ATOM-AID: DETECTING AND SURVIVING ATOMICITY VIOLATIONS

HARDWARE CAN PLAY A SIGNIFICANT ROLE IN IMPROVING RELIABILITY OF MULTITHREADED SOFTWARE. RECENT ARCHITECTURAL PROPOSALS ARBITRARILY GROUP CONSECUTIVE DYNAMIC MEMORY OPERATIONS INTO ATOMIC BLOCKS TO ENFORCE COARSE-GRAINED MEMORY ORDERING, PROVIDING IMPLICIT ATOMICITY. THE AUTHORS OF THIS ARTICLE OBSERVE THAT IMPLICIT ATOMICITY PROBABILISTICALLY HIDES ATOMICITY VIOLATIONS BY REDUCING THE NUMBER OF INTERLEAVING OPPORTUNITIES BETWEEN MEMORY OPERATIONS. THEY PROPOSE ATOM-AID, WHICH CREATES IMPLICIT ATOMIC BLOCKS INTELLIGENTLY INSTEAD OF ARBITRARILY, DRAMATICALLY REDUCING THE PROBABILITY THAT ATOMICITY VIOLATIONS WILL MANIFEST THEMSELVES.

Brandon Lucia
Joseph Devietti
Luis Ceze
University of Washington
Karin Strauss
AMD and
University of Washington

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 must make sure shared data is kept consistent. This usually involves specifying critical sections using locks or transactions. However, this practice is typically error-prone and can lead to synchronization defects, such as data races, deadlocks, and atomicity violations.

Atomicity violations are challenging concurrency errors. They occur 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,[1] atomicity violations account for about two-thirds of all examined nondeadlock concurrency bugs. Finding these bugs is difficult because of their subtle nature and the nondeterminism in multithreaded execution. Hence, no one can afford to assume code will be free of bugs, so it's important to both detect and prevent their manifestation. Interestingly, the manifestation of concurrency bugs is considerably influenced by how multithreaded programs are executed, which determines the global interleaving of memory operations. For a given set of memory semantics exposed to the software, multiple valid global interleavings of memory operations exist. The system can 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. Such systems provide what

0272-1732/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

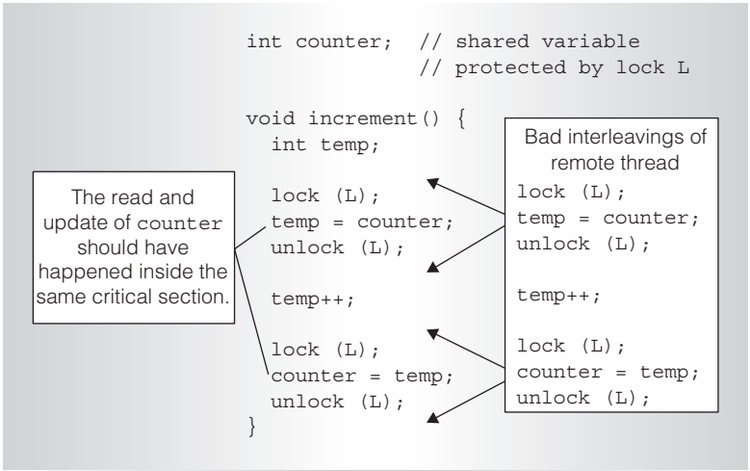

Figure 1. Simple example of an atomicity violation. The read and update of the `counter` from two threads could interleave such that `counter` is incremented only once.

we call *implicit atomicity*: 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, because interleaving only occurs at coarse granularity. Such proposals aim to support stricter memory consistency models with high performance by enforcing consistency at a coarse grain.

In this article, we show that implicit atomicity can hide atomicity violations if their memory operations fall within a chunk's boundaries. We support this observation through probability analysis and empirical evidence. Building on this observation, we propose Atom-Aid, a hardware-supported mechanism to detect and dynamically avoid atomicity violations on behalf of software without requiring any program annotation. The Atom-Aid architecture uses hardware signatures to detect likely atomicity violations and avoids them by dynamically adjusting chunk boundaries. To the best of our knowledge, this is the first article on surviving atomicity violations without requiring global checkpointing and recovery.[2] To allow programmers to find atomicity violations, Atom-Aid reports where atomicity violations could exist in the code, aiding in the debugging process. Finally, through an evaluation using buggy code from real applications, we show that Atom-Aid can reduce, by several orders of magnitude, the chances that an atomicity violation will lead to incorrect program behavior.

