), which is significantly lower in *Exact Atom-Aid* — as noted before, this has negligible impact on performance. Finally, the *% Hidden* for the exact version is essentially the same as the signature-based, showing that the impact of aliasing on the effectiveness of Atom-Aid is negligible.

6.4 Debugability Discussion

Showing that Atom-Aid is able to hide almost all atomicity violation instances demonstrates that the algorithm works well enough to place chunk boundaries at the right places. Atom-Aid is also able to report the program counter (PC) of the memory instruction where chunk boundaries were automatically inserted. Since these places in the program are the boundaries of potentially buggy or missing critical sections, they can be used to aid the process of locating bugs in the code. While a detailed analysis of a complete debugging tool is out of the scope of this paper, we now show that using the feedback from Atom-Aid, we were able to locate the code for the bugs in MySQL and Apache used in past work on bug detection [**?**, 15] and even detect a new bug in XMMS.

We used the following process to locate bugs: (i) collect the set of PCs where chunk boundaries were

inserted; (ii) group PCs into the *line of code* and *function* in which they appear; and finally (iii) traverse the resulting list of functions, from most frequently-appearing to least, and then examine the lines of each function, from the most frequently-appearing line to least. Using this process, we were able to locate bugs by inspecting a relatively few number of points in the code. Table 6 shows some data on our experience. With Apache, only 85 lines of code in 6 files needed to be inspected to locate the bug. With MySQL, this number is larger (over 300), but MySQL has almost 400k lines of code. We identified a bug in XMMS that was *not previously known* after inspecting only 9 lines of code. Overall, there is room for improvement in using Atom-Aid's output for debugging, especially by coupling it with static analysis and more elaborate statistics.

	Program Totals		# of Places with Chunk Breaks			# of Inspections		
	Files	Lines	Files	Function	Lines	Files	Function	Lines
Apache	729	290k	52	206	956	6	8	85
MySQL	871	394k	44	228	681	27	84	353
XMMS	268	81k	7	23	42	2	4	9

Table 6: Characteristics of the bug detection process for real applications using Atom-Aid.

7 Related Work

Atom-Aid is a hardware-supported mechanism to detect and survive atomicity violations. While there is a significant amount of work on concurrency bug detection, few works address concurrency bug survival.

The most relevant prior work on hardware support for atomicity violation detection is AVIO [**?**]. AVIO uses training runs to extract interleaving invariants and then checks if these invariants hold in future runs. AVIO monitors interleaving by extending the caches and leveraging the cache coherence protocol. In contrast, Atom-Aid monitors interleavings in a new way, by leveraging hardware signatures. In addition, Atom-Aid does not distinguish training from detection, and leverages implicit atomicity to survive concurrency bugs.

Serializability Violation Detection (SVD) [15] uses a heuristic to infer potential critical sections based on data and control dependences with the goal of determining if they are unserializable. In [15], the authors briefly mention that their algorithm could possibly be implemented in hardware and bug avoidance could be offered using a global checkpointing mechanism [**?**]. Atom-Aid, like SVD, attempts to infer critical section boundaries dynamically. However, Atom-Aid determines *potentially* unserializable accesses to enable proactive bug avoidance. In addition, Atom-Aid avoids bugs without using global checkpointing mechanisms, relying only on implicit atomicity.

ReEnact [10] is another hardware proposal that targets concurrency bugs. However, it focuses on identifying and surviving data-races only, not atomicity violations, as Atom-Aid does. It discusses attempting to automatically repair data-races based on a library of race patterns.

There has been substantial work on hardware-supported TM systems [7, 6, 11, 9], as well as languages with new constructs for atomicity [**?**, **?**, 13, **?**]. Note that Atom-Aid can also be applied to these new proposals because all of them are still subject to atomicity violations caused by the programmer specifying incorrect atomicity constraints.

8 Conclusion

With parallel programming going mainstream, it is inevitable that programmers will have to deal with concurrency bugs, as such bugs are very easy to introduce and very difficult to remove. For these reasons, we believe that multiprocessor systems should not only help detect bugs but also *survive* them. Atomicity violations are a common and challenging category of concurrency bugs as they are often the result of incorrect assumptions about atomicity made by the programmer.

In this paper, we have shown that implicit atomicity has the property of naturally hiding some atomicity violations by significantly reducing the degree of memory operation interleaving. We justify this observation with a probability analysis and extensive experimental data. Building on top of this observation, we proposed Atom-Aid, a new approach to detecting potential atomicity violations and proactively choosing chunk boundaries to avoid exposing the violations without requiring any special program annotation or global checkpointing mechanism.

In our evaluation of Atom-Aid using both sandboxed versions of known bugs from the literature, as well as full applications such as MySQL, Apache and XMMS, we show that Atom-Aid reduces the chances that an atomicity violation will lead to wrong program behavior by several orders of magnitude, and in some cases hiding 100% of the atomicity violations. We also show that the information derived by Atom-Aid to guide chunk boundary placement can be used to aid debugging efforts. We believe Atom-Aid is a meaningful step toward a system that offers both resilience and detectability of concurrency bugs. Our future work will focus on the debugging abilities of Atom-Aid, improving the mechanisms for feeding useful information back to the programmer.

References

[1] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, July 1970.

[2] C. Blundell, E. Lewis, and M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of the 4th Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[4] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Inter. Symp. on Computer Architecture*, June 2006.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, June 1990.

[6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, June 2004.

[7] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, May 1993.

[8] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation*, June 2005.

[9] K. Moore, J. Bobba, M. J. Moravam, M. Hill, and D. Wood. LogTM: Log-based Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, February 2006.

[10] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 110–121, June 2003.

[11] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *International Symposium on Computer Architecture*, June 2005.

[12] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *International Conference on Pervasive Services*, July 2005.

[13] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *Symposium on Principles of Programming Languages*, February 2006.

[14] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.

[15] M. Xu, R. Bodik, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *Conference on Programming Language Design and Implementation*, June 2005.