Greedy Coherence

Emily Fortuna Brandon Lucia Adrian Sampson Benjamin P. Wood Luis Ceze
{fortuna,blucia0a,asampson,bpw,luisceze}@cs.washington.edu
University of Washington, Department of Computer Science and Engineering

ABSTRACT

Atomicity violations and violations of sequential consistency are two broad classes of concurrency errors that reduce the reliability of software. These failures often occur when different processors interleave their memory accesses at a fine grain. Greedy Coherence (GreCo) is a simple hardware technique that delays responses to some coherence requests to make this kind of sharing coarser, separating different processors' accesses to the same memory location in time. These delays avoid operation interleavings that lead to some atomicity violations and sequential consistency (SC) violations, making software more robust to these types of faults. We describe two implementations of GreCo: one that monitors coherence protocol messages and uses additional hardware structures to identify and avoid likely atomicity and SC violations; and a lower-complexity implementation that leverages the existing processor write buffer and focuses mainly on avoiding SC violations. Simulation results show that GreCo avoids failures in several bug kernel programs and incurs a negligible performance impact—less than 2%—on programs from the PARSEC benchmark suite.

1. INTRODUCTION

Multicores are already pervasive. In order to reap their potential performance benefits, concurrent and parallel programming must gain mainstream acceptance. Concurrent programming is also required for reasons other than performance in a broad class of important applications that are inherently concurrent, such as operating systems, web browsers, and mobile applications.

Unfortunately, concurrent programming is extremely challenging. Programmers must consider the interactions of operations in different threads of execution. These interactions are mind-bogglingly numerous and often subtle. The complexity of concurrent programming often leads to errors such as atomicity violations and data races, which can in turn lead to violations of sequential consistency (SC). Concurrency errors are especially troubling because they may only manifest in the presence of certain, rare thread interactions. Hence, even with thorough testing and experienced development teams, latent errors can reach deployment, leading to costly failures in deployed systems.

A substantial body of prior work has addressed the complexity of concurrent software with new testing strategies [2, 5, 10, 20] and techniques for identifying the causes of concurrency errors [4,8,18, 23, 26], in many cases using hardware support [9, 13, 14]. Testing and bug identification techniques have had considerable success in exposing faults to programmers and shepherding programmers to their root causes. However, both strategies rely on the assumption that programmers are available and capable of fixing their errors. A study of concurrency bugs and their fixes [12] suggests that many difficult bugs are left unaddressed for long periods of time—sometimes years. Additionally, when errors are eventually addressed, programmers often "fix" them incorrectly.

There have also been proposals to prevent concurrency bugs from causing failures by preventing untested thread interactions [6, 25]. These approaches show promise in that they obviate the need for the involvement of programmers. However, by relying on memoization of prior executions to determine acceptable behavior, these approaches remain limited due to the enormous space of possible multithreaded executions. Furthermore, these techniques and others that have targeted specific bug classes [15] cannot avoid the effects of sequential consistency violations. Some prior work has been successful at using hardware support to avoid sequential consistency violations [9, 16], but is limited by high implementation complexity [16] or inability to deal with sequential consistency violations involving store reorderings [9]. Work remains to be done to provide a low-complexity, high-efficiency mechanism that covers a broad variety of concurrency errors.

In this work, we propose a new technique for dynamically avoiding failures in concurrent programs, focusing on atomicity violations and sequential consistency violations. Prior work has shown that unspecified atomic regions tend to contain few instructions [13,17], and architectural write buffers hold values for a short window during which sequential consistency can be violated [22]. Motivated by these observations, we look for bugs by focusing on fine-grained memory access interleavings over short windows of dynamic execution. In this work, we are driven by simplicity: we consider all fine-grained interleavings of memory accesses to the same memory location by different processors *hazardous*.

The core of our proposed mechanism is making processors *greedy* with the data in their caches. Using existing cache coherence support and simple hardware data structures, processors monitor memory access interleavings. Our goal is to identify incoming memory access requests that indicate fine-grained sharing. Processors consider fine-grained sharing hazardous, so they delay their reply to requests that could cause it. This delay gives write buffers time to flush, preventing potential SC violations, and permits partially executed atomic regions to execute without being erroneously interleaved. We call the resulting technique *Greedy Coherence* (GreCo).

To summarize our main contributions:

- We develop a technique that identifies and delays certain inter-thread communication events to prevent the manifestation of atomicity and SC violations.
- We propose Greedy Coherence (GreCo): a simple architectural mechanism that implements our detection and delay techniques to prevent failures.
- We describe two implementations of GreCo: One that requires minimal hardware modification and focuses on preventing SC violations; and another that uses additional simple hardware structures to better handle both atomicity and SC violations.
- Using simulation, we evaluate both our implementations and show that GreCo effectively avoids errors with negligible impact on application performance.