Atomicity violations

Data races are the most widely known concurrency defects, and there has been significant work on tools[3] and hardware support[4] for data-race detection. However, as Flanagan et al. point out,[5] data-race freedom doesn't imply that a concurrent program is correct, because the program could still suffer from atomicity violations.

Figure 1 shows a simple example of an atomicity violation. In this 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, because a call to `increment()` from another thread could be interleaved between the read and update of `counter`, leading to incorrect behavior. For example, two concurrent calls to `increment()` could cause `counter` to be incremented only once. This is a subtle error, and an easy mistake to make, because determining when atomicity is necessary and which operations must be grouped atomically can be difficult and prone to misconceptions.

The code snippet in Figure 1 doesn't 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, because both the read and update of `counter` should have been atomic to avoid unwanted interleaving of accesses to `counter` from other threads. An atomicity violation exists in the code when a programmer makes incorrect assumptions about atomicity and fails to enclose accesses that should have been performed atomically inside the same critical section. Atomicity violations can also exist in programs that use transactional-memory-based synchronization[6] instead of locks. The programmer could fail to enclose memory accesses that should be performed atomically in the same transaction.

Table 1. Memory interleaving serializability analysis cases.[7]

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

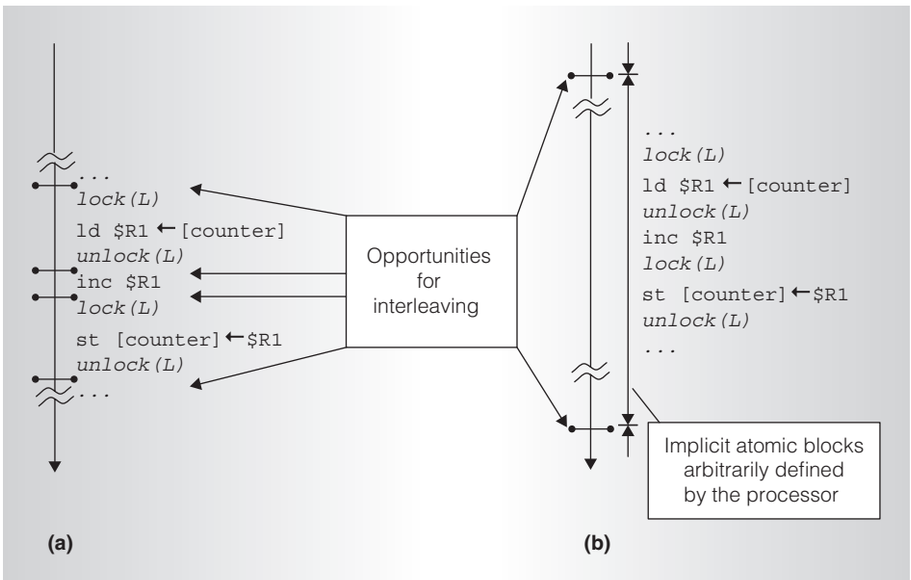

Figure 2. Opportunities for interleaving in traditional systems (a) and in systems that provide implicit atomicity (b).

Atomicity violations lead to incorrect program behavior if there is an interleaving of memory accesses from different threads that breaks the assumptions the programmer makes about atomicity. The chance of an atomicity violation manifesting itself depends on the chance of such an unfortunate interleaving. Figure 2a shows four opportunities in which interleavings can occur in traditional fine-grained systems. In contrast, Figure 2b shows where interleavings can occur when the atomicity violation's memory operations are inside the same chunk in coarse-grained systems. In such cases, the atomicity violation is hidden.

Serializability is a property of concurrent program executions. An execution in which code assumed atomic by a programmer is interleaved by code in another thread is serializable if there is a noninterleaved execution that produces the same resulting state as the interleaved execution. Thus, if an atomicity violation manifests itself in an execution, the execution is not serializable. 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 system (see Table 1).[7]

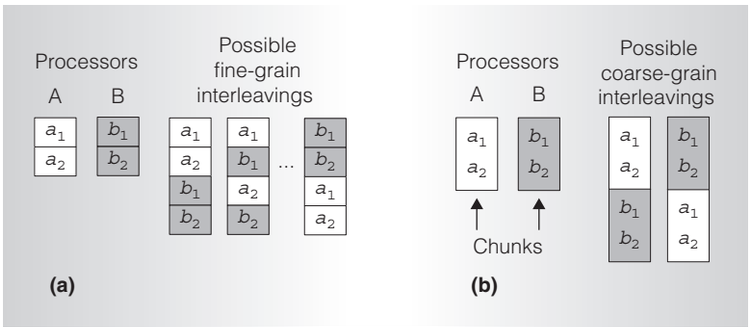

Figure 3. Fine-grained (a) and coarse-grained (b) access interleaving. There are six possible interleavings for the fine-grained system and two possible interleavings for the coarse-grained system.

