Atom-Aid: Detecting and Surviving Atomicity Violations*

Brandon Lucia[†] Joseph Devietti[†] Karin Strauss[†‡] Luis Ceze[†]

[†]Computer Science and Engineering
University of Washington
{blucia0a, devietti, kstrauss, luisceze}@cs.washington.edu

[‡]Advanced Architecture and Technology Laboratory
AMD

Abstract

Writing shared-memory parallel programs is error-prone. Among the concurrency errors that programmers often face are atomicity violations, which are especially challenging. They happen when programmers make incorrect assumptions about atomicity 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 to enforce memory ordering at a coarse grain. 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, which creates implicit atomic blocks intelligently instead of arbitrarily, dramatically reducing the probability that atomicity violations will manifest themselves. Atom-Aid is also able to report where atomicity violations might exist in the code, providing resilience and debuggability. We evaluate Atom-Aid using buggy code from applications including Apache, MySQL, and XMMS, showing that Atom-Aid virtually eliminates the manifestation of atomicity violations.

1. Introduction

Extracting performance from emerging multicore architectures requires parallel programs. However, writing such programs is 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, deadlocks, and atomicity violations.

Atomicity violations are very challenging concurrency errors. They happen when programmers make incorrect assumptions about atomicity and fail to enclose memory accesses that should occur atomically inside the same critical section. According to a recent comprehensive study [15], atomicity violations account for about two thirds of all the examined non-deadlock concurrency bugs. These bugs are hard to find due to their subtle nature and the non-determinism in multithreaded execution. Hence, we can not afford to assume code will be free of bugs, so it is important to both detect bugs and survive them by preventing their manifestation. Interestingly, the manifestation of concurrency bugs is very much influenced by how multithreaded programs are executed, which determines the global interleaving of memory operations. For a given memory semantics exposed to the software, there are multiple valid global interleavings of memory operations. We can choose to allow only a subset of those interleavings to avoid concurrency bugs while still exposing the same memory semantics to the software. We leverage this property in our work.

Recent architectural proposals arbitrarily group consecutive memory operations into atomic blocks (*e.g.*, BulkSC [6], ASO [29] and Implicit Transactions [26]). Those systems provide what we call *implicit atomicity*, as they arbitrarily group a sequence of dynamic memory operations of a program thread into an atomic block (or chunk) without following any program annotation. This significantly reduces the amount of interleaving between memory operations of different threads, as interleavings only happen at a coarse granularity. The main goal of such proposals is to bridge the performance gap of memory consistency models by enforcing consistency at a coarse grain.

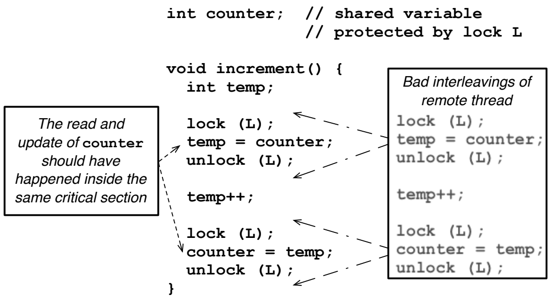
In this paper, we show that *implicit atomicity* has the very interesting property of being able to hide atomicity violations if their memory operations happen to fall within the boundaries of a chunk. We build upon this observation and propose Atom-Aid, an architecture that divides the dynamic instruction stream into chunks intelligently instead of arbitrarily, further increasing the odds that an atomicity violation will be hidden.

Figure 1 shows a simple 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*, implying that the code is free of data-races. However, the code is still incorrect as 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). This is a subtle error, and an easy mistake to make, since determining when atomicity is necessary, and which operations need to be grouped atomically can be difficult and prone to misconceptions. In fact, even programs written for transactional memory (TM) [12, 14, 18, 22] are subject to atomicity violations if the transactions are not inserted in the appropriate places.

Atomicity violations lead to incorrect program behavior if there is an unfortunate interleaving between memory accesses of differ-

[*] This work was supported in part by the National Science Foundation under grant CNS-0720593 and a gift from Microsoft Research.

1063-6897/08 $25.00 © 2008 IEEE
DOI 10.1109/ISCA.2008.4



```
int counter;  // shared variable
              // protected by lock L

void increment() {
  int temp;

  lock (L);
  temp = counter;
  unlock (L);

  temp++;

  lock (L);
  counter = temp;
  unlock (L);
}
```

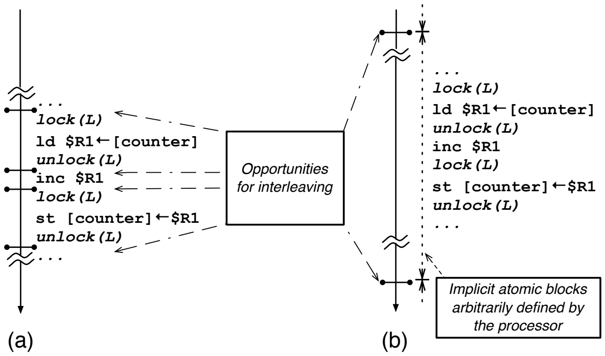
Bad interleavings of remote thread

```
lock (L);
temp = counter;
unlock (L);

temp++;

lock (L);
counter = temp;
unlock (L);
```