2. BACKGROUND: THE PROBLEM WITH FINE-GRAINED SHARING

The goal of GreCo is to avoid atomicity violations and sequential consistency violations. The motivation behind GreCo is that concurrency errors often occur when multiple processors interleave their accesses to shared memory at a fine grain. In this section, we show how fine-grained sharing leads to the manifestation of concurrency errors.

SC Violations. Figure 1 shows a snippet of a program that illustrates how fine-grained memory access interleaving can lead to *violations of sequential consistency* on architectures with relaxed memory consistency models. The error in this program is due to the fact that processors P1 and P2 both access x and y without synchronizing their accesses. Executing this program on a system that employs write-buffering (such as all modern x86 machines) can lead to a violation of SC if each processor buffers its write and then executes its read without flushing the write buffer. In the case where neither buffered write is propagated to memory, both threads' reads see 0. There is no SC execution under which this result is possible, so such an execution manifests an SC violation.

Notice that only *fine-grained* interleavings of memory accesses can lead to such violations of SC. In this example, the error manifests because the reads execute immediately after the writes and within a short time span of one another. If enough time had elapsed for either write to leave its processor's write buffer before the other processor's read, the execution would correspond to an SC interleaving.

Atomicity Violations. Figure 2 illustrates how fine-grained memory access interleaving can lead to the manifestation of *atomicity violations*. In the example, processor P1 is executing a pair of writes that should be atomic with one another. Concurrently, P2 executes a read to the same memory location. If P2's read interleaves P1's writes, as shown, P2 reads the intermediate value produced by the first of P1's writes. P1's writes should have been atomic, so P2 should not have been able to read this value. Prior work has shown that such intended atomic regions tend to contain just a few memory operations [12,13,17]. As a result, violation of the atomicity of these regions only occurs when accesses interleave at a fine grain.

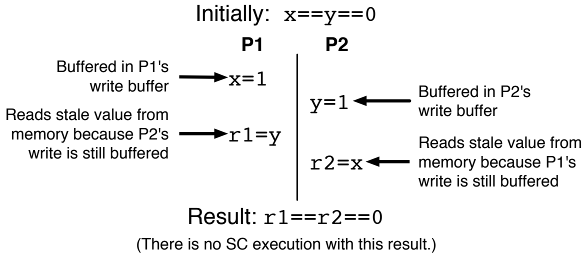

Initially: x==y==0
P1
P2
Buffered in P1's write buffer
x=1
y=1
Buffered in P2's write buffer
Reads stale value from memory because P2's write is still buffered
r1=y
r2=x
Reads stale value from memory because P1's write is still buffered
Result: r1==r2==0
(There is no SC execution with this result.)

Figure 1: An SC violation resulting from a data race on an architecture with write buffering. Accesses to x and y are not properly synchronized, permitting both processors to buffer write accesses locally.

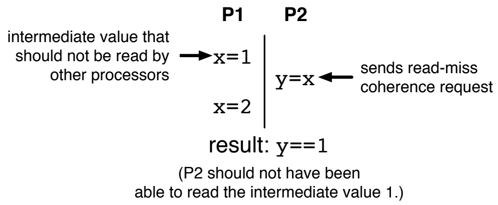

P1
P2
intermediate value that should not be read by other processors
x=1
y=x
sends read-miss coherence request
x=2
result: y==1
(P2 should not have been able to read the intermediate value 1.)

Figure 2: A violation of atomicity due to improper protection of an atomic region. The two writes to x in P1 are not properly synchronized, so P2 is able to read an intermediate value.

3. GREEDY COHERENCE

As illustrated in Figures 1 and 2, SC violations and atomicity violations manifest when different processors interleave operations at a fine grain. If the interleaving of accesses in the examples was coarser, the bugs would not have manifested. The key idea in GreCo is to detect fine-grained interleavings and treat them as potential concurrency errors. To prevent failures, GreCo changes the access interleaving by selectively increasing the latency of some memory accesses with delays. Such delays coarsen the granularity of interleavings, preventing many violations of SC and atomicity.