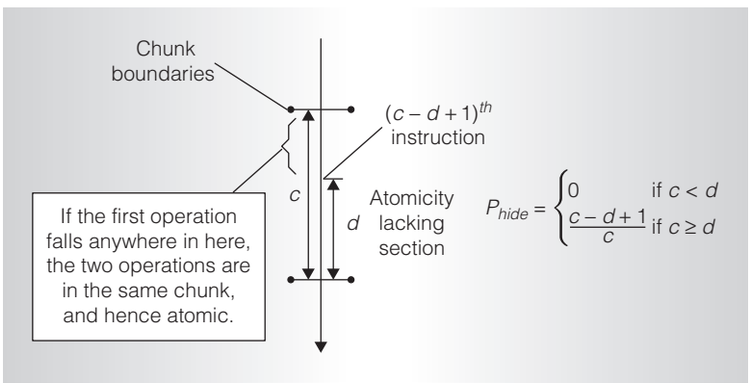

Figure 4. Atomicity violation within chunk boundaries. (P_{hide} is the probability that the violation will execute entirely in one chunk.)

The first column represents the interleaving in the format AB ← C, where AB is the pair of local memory accesses interleaved by C, the access from a remote thread. For example, RR ← W corresponds to two local read accesses interleaved by a remote write access. In this work, we leverage this analysis, considering unserializable interleavings of shared-memory accesses as indicators of potential violations of the atomicity of code that a programmer has assumed to be atomic.

Implicit atomicity

Several recent proposals (for example, BulkSC,[8] atomic sequence ordering (ASO),[9] and implicit transactions[10]) describe systems that support implicit atomicity. In such systems, memory operations in the dynamic instruction stream are arbitrarily grouped into atomic chunks. This allows coarse-grained hardware memory consistency enforcement, as opposed to instruction granularity enforcement. Coarse-grained consistency enforcement lets a system support sequential consistency with the performance of more relaxed memory models such as release consistency.

Implicit atomicity has two interesting properties, which we leverage in Atom-Aid. First, the amount of interleaving of memory operations from different processors is reduced because interleaving occurs only at chunk boundaries. As a result, the effects of remote threads are visible only at these chunk boundaries. Figure 3 contrasts fine- and coarse-grained interleaving. In Figure 3a, interleaving can occur between any instructions, and there are six possible interleavings. In Figure 3b, interleaving opportunities exist only between chunks, and there are far fewer possible interleavings—only two in this example. Second, the software is oblivious to the granularity of the chunks used, letting the system arbitrarily choose chunk boundaries and adjust the size of chunks dynamically without affecting program semantics.

Atom-Aid can be implemented in any architecture that supports forming arbitrary atomic blocks from the dynamic-instruction stream. In this article, we assume a system similar to BulkSC,[8] in which processors repeatedly execute chunks separated by local checkpoints, and no dynamic instruction is executed outside an atomic chunk.

Implicit atomicity naturally hides atomicity violations

Chunk granularity systems naturally hide atomicity violations by reducing the probability that code assumed atomic is actually interleaved. This is because, by chance, the violations can execute completely inside one atomic chunk.

Figure 4 illustrates how we derive the probability that an atomicity violation will execute inside a chunk. Let c be the default size of a chunk in dynamic instructions, and d be the size of the atomicity violation in dynamic instructions. Let P_{hide} be the probability that the violation executes inside a single chunk—that is, the probability of hiding the atomicity violation. If the first operation of the violation is within a chunk's first $(c - d + 1)$ instructions, the first and last

operations in the violation will fall in the same chunk and will be committed atomically, hiding the atomicity violation. We can then express the probability of hiding an atomicity violation as shown in Figure 4.

In this model, an *instruction granularity system* is one with a chunk size equal to one instruction ($c = 1$) and, hence, where $P_{hide} = 0$, because the size of an atomicity violation is at least two instructions ($d \geq 2$). This is consistent with the observation that an instruction granularity system can't hide atomicity violations. Also, in the worst case, atomicity violations are bigger than the chunks in chunk-based systems, which prevents any violation from being hidden ($P_{hide} = 0$). As a result, chunk granularity systems can hide atomicity violations but never increase the chances of them manifesting themselves.