The read and update of counter should have happened inside the same critical section

Figure 1. A simple example of an atomicity violation. The read and update of `counter` from two threads may interleave such that the counter is incremented only once.

ent threads that breaks the atomicity assumptions made by the programmer. The chance of an atomicity violation manifesting itself depends on the chance of such an unfortunate interleaving. In Figure 2(a), we show the four opportunities where interleavings can happen in traditional systems with fine-grain interleaving. In contrast, Figure 2(b) shows where interleavings can happen when the memory operations of the atomicity violation happen to be inside the same chunk. In these cases, the atomicity violation is *hidden*. In Section 3, we explain and analyze this observation in detail.



```
...
lock(L)
ld $R1←[counter]
unlock(L)
inc $R1
lock(L)
st [counter]←$R1
unlock(L)
...
```

Opportunities for interleaving
Implicit atomic blocks arbitrarily defined by the processor
(a)
(b)

Figure 2. Opportunities for interleaving. (a) shows where interleaving from other threads can happen in a traditional system. (b) shows where such interleavings can happen in systems that provide implicit atomicity.

This paper makes two contributions. First, we make the fundamental observation that systems with implicit atomicity can naturally hide some atomicity violations. We justify this observation with a probability analysis and empirical evidence. Second, we propose Atom-Aid, an architecture that uses hardware signatures to detect likely atomicity violations and dynamically adjust chunk boundaries, making the system both *detect and survive* atomicity violations without requiring any program annotation. To the best of our knowledge, this is the first paper on surviving atomicity violations without requiring global checkpointing and recovery [21, 25, 30]. Since we do not want atomicity violations to go unnoticed by the programmer, Atom-Aid is also able to report where atomicity violations may exist in the code, providing resilience and debuggability. Finally, we provide an evaluation using buggy code from real applications which 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 (98.7% to 100% reduction).

This paper is organized 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. Finally, Section 7 discusses related work and Section 8 concludes.

2. Background

2.1. Implicit Atomicity

In systems that support implicit atomicity, memory operations in the dynamic instruction stream are arbitrarily grouped into atomic chunks. This way, consistency enforcement can be supported at the coarse grain of chunks, as opposed to being supported at the granularity of individual instructions. Examples of such systems are BulkSC [6], Atomic Sequence Ordering (ASO) [29], and Implicit Transactions [26], all of them proposed recently. We say these systems provide *implicit* atomicity because chunks do not follow any annotation in the program, in contrast to explicit atomic transactions in TM. In essence, systems with implicit atomicity take periodic checkpoints (*e.g.*, every 2,000 dynamic instructions) to form chunks of dynamic 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 [12, 28], use periodic transaction commits to support legacy code, showing that implicit atomicity can be implemented as a direct extension of hardware-based TM systems. TM systems will soon be available commercially [1].

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 completing 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, BulkSC [6], ASO [29] and Implicit Transactions [26] can all offer sequential consistency with the performance of more relaxed memory models such as release consistency [11].

Supporting coarse-grain consistency enforcement with implicit atomicity has two interesting properties, which we leverage in creating Atom-Aid. 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 implies that the effects of remote threads are only visible at chunk boundaries. Figure 3 contrasts fine with coarse-grain interleaving. In Figure 3(a), interleaving can happen in between any instructions (shown on the left side) and there are six possible interleavings (shown on the right side), whereas in Figure 3(b), interleaving opportunities only happen between chunks, and there are far fewer possible interleavings — only two in this example. The second interesting property is that the software is oblivious to the granularity of consistency enforcement, allowing the system to arbitrarily choose chunk boundaries and adjust the size of chunks dynamically without affecting program correctness or the memory semantics observed by the software.

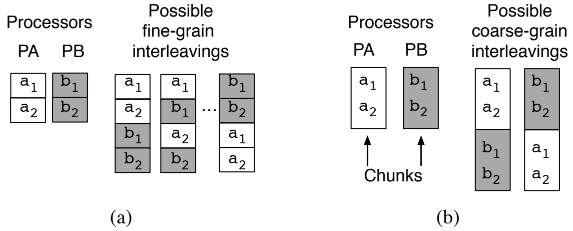


Figure 3. Fine (a) and coarse-grain (b) access interleaving. There are six possible interleavings for the fine-grained system and two possible interleavings for the coarse-grained system.

Atom-Aid can be implemented in any architecture that supports implicit atomicity or, more generally, any system that supports forming arbitrary atomic blocks from the dynamic instruction stream. However, for the purpose of illustration, in this paper we assume an underlying system similar to BulkSC [6], in which processors repeatedly execute chunks separated by checkpoints — no dynamic instruction is executed outside a chunk. We now briefly describe BulkSC, as its mechanisms naturally provide much of what an implementation of Atom-Aid needs, irrespective of the underlying system. Once again, it is important to note that Atom-Aid does not rely on BulkSC specifically.

Bulk [5] is a set of hardware mechanisms that simplify the support of common operations in environments with multiple speculative threads (or 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 [2], which are subject to aliasing. Consequently, a signature represents a superset of the original address set. The BDM also includes units that perform signature operations such as union, intersection, expansion, etc.

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 2,000 instructions). Finally, forward progress is guaranteed by reducing chunk sizes in the presence of repeated chunk aborts.