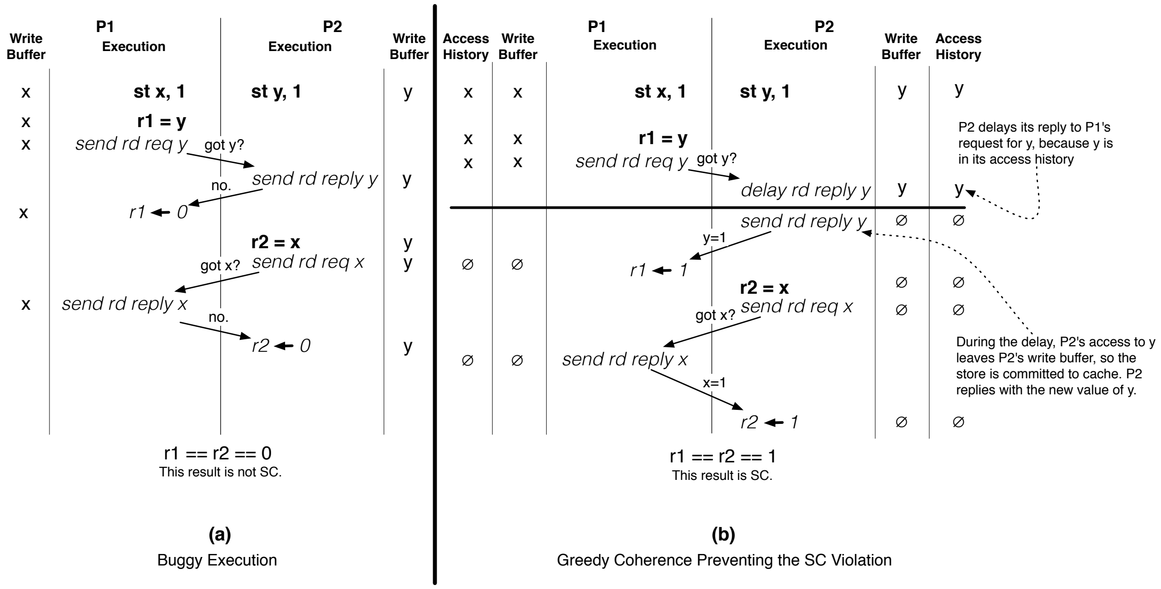
GreCo has two parts. First, GreCo uses a program monitoring mechanism to detect fine-grained memory access interleavings. Second, GreCo uses carefully placed delays to guide the execution away from these interleavings. We now describe GreCo's basic mechanisms abstractly; Section 4 details GreCo's implementation.

3.1 Detecting Potential Concurrency Errors

GreCo monitors the execution of programs in order to identify fine-grained memory access interleavings. As a basis for our design, we assume a cache-coherent shared-memory multiprocessor.

In GreCo, each processor maintains an *access history*, which records a short, fixed-length sequence of its recently-accessed addresses. Read and write accesses histories are maintained separately.

When a processor receives an access request from another processor for a piece of data, the processor checks its access history for an access to the requested data. The access history only contains a



(a)
Buggy Execution

(b)
Greedy Coherence Preventing the SC Violation

Figure 3: Greedy Coherence avoiding a violation of SC by delaying `P2`'s reply to `P1`'s request to read `y`. The horizontal line in (b) indicates a delay inserted by GreCo. *Real time flows down.*

few entries, so a matching entry indicates that fine-grained memory access interleaving is occurring with the requesting processor.

3.2 Avoiding Errors

The essence of the errors shown in Figures 1 and 2 is that a processor receives an access request from another processor for some memory location too soon after it has accessed that location locally. In GreCo, processors avoid failures by delaying their response to incoming requests for cache lines that they have accessed very recently (i.e., lines in their access histories). By delaying the completion of the incoming access, GreCo increases the time between the two accesses in the execution, reducing the likelihood of a failure. Figures 3 and 4 show how GreCo avoids the manifestation of errors by detecting such access interleavings.

Note that fine-grained interleaving of accesses to a memory location is not necessarily a sign of an error. However, in a program with a data race or an incorrectly specified atomic region, the manifestation of the error is often the result of such a fine-grained access interleaving. Spurious delays during benign fine-grained sharing are a potential performance problem for GreCo. However, we show in Section 6.3 that GreCo's performance impact is minimal.

SC Violations. GreCo increases the latency of some coherence requests to prevent processors from accessing memory locations for which other processors have buffered writes. Figure 3 shows the program from Figure 1 executing on a system with GreCo. There are two processors, `P1` and `P2`. The processors each have a write buffer, which holds write accesses for a short period of time before their results are made visible to other processors. When a processor performs a memory operation, it sends an access (coherence) request to other processors. When the recipient of the request replies, the originator of the request proceeds with its memory operation. Some access requests are omitted from the figure for illustrative purposes. The processors' GreCo access histories are also shown. The execution begins with both `P1` and `P2` executing their store instructions. The stores are placed in the processors' respective write buffers, establishing the necessary condition for the violation of SC. Without intervention, `P1`'s execution of `r1 = y` would read a stale value of `y`. However, with GreCo, the presence of `y` in `P2`'s access history causes `P2` to delay its reply to