We validated this model with empirical data. Figure 5 shows the percentage of atomicity violation instances naturally hidden for bug kernels as the chunk size increases. Most experimental data points are very close to the lines derived from the analytical model. This verifies the model's accuracy as well as our hypothesis that implicit atomicity naturally hides atomicity violations.

Figure 5 shows that a chunk size of 2,000 instructions yields a probability of naturally hiding atomicity violations that ranges from 53 to 95 percent, excluding BankAccount2, LogProc&Sweep2, and CircularList2, whose atomicity violations are longer than 2,000 instructions.

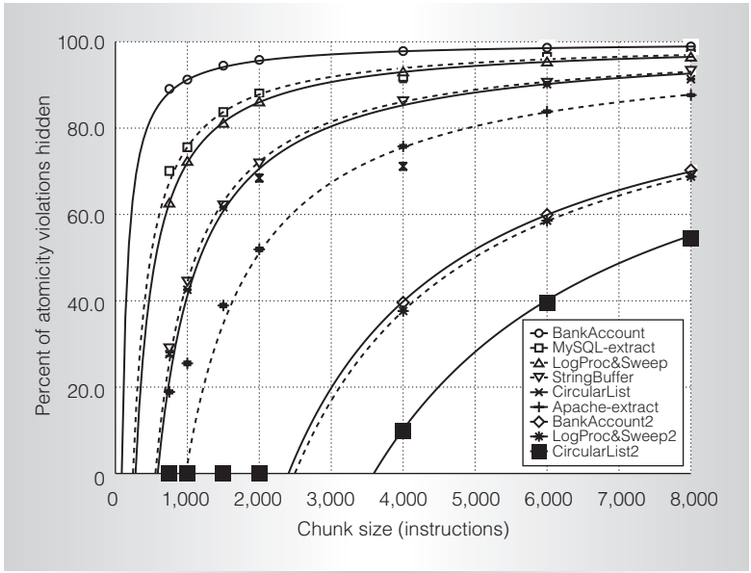

Figure 5. Experimental data on the natural hiding of atomicity violations with implicit atomicity for various chunk sizes and bug kernels. Points show empirical data; curves show data predicted by our analytical model (P_{hide}).

Actively hiding atomicity violations with smart chunking

Atom-Aid uses a technique called *smart chunking* to automatically determine where to place chunk boundaries, to further reduce the probability of exposing atomicity violations. Atom-Aid does this by detecting potential atomicity violations and inserting a chunk boundary just before the first memory access of these potential violations, thus attempting to enclose all memory accesses of an atomicity violation inside the same chunk. In essence, Atom-Aid infers where critical sections should be in the dynamic-instruction stream and inserts chunk boundaries accordingly. This process is transparent to software and 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 examines the types of accesses involved and determines whether these accesses are potentially unserializable, according to Table 1. If so, Atom-Aid starts monitoring accesses to *a*. When the local thread accesses *a* again, Atom-Aid inserts a chunk boundary. 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.

Figure 6 shows how the idea is applied to the counter increment example given in Figure 1, assuming BulkSC provides implicit atomicity and uses hardware signatures to efficiently keep read and write sets.[8] Atom-Aid also maintains the read and write sets of the previously committed chunk, R_P and W_P. Recall that, in BulkSC, processors committing