2.2. Atomicity Violations

Data-races are the most widely known concurrency defects. They occur when there are conflicting accesses from different threads to the same piece of shared data, at least one access is a write and there are no synchronizing events between the threads to prevent these simultaneous accesses. There has been significant work on tools [8, 9, 19, 24] and hardware support [20, 31] for data-race detection. However, as pointed out by Flanagan *et al.* [10], 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 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.* [10]. An atomicity violation exists in the code when the programmer makes incorrect 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 TM-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 some sequential execution of the sections assumed atomic by the programmer — one in which the interleaving does not occur — that is equivalent, and guaranteed to produce the same final state as the one produced when the interleaving occurs. Conversely, if there is an atomicity violation, the concurrent execution is not serializable — *i.e.*, there is no equivalent sequential execution that is guaranteed to produce the same final state. 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.* [16] used this analysis to determine same-variable unserializable access interleavings detected by their AVIO system. Table 1 reproduces the analysis presented in [16]. The column "Interleaving" represents the interleaving in the format $AB \leftarrow C$, where AB is the pair of local memory accesses interleaved by C, the access from a remote thread. For example, $RR \leftarrow W$ corresponds to two local read accesses interleaved by a remote write access.

Interleaving	Serializes?	Comment
R R ←R	Yes	—
R R ←W	No	Interleaved write makes two local reads inconsistent.
R W ←R	Yes	—
R W ←W	No	Local write may depend on result of read, which is overwritten by remote write before local write.
W R ←R	Yes	—
W R ←W	No	Local read does not get expected value.
W W ←R	No	Intermediate value written by first write is made visible to other threads.
W W ←W	Yes	—

Table 1. Memory interleaving serializability analysis cases (from [16]).

Terminology used in this paper. Throughout the rest of 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 (and including) the read and update to `counter` in Figure 1. 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 be observed. 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 it is possible for two memory operations assumed to be atomic by the programmer to be interleaved by another remote memory operation — *i.e.*, there is at least one opportunity for interleaving inside the atomicity-lacking section. Conversely, an atomicity violation is *hidden* if the memory operations cannot be interleaved — *i.e.*, there are no opportunities for interleaving in the atomicity-lacking section. An atomicity violation *manifests itself* if and only if it is exposed and the specific concurrent execution is unserializable (according to Table 1), which will likely lead to incorrect 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 in time but were not necessarily interleaved). Finally, we refer to *implicitly atomic systems* and *chunk granularity systems* interchangeably.

3. Implicit Atomicity Hides Atomicity Violations

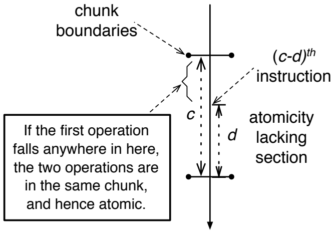
In this section, we contrast chunk granularity systems with instruction granularity systems, and show that the implicit atomicity in chunk granularity systems probabilistically reduces the chances of atomicity violations being exposed and, consequently, manifesting themselves. This happens when an atomicity-lacking section is completely enclosed within 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 [3]. On the other hand, systems that support implicit atomicity do provide forward progress guarantees [6].

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 4 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 4.


chunk boundaries
(c-d)^th instruction
If the first operation falls anywhere in here, the two operations are in the same chunk, and hence atomic.
c
d
atomicity lacking section

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

Figure 4. Atomicity-lacking section within chunk boundaries. P_{hide} is the probability that two atomicity-lacking operations will fall within 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. Also note that, in the worst case, 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 5 shows the probability of hiding an atomicity violation for various violation sizes as chunk sizes increase, according to the expression of P_{hide} shown in Figure 4. As expected, we observe that increasing chunk size increases the probability of hiding an atomicity violation, but this is subject to diminishing returns.

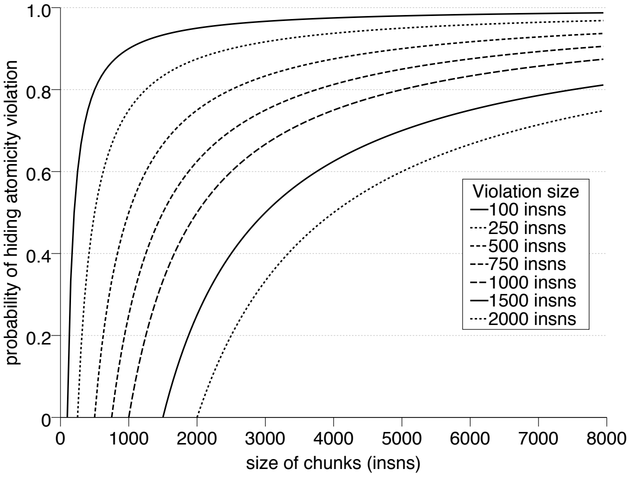


Figure 5. Probability of hiding atomicity violations of different sizes as a function of chunk size.