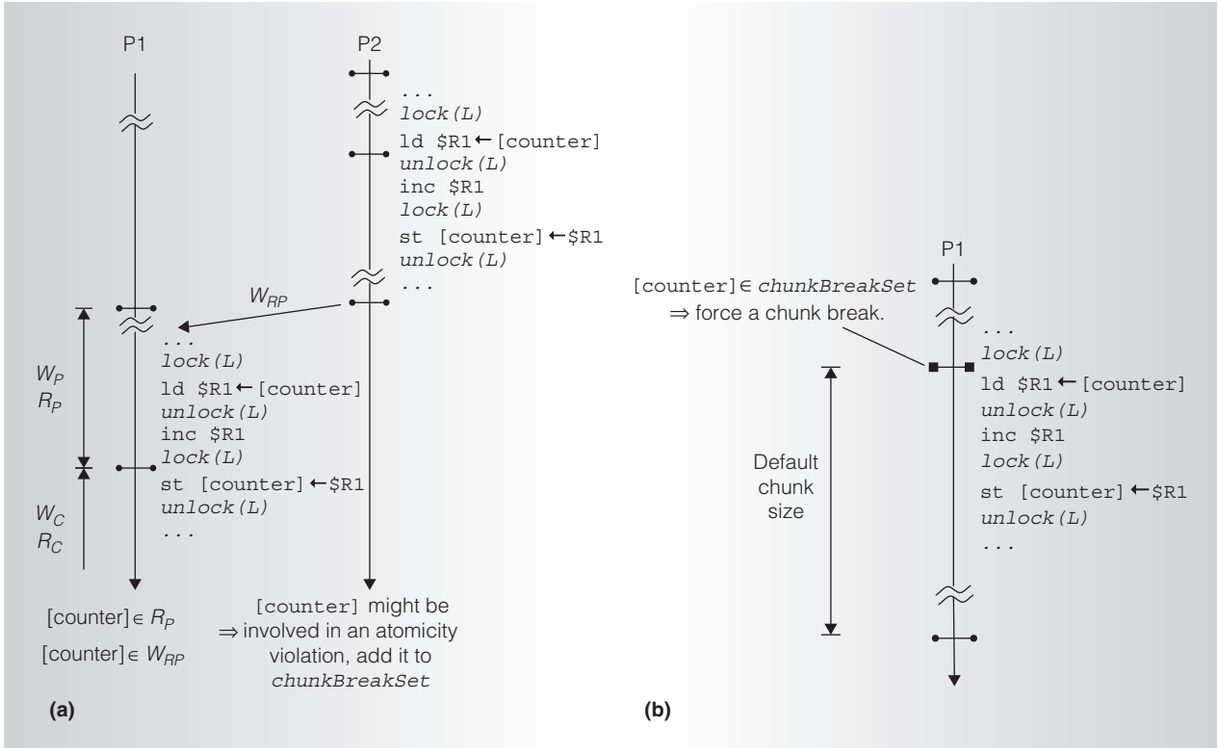

Figure 6. Actively hiding an atomicity violation. Atom-Aid discovers that `counter` might be involved in an atomicity violation and adds it to the `chunkBreakSet` (a). When `counter` is accessed, a chunk boundary is automatically inserted because it belongs to the `chunkBreakSet` (b).

chunks send their write sets to other processors in the system, allowing Atom-Aid to learn what was written recently by remotely committing chunks (W_{RP}). In Figure 6a, processors P1 and P2 are both executing `increment()`. Although there is a chance that the read and update of `counter` will be atomic due to natural hiding, this isn't the case in Figure 6a. In the example, the read of the `counter` is inside the previously committed chunk (P), whereas 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, when P1 accesses `counter` again (Figure 6b), 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. Although the atomicity violation in Figure 6a is exposed, it doesn't mean it has manifested itself, because the interleaving might not actually have occurred. Also, even if the atomicity violation is naturally hidden, Atom-Aid can still detect it. This shows that Atom-Aid can detect atomicity violations before they manifest themselves.

Detecting likely atomicity violations

The goal of Atom-Aid is to detect potential atomicity violations before they occur. 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're potentially unserializable. If they are,

Atom-Aid treats them as a potential atomicity violation. Atom-Aid needs to keep track of three pieces of information, as shown in Figure 7:

- the type t (read or write) and address a of the memory operation currently executing;
- the read and write sets (as signatures) of the current and previously committed chunk, R_C, W_C, R_P, and W_P; and
- the read and write sets of chunks committed by remote processors while the previously committed chunk was executing (R_{RP} and W_{RP}), along with read and write sets of chunks committed by other processors while the current chunk is executing (R_{RC} and W_{RC}).

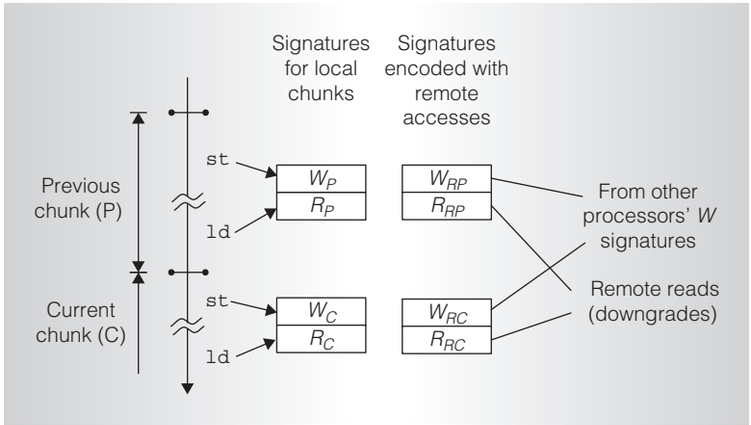

Figure 7. Signatures used by Atom-Aid to detect likely atomicity violations.

Table 2 shows how Atom-Aid uses this information to determine whether these accesses constitute a potential atomicity violation. 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 these interleavings. For example, consider the first two cases: When the local memory access is a read, the two possible nonserializable interleavings are RR ← W and WR ← W. To detect whether either of them has occurred, Atom-Aid uses the corresponding set of expressions in the third column. Specifically, to identify a potential RR ← W interleaving, Atom-Aid first checks whether a is in any of the local read sets ($a \in R_C \vee a \in R_P$). If so, Atom-Aid 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`. This case is not necessarily an atomicity violation, because the remote write might not have actually interleaved between the two reads. Also, because Atom-Aid keeps only two chunks' worth of history, it is capable of detecting only atomicity violations that are shorter than the size of two chunks. But this isn't a limiting factor, because Atom-Aid can't hide atomicity violations larger than a chunk. Because Atom-Aid uses signatures to keep read and write sets, it is very simple to evaluate the necessary set expressions.

Adjusting chunk boundaries

After an address is added to the processor's `chunkBreakSet`, every access to this address by the local thread triggers the Atom-Aid chunk-breaking mechanism. If Atom-Aid placed a chunk boundary immediately before all accesses that triggered it, Atom-Aid wouldn't actually prevent any atomicity violation from being exposed. To see why, consider Figure 6b again. Suppose the address of variable `counter` had been previously inserted in the `chunkBreakSet`. When the load from `counter` executed, it would trigger Atom-Aid, which could then place a chunk boundary immediately before this access. When the store to `counter` executed, it would trigger 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,

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

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

but not before every single access. Another example is when an address has just been added to the `chunkBreakSet`. Most likely, the local thread is still manipulating the corresponding variable, in which case avoiding a chunk break can actually be beneficial. Atom-Aid avoids these undesirable chunk breaks by detecting the conditions just described (see the full version of this article for more details[11]).

Evaluation

We modeled a system that resembles BulkSC using the Pin dynamic binary instrumentation infrastructure.[12] We also modeled chunk-based execution, signature-based read and write sets, and the mechanisms required by Atom-Aid's algorithm. Unless noted, the signature configuration used is the same as that used by Ceze et al.[13] We ran experiments on a real multiprocessor, so executions were subject to nondeterminism. For this reason, we show results averaged across many runs, with error bars indicating a 95 percent confidence interval for the average.

To check whether an atomicity violation has been hidden, we annotated each one and checked whether these markers fell within the same chunk at runtime. These markers are invisible to Atom-Aid; their sole purpose is to evaluate Atom-Aid's bug-hiding capability.

We experimented with two types of workloads: bug kernels and full applications. Bug kernels let us experiment under stress-test conditions in which buggy code was executed more often than in full applications. We also included three full applications: MySQL, Apache, and XMMS (X Multimedia System) in our evaluation. To test MySQL, we used the SQL-bench test-insert workload. We tested Apache using ApacheBench. To test XMMS, we played a media file with the visualizer on.

We created bug kernels from applications used previously in the literature.[2,5,7] In the kernels, we carefully preserved the bug from the original application and, when possible, included timing-sensitive events such as I/O operations to mimic realistic behavior.

Violations in our benchmarks ranged widely, from about 80 to approximately 3,600 dynamic instructions. In the full-application workloads, violations were as large as several hundred instructions, but never exceeded 1,000 instructions. The reason that these violations are relatively short is that long violations are easier to find in testing because they likely manifest themselves more often. For the bug-kernel experiments, we varied the size of chunks from 750 to 8,000 instructions. For the full-application experiments, we used 4,000 instruction chunks.

Active hiding with smart chunking

Figure 8a shows that Atom-Aid hid nearly 100 percent of violations for almost all chunk sizes, with a few exceptions—namely Apache-extract and three bug kernels with artificially enlarged atomicity violations: BankAccount2, CircularList2, and LogProc&Sweep2. The artificially large violations were too large to execute within small chunks; hence, they couldn't be hidden, except by larger chunks. Apache-extract suffered from early chunk breaks, decreasing Atom-Aid's ability to hide violations with smaller chunks. When the chunk size reached 4,000 instructions, however, chunks were large enough to enclose both the access that caused the early break and the violation, ensuring survival.