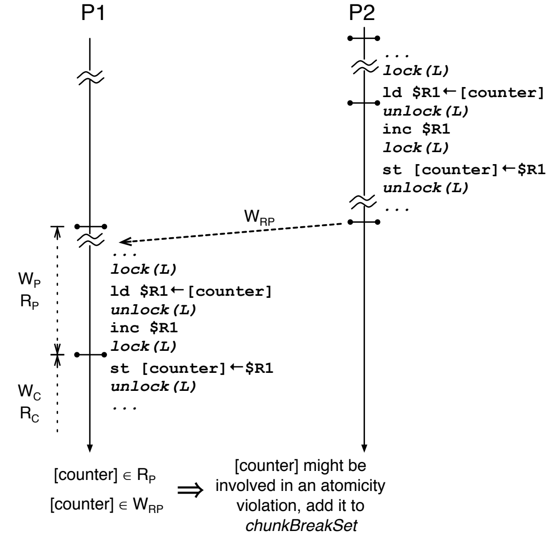
From our experiments, we observe that the typical atomicity violation ranges from 500 to 750 instructions. Assuming a chunk size of 2,000 instructions, we observe in Figure 5 that the expected probability of naturally hiding these typical atomicity violations is 63-75%.

4. Actively Hiding Atomicity Violations with Smart Chunking

In Section 3, we show 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 of exposing atomicity violations. Atom-Aid does this by detecting potential atomicity violations and inserting a chunk boundary right before the first memory access of these potential violations, hoping to enclose all memory accesses of an atomicity-lacking section inside 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.

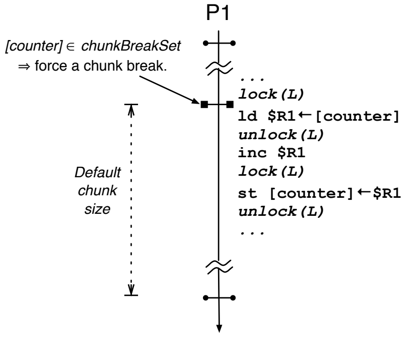
Atom-Aid detects potential atomicity violations by observing the memory accesses of each chunk and the interleaving of memory accesses from other committing chunks in the system. When Atom-Aid detects at least two nearby accesses to the same variable a by the local thread and at least one recent access to a by another thread, it looks at the types of the accesses involved 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 committed remote chunks.

Figures 6 and 7 show 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 6, 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 6 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 7), 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 the atomicity violation in Figure 6 is exposed, it does not mean it has manifested itself. Also, as will be explained in the next section, even if the atomicity violation is naturally hidden, Atom-Aid is still able to detect it. This shows that Atom-Aid is able to detect atomicity violations before they manifest themselves.


P1
P2
W_RP
W_P
R_P
W_C
R_C
[counter] ∈ R_P
[counter] ∈ W_RP
⇒
[counter] might be involved in an atomicity violation, add it to chunkBreakSet

```
...
lock(L)
ld $R1←[counter]
unlock(L)
inc $R1
lock(L)
st [counter]←$R1
unlock(L)
...
```

Figure 6. Actively hiding an atomicity violation: how Atom-Aid discovers that `counter` might be involved in an atomicity violation and adds it to the *chunkBreakSet*.



```
...
lock(L)
ld $R1←[counter]
unlock(L)
inc $R1
lock(L)
st [counter]←$R1
unlock(L)
...
```

Figure 7. Actively hiding an atomicity violation: 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

As mentioned earlier, 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 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 $RR \leftarrow W$ and $WR \leftarrow W$. To detect if either of them has happened, Atom-Aid uses the corresponding set expressions in the third column. Specifically, to identify a potential $RR \leftarrow W$ 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. Also, 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 limiting factor since Atom-Aid can not hide atomicity violations larger than a chunk.

Local Op.	Interleaving	Expression
read	R ←W R	$(a \in R_C \vee a \in R_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
	W ←W R	$(a \in W_C \vee a \in W_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
write	R ←W W	$(a \in R_C \vee a \in R_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
	W ←R W	$(a \in W_C \vee a \in W_P) \wedge$ $(a \in R_{RC} \vee a \in R_{RP})$

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 7 again. Suppose the address of variable `counter` has been previously inserted 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 multiple accesses 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 8 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.

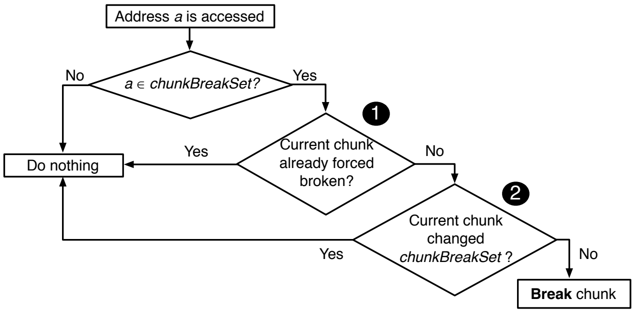


Figure 8. Chunk-breaking policy in a flowchart.

4.3. An Architecture Based on Signature Operations

We based our Atom-Aid implementation on BulkSC because its signatures offer a convenient way of storing read and write sets of chunks. 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 9 shows all signatures used by Atom-Aid. 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 is executing. 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.

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 as an extension to the cache tags. An extra bit per cache line tag indicates whether or not the corresponding line address is part of the *chunkBreakSet*. When a cache line is accessed, this bit is

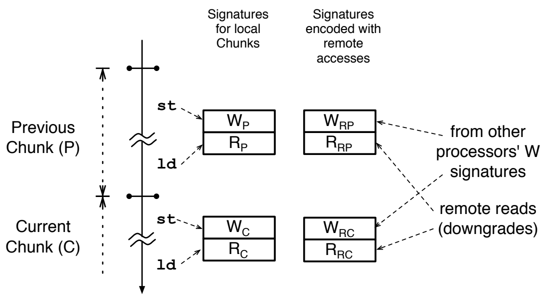

Signatures for local Chunks
Signatures encoded with remote accesses
Previous Chunk (P)
Current Chunk (C)
st
ld
st
ld
W_P
R_P
W_{RP}
R_{RP}
W_C
R_C
W_{RC}
R_{RC}
from other processors' W signatures
remote reads (downgrades)

Figure 9. Signatures used by Atom-Aid to detect likely atomicity violations.

checked to determine whether the corresponding address belongs to the *chunkBreakSet*.

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* contains 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.

As mentioned earlier, we want Atom-Aid to be useful for debugging tools 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*.

4.3.1. Implementation Discussion

While we have assumed a BulkSC-like system in this paper, other systems that support implicit atomicity can leverage Atom-Aid. Take, for example, a TCC-like [12] 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. Furthermore, with minor extensions, Atom-Aid can also be implemented in systems that do not support full-fledged implicit atomicity for coarse-grain memory ordering, as long as they support forming atomic blocks dynamically.

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 [5, 23]. 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 from all the state required by Atom-Aid, the only that would make sense to be preserved through a context switch is the *chunkBreakSet* — but it is not necessary to do so.

5. Experimental Setup

5.1. Simulation Infrastructure

We model a system that resembles BulkSC [6] using the PIN [17] dynamic binary instrumentation infrastructure. Our model includes chunk-based execution, using signatures to represent read and write sets, and the mechanisms required by Atom-Aid's algorithm. Unless otherwise noted, the signature configuration used is the same as in [5]. Since the simulator is based on binary instrumentation and runs workloads in a real multiprocessor environment, it is subject to non-determinism. For this reason, we present results averaged across a large number of runs, with error bars showing the 95% confidence interval for the average.

Our simulations need to determine whether a particular atomicity-lacking section is fully enclosed within a chunk to assess how often they are hidden. We do that by explicitly annotating the code with the beginning and end of each atomicity-lacking section. The model then checks dynamically if these markers fall within the same chunk. If so, the corresponding atomicity violation is considered hidden. It is important to note that these annotations are not used by Atom-Aid's algorithm in any way — their sole purpose is to evaluate the techniques we propose.

5.2. Simulated Workloads

For our experiments, we use two types of workloads: bug kernels and entire applications. The goal of using bug kernels is to generate extreme conditions in which atomicity-lacking sections are executed more often than in real applications. We can then use them to perform stress tests of Atom-Aid in a short amount of time. We also include entire applications (MySQL, Apache, XMMS) for a more complete evaluation. For the 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 bug kernels from real applications based on previous literature on atomicity violations [10, 16, 30]. We made sure that the atomicity violation in the original application remained intact in the kernel version. Wherever possible, we also included program elements that affect timing, such as I/O, to mimic realistic interleaving behavior in the kernel workloads.

Table 3 lists the workloads we use in our evaluation. We provide the number of threads each workload uses, the average, minimun, maximum and standard deviation values of violation sizes, along with a brief description of each bug.

We have a reasonably wide range of violation sizes, from 80 to 3,600 dynamic instructions. The violation sizes found in real

			Atomicity Violation Size			
Workload Type	Workload Name	Threads	Avg.	Std. Dev.	Min-Max	Description
kernels	Apache-extract	2	973	18.63	909-1014	Kernel version of above Apache log system bug.
	BankAccount	2	85	1.21	81-91	Shared bank account data structure bug.Simultaneous withdrawal and deposit with incorrectly synchronized program may lead to inconsistent final balance.
	BankAccount2	2	2407	1.38	2403-2411	Same as previous, with larger atomicity violation.
	CircularList	2	587	1.88	585-595	Shared work list data ordering bug. Removing, processing, and adding work units to list non-atomically may reorder work units in list.
	CircularList2	2	3593	2.92	3588-3608	Same as previous, with larger atomicity violation.
	LogProc&Sweep	5	278	1.69	272-282	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	5	2498	5.55	2489-2514	Same as previous, with larger atomicity violation.
	MySQL-extract	2	239	0.40	239-243	Kernel version of above MySQL log system bug.
	StringBuffer	2	556	0.00	556-556	java.lang.StringBuffer overflow bug [10]. On append, not all required locks are held. Another thread may change buffer during append. State becomes inconsistent.
real	Apache	25	464	0.00	464-464	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	28	722	8.37	713-736	Security backdoor in MySQL-4.0.12. While one thread closes file and sets log status to closed, other thread accesses log. Logging thread sees closed log, and discards entries.
	XMMS	6	586	6.93	572-595	Visualizer bug in XMMS-1.2.10, a media player. While visualizer is accessing PCM stream data, data in PCM can be changed, or freed, causing corruption or crash.