Figure 8b contrasts the hiding effects of Atom-Aid with those of implicit atomicity alone, in full applications. Our results show

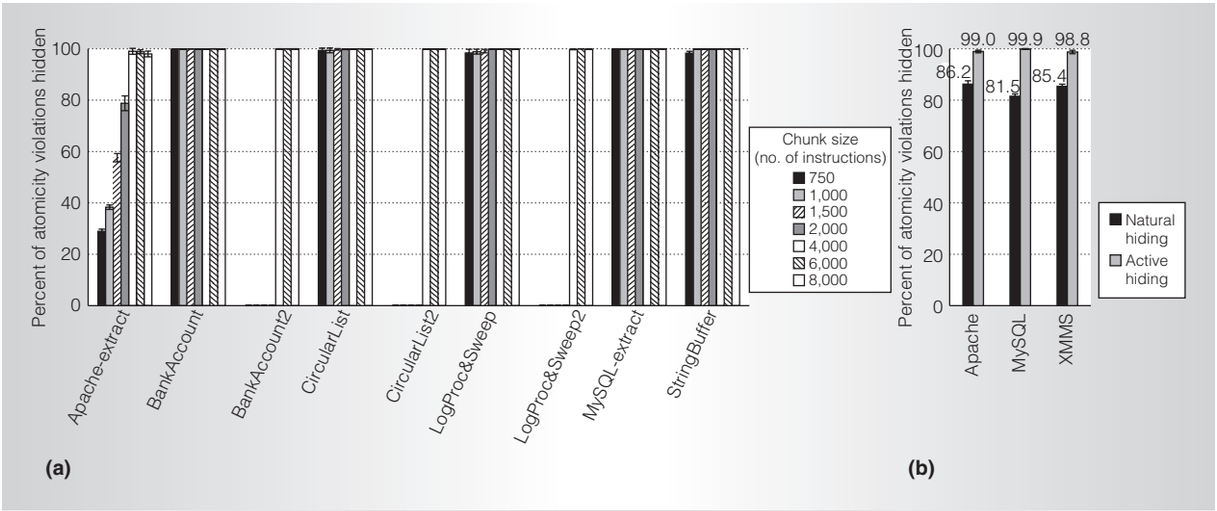

Figure 8. Average percentage of atomicity violations hidden by Atom-Aid: bug kernels (a) and real applications (chunks of 4,000 instructions) (b). (Error bars show the 95 percent confidence interval.)

that Atom-Aid can hide virtually 100 percent of atomicity violations in our full-application benchmarks with chunks of just 4,000 instructions. Figure 8 clearly shows that Atom-Aid's smart chunking algorithm can hide a far higher percentage of atomicity violations than implicit atomicity alone. Moreover, Atom-Aid reduces the number of exposed atomicity violations by several orders of magnitude when compared to current commercial systems. In fact, it hides more than 99 percent of atomicity violations.

Debugging with Atom-Aid

In addition to helping software survive atomicity violations, Atom-Aid also reports the program counter (PC) of the memory instruction at which a data address was added to the `chunkBreakSet`. At such points in the execution Atom-Aid detected a potentially unserializable interleaving. These points could help a developer locate bugs in code.

We simply sorted the list of code points at which Atom-Aid detected a potentially unserializable interleaving according to the frequency with which each PC appeared in the list. We were able to locate buggy code in MySQL and Apache, which have been used in prior bug detection research[2,7] and even to detect a new bug in XMMS.

For Apache, only 85 lines of code in six files needed to be inspected. For MySQL, this number was larger (more than 300), but MySQL has a very large code base, with nearly 400,000 lines of code. We identified a bug in XMMS that was not previously known after inspecting only nine lines of code. Overall, the information provided by Atom-Aid is useful in directing the programmer's attention to the correct region of code, even when using a simple heuristic like the one we present here. More sophisticated techniques could result in an even more efficient debugging methodology.

Other work related to Atom-Aid

There is significant work in concurrency bug detection, but survival is not widely addressed. AVIO,[7] the most relevant prior work in hardware atomicity violation detection, extracts interleaving invariants in training and checks whether they hold in future runs. AVIO monitors interleavings using cache extensions. Atom-Aid is different in that it uses hardware signatures, monitors potential violations (ones that might not have necessarily occurred), and leverages implicit atomicity to detect and help software survive concurrency bugs.