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

applications are as large as several hundred instructions, never exceeding one thousand instructions. We believe that the reason these atomicity violations are relatively short is because long atomicity violations are easier to find during testing, since they are bound to manifest more often. The violations found in most bug kernels are similar in size to the real applications. However, we artificially added more work to a few atomicity-lacking sections to evaluate larger violation sizes (BankAccount2, CircularList2 and LogProc&Sweep2). Note that, for Apache and MySQL, the violation sizes in the full application and the kernel versions are different. This is because in Apache-extract there was additional work in generating random log entries, and MySQL-extract does not use MySQL's custom implementation of `memcpy`.

The real applications we study use a significant number of threads, ranging from 6 to 28. Most bug kernels use only 2 threads because they are sufficient to reproduce the original atomicity violation.

For the experiments we present in Section 6, we simulate each of the bug kernels forty times for each chunk size, varying chunk size from 750 to 8,000 instructions. Due to restrictions in simulation time, we run each of the real applications five times, with a chunk size of 4,000 instructions.

6. Evaluation

Our evaluation consists of four parts. Section 6.1 shows an experimental validation of the probability study we present in Section 3.1 and verifies that implicit atomicity is indeed capable of hiding a significant number of atomicity violation instances. Section 6.2 shows that Atom-Aid further improves that number, hiding almost all atomicity violation instances. Section 6.3 presents data on Atom-Aid's dynamic behavior and includes a comparison between an implementation that uses hardware signatures and one that uses exact sets. Finally, Section 6.4 discusses how the information collected by Atom-Aid can be used to help locate bugs in the source code.

6.1. Natural Hiding

We validate the probability study in Section 3.1 by running the workloads from Table 3 in our simulator configured as a system with implicit atomicity but no Atom-Aid support. We vary chunk sizes and measure how often atomicity violation instances fall within a single chunk.

Figure 10 shows the percentage of atomicity violation instances naturally hidden for each of the bug kernels as the chunk size increases. The lines in the plot correspond to P_{hide} (see Figure 4) for the average violation size of each bug kernel shown in Table 3. Most experimental data points are very close to the lines derived from the analytical model. This verifies the accuracy of the model as well as our hypothesis that implicitly atomic systems can naturally hide atomicity violations. While Figure 10 does not include data for real applications, the left bar of each cluster in Figure 11(b) shows the natural hiding effect in real applications for a chunk size of 4,000 instructions, which also follows the analytical model.

Overall, implicit atomicity with chunk sizes as small as 4,000 dynamic instructions naturally hides 70% or more of the atomicity violation instances for nine of the twelve workloads. The remaining workloads have artificially large atomicity violations that prevent natural hiding at chunk sizes of 2,000 dynamic instructions or less, and keep the probability of natural hiding lower than for other workloads at larger chunk sizes. Intuitively, as the chunk size increases, this difference is gradually reduced.

These results show that implicit atomicity by itself can naturally hide a large fraction of atomicity violations. However, as we will show in the next section, Atom-Aid can significantly improve the hiding effect with a proactive approach.

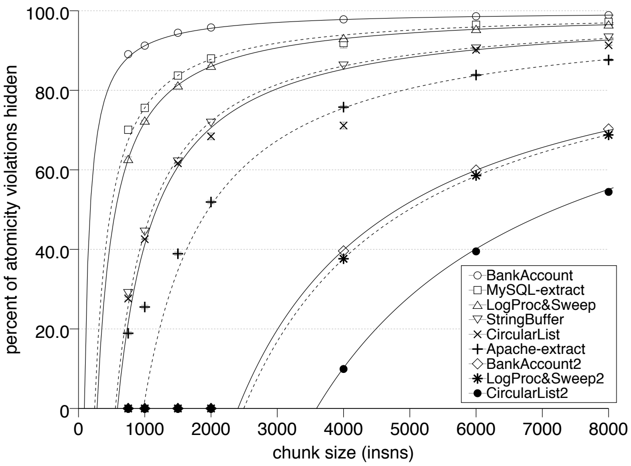


Figure 10. Experimental data on the natural hiding of atomicity violations with implicit atomicity for various chunk sizes and bug kernels. Points show empirical data, curves show data predicted by our analytical model (P_{hide}).

6.2. Active Hiding with Smart Chunking

In this section, we assess how Atom-Aid improves the hiding capabilities of implicit atomicity with smart chunking. Figure 11(a) shows the percentage of atomicity violations hidden by Atom-Aid for each bug kernel as the chunk size increases, whereas Figure 11(b) contrasts the hiding effects of Atom-Aid with plain natural hiding of implicit atomicity for real applications.

Our results show that Atom-Aid is able to hide virtually 100% of atomicity violation instances present in our benchmarks, including the real applications, with chunk sizes of only 4,000 dynamic instructions. Even with smaller chunk sizes, Atom-Aid hides the majority of atomicity violation instances. Notable exceptions are Apache-extract and the three bug kernels with artificially larger atomicity violations. As explained in Section 6.1, these larger atomicity violations cannot be hidden by smaller chunks. Apache-extract suffers from early chunk breaks, which decrease the chance of hiding atomicity violations when smaller chunk sizes are used. However, the problem disappears when chunk sizes reach 4,000 dynamic instructions because chunks become large enough to enclose both the access that caused an early break and the actual atomicity violation in its entirety, and thus hide it completely.

Overall, Atom-Aid's smart chunking algorithm is capable of hiding a much higher percentage of atomicity violation instances than just natural hiding. Moreover, Atom-Aid reduces the number of exposed atomicity violation instances by *several orders of magnitude* when compared to current commercial systems — i.e. hides more than 99% of atomicity violation instances in virtually all workloads.

6.3. Characterization and Sensitivity

Table 4 characterizes Atom-Aid's behavior by providing data collected from each of the bug kernels. We only use kernels in this study, as opposed to real applications, because they provide a more controlled environment for our measurements and they run faster.

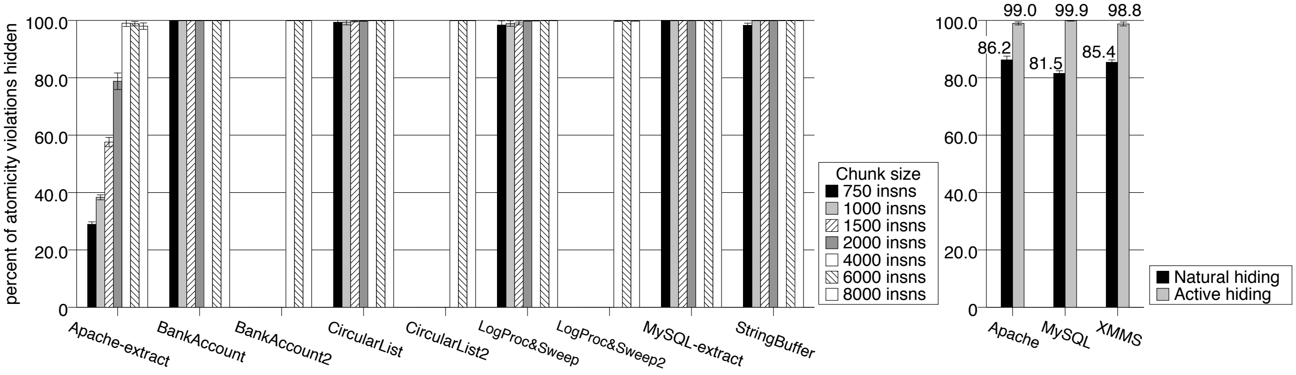
Columns 2 and 3 reproduce data from Figures 10 and 11, respectively. They show the percentage of hidden atomicity violations with natural hiding and smart chunking for a chunk size of 4,000 dynamic instructions. Again, while about 67% of atomicity violations are hidden naturally on average, smart chunking is able to hide virtually 100% of them.

Column 4 (*% Smart Chunks*) shows what fraction of chunks are created by the smart chunking algorithm as the program executes, while Column 5 (*% Unnecessary Smart Chunks*) shows what percentage of these additional chunks does not help hide atomicity violations. *% Unnecessary Smart Chunks* is large for some workloads, showing that Atom-Aid may often create chunks unnecessarily. However, *% Smart Chunks* is typically low, so even if it creates many unnecessary chunks, Atom-Aid still adds only a small fraction of all chunks. This implies Atom-Aid is unlikely to have noticeable impact on performance [6].

Columns 6 and 7 illustrate the behavior of Atom-Aid's atomicity violation detection algorithm. Column 6 (*chunkBreakSet Size*) shows how many distinct data addresses, at a line granularity, are identified as involved in a potential atomicity violation. Atom-Aid's algorithm selects, on average, only four data items as being potentially involved in an atomicity violation. Column 7 (*# Break PCs*) shows how many distinct static memory operations in the code caused Atom-Aid's smart chunking algorithm to break a chunk. On average, it breaks chunks in only three places in the program. These results show that Atom-Aid is quite selective at identifying data addresses and points in the program that are potentially involved in atomicity violations. These can be reported to a programmer who in turn has reasonably precise information about the potential atomicity violation and can use it to debug the application. We further explore this aspect of Atom-Aid in Section 6.4.

So far, we have discussed data on an implementation of Atom-Aid that exclusively uses hardware signatures for disambiguating chunks, detecting chunk interleaving and maintaining the *chunkBreakSet*. *Exact Atom-Aid* corresponds to the behavior of a non-signature based implementation of Atom-Aid. For that, all signatures in the design presented in Section 4.3 are simulated ideally as unlimited size exact sets — 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. We present these results in the group of columns entitled *Exact Atom-Aid* in Table 4. The behavior of the exact implementation of Atom-Aid would be similar to the behavior of an implementation that uses cache tag extensions as a way of keeping the sets of addresses (see Section 4.3).

First and most important, *% Hidden* for the exact implementation (Column 8) is practically the same as *% Hidden* for the signature-based implementation (Column 3), showing that the impact of signature impreciseness on the effectiveness of Atom-Aid is negligible. As expected, *% Smart Chunks* (Columns 4 and 9) is, on average, higher for the signature-based Atom-Aid, since aliasing in signatures causes chunks to be broken more frequently. However, the difference is small. The same phenomenon is also reflected in the percentage of unnecessary smart chunks (Columns 5 and 10), which is significantly lower for *Exact Atom-Aid*. As noted previously, however, this has negligible impact on performance.



(a) Bug kernels.

(b) Real applications. Chunks of 4,000 instructions.

Figure 11. Average percentage of atomicity violations hidden by Atom-Aid. Error bars show the 95% confidence interval.