ReEnact is another hardware proposal that targets concurrency bugs.[4] However, it

focuses only on data races, not atomicity violations. ReEnact attempts to dynamically repair data races on the basis of a library of race patterns.

Serializability violation detection (SVD) heuristically infers critical sections based on data and control dependences and attempts to determine their serializability.[2] The authors suggest that their algorithm could be implemented in hardware, and could avoid bugs via backwards error recovery (BER) using global checkpointing and restart. Atom-Aid, like SVD, attempts to infer critical sections dynamically. Atom-Aid, however, uses memory interleavings to detect potential violations. Moreover, unlike BER, Atom-Aid's bug avoidance doesn't require global checkpointing and restart.

Atom-Aid, building on the observation that implicit atomicity can naturally hide some atomicity violations, proactively chooses chunk boundaries to avoid exposing the violations, without requiring any special program annotations or global checkpointing mechanism. Achieving the same effect of implicit atomicity statically, by having a compiler automatically insert arbitrary transactions in a program, would be challenging because it could hurt performance or even prevent forward progress. On the other hand, systems that support implicit atomicity facilitate forward progress.[8] Atom-Aid is a meaningful step toward a system that offers both resilience to and detectability of concurrency bugs. MICRO

References

1. S. Lu et al., ''Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics,'' *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 08), ACM Press, 2008, pp. 329-339.
2. M. Xu, R. Bodik, and M.D. Hill, ''A Serializability Violation Detector for Shared-Memory Server Programs,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 05), 2005, pp. 1-14.
3. S. Savage et al., ''Eraser: A Dynamic Data Race Detector for Multi-threaded Programs,'' *ACM Trans. Computer Systems,* vol. 30, no. 4, Nov. 1997, article no. 20.
4. M. Prvulovic and J. Torrellas, ''ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multi-threaded Codes,'' *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), ACM Press, 2003. pp. 110-121.
5. C. Flanagan and S. Qadeer, ''A Type and Effect System for Atomicity,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 338-349.
6. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), IEEE CS Press, 1993, pp. 289-300.
7. S. Lu et al., ''AVIO: Detecting Atomicity Violations via Access Interleaving Invariants,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 37-48.
8. L. Ceze et al., ''BulkSC: Bulk Enforcement of Sequential Consistency,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 278-289.
9. T.F. Wenisch et al., ''Mechanisms for Store-Wait-Free Multiprocessors,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 266-277.
10. E. Vallejo et al., ''Implementing Kilo-instruction Multiprocessors,'' *Proc. Int'l Conf. Pervasive Services* (ICPS 05), IEEE CS Press, 2005, pp. 325-336.
11. B. Lucia et al., ''Atom-Aid: Detecting and Surviving Atomicity Violations,'' *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 35), IEEE CS Press, 2008, pp. 277-288.
12. C.K. Luk et al., ''Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 190-200.
13. L. Ceze et al., ''Bulk Disambiguation of Speculative Threads in Multiprocessors,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture* (ISCA 06), ACM Press, 2006, pp. 227-238.

Brandon Lucia is a graduate student in computer science at the University of Washington. His research interests include hardware support for dynamic analysis of concurrent software, with a focus on debugging and increasing software reliability. Lucia has a BS in computer science from Tufts University. He is a member of the ACM.

Joseph Devietti is a graduate student in computer science at the University of Washington. His research interests include parallel-programming models, and the intersection of architecture and programming languages. Devietti has a BS in engineering and computer science from the University of Pennsylvania. He is a member of the ACM.

Karin Strauss is a researcher at the AMD Research and Advanced Development Labs and an affiliate assistant professor in the Department of Computer Science and Engineering at the University of Washington. Her research interests encompass multiprocessor systems and enabling technologies for multiprocessor adoption, including hardware support for debugging. She has a PhD in computer science from the University of Illinois at Urbana-Champaign. She is a member of the ACM and the IEEE.

Luis Ceze is an assistant professor in the Department of Computer Science and Engineering at the University of Washington. His research interests include computer architecture and systems to improve the programmability of multiprocessor systems. He has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the ACM and the IEEE.

Direct questions and comments about this article to Brandon Lucia, Dept. of Computer Science and Engineering, Univ. of Washington, PO Box 352350, Seattle, WA 98195; blucia0a@cs.washington.edu.