	Natural	Signature-Based Atom-Aid					Exact Atom-Aid		
			% Smart	% Unnecessary	*chunkBreakSet*	# Break		% Smart	% Unnecessary
Bug Benchmark	% Hidden	% Hidden	Chunks	Smart Chunks	Size	PCs	% Hidden	Chunks	Smart Chunks
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 4. Characterization of Atom-Aid for both the signature and non-signature implementations.

6.4. Debuggability Discussion

Showing that Atom-Aid is able to hide almost all atomicity violation instances demonstrates that the algorithm inserts chunk boundaries in the appropriate 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 outside the scope of this paper, we were able to use the feedback from Atom-Aid to locate the code for the bugs in MySQL and Apache used in past work on bug detection [16, 30], 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 small number of points in the code.

Table 5 shows some data on our experience of finding atomicity bugs in real applications. The first group of columns (*Program Totals*) shows the total number of files, functions, and lines of code for the entire application. The second group (*Chunk Break Points*) shows the number of files, functions and lines of code for which Atom-Aid broke chunks while the application executed. The third group (*# of Inspections*) shows the number of files, functions and lines of code we had to inspect before we located a bug.

For Apache, only 85 lines of code in 6 files need to be inspected to locate the bug. For MySQL, this number is larger (more than 300), but MySQL has a larger code base, with almost 400,000 lines of code. We identified a bug in XMMS that was not previously known[1] after inspecting only 9 lines of code. Overall, the information provided by Atom-Aid is useful in directing the programmer's attention to the right region of code, even if using a simple heuristic like the one we present here. However, we feel that more sophisticated techniques could result in even more effective methods.

	Program Totals			Chunk Break Points			# of Inspections		
	Files	Func.	Lines	Files	Func.	Lines	Files	Func.	Lines
Apache	729	3361	290k	52	206	956	6	8	85
MySQL	871	15231	394k	44	228	681	27	84	353
XMMS	268	1368	81k	7	23	42	2	4	9

Table 5. Characterization 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 significant amount of work on concurrency bug detection, survival is not widely discussed.

[1] The XMMS project leads were contacted regarding the bug. However, no feedback was received by the time the final version of this paper was submitted.

The most relevant prior work on hardware support for atomicity violation detection is AVIO [16]. 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. Atom-Aid monitors interleavings differently, by leveraging hardware signatures. In addition, Atom-Aid monitors potential violations (ones that might not have necessarily happened), does not distinguish training from detection, and leverages implicit atomicity to survive concurrency bugs.

Serializability Violation Detection (SVD) [30] uses a heuristic to infer potential critical sections based on data and control dependences with the goal of determining if they are unserializable. In [30], the authors briefly mention that their algorithm could possibly be implemented in hardware and envision bug avoidance via global checkpoint and restart [21, 25]. Atom-Aid, like SVD, attempts to infer critical section boundaries dynamically. However, Atom-Aid uses memory access history and interleaving to detect potential violations and its bug avoidance relies only on dynamically making the potential violation atomic, not requiring global checkpoint and restart.

ReEnact [20] 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 [12, 14, 18, 22], as well as languages with new constructs for atomicity [4, 7, 13, 27]. 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 kernels of known bugs from the literature and full applications such as MySQL, Apache, and XMMS, we show that Atom-Aid reduces the chance that an atomicity violation will lead to wrong program behavior by several orders of magnitude, 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 to and detectability of concurrency bugs.

Acknowledgments

We thank the anonymous reviewers for their very useful comments. We thank Jim Larus and Shaz Qadeer from Microsoft Research for their feedback on the initial idea. We also thank Mark Oskin, Susan Eggers, Tayfun Elmas, Martha Mercaldi Kim, Lucas Kreger-Stickles and Andrew Putnam from the University of Washington for their invaluable feedback on the manuscript. Finally, we thank David Schlosser, Rich Witek and Alan Lee from AMD for their feedback and support. Brandon Lucia was supported by the Clairmont L. Egtvedt Fellowship and The Faithful Steward Endowed Fellowship.

References

[1] Sun slots transactional memory into Rock. http://www.theregister.co.uk/2007/08/21/sun_transactional_memory_rock/.

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

[3] C. Blundell, E. Lewis, and M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[4] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The ATOMOS Transactional Programming Language. In *Conference on Programming Language Design and Implementation*, 2006.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, 2006.

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

[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[8] A. Dinning and E. Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Symposium on Principles and Practices of Parallel Programming*, 1990.

[9] D. Engler and K. Ahscraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Symposium on Operating Systems Principles*, 2003.

[10] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Conference on Programming Language Design and Implementation*, 2003.

[11] 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*, 1990.

[12] 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*, 2004.

[13] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

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

[15] 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*, 2008.

[16] 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*, 2006.

[17] 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*, 2005.

[18] 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*, 2006.

[19] R. Netzer and B. Miller. Improving the Accuracy of Data Race Detection. In *Symposium on Principles and Practices of Parallel Programming*, 1991.

[20] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *International Symposium on Computer Architecture*, 2003.

[21] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, 2002.

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

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *International Symposium on Microarchitecture*, 2007.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems*, November 1997.

[25] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, 2002.

[26] 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*, 2005.

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

[28] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A Practical FPGA-based Framework for Novel CMP Research. In *International Symposium on Field-Programmable Gate Arrays*, 2007.

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

[30] 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*, 2005.

[31] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*, 2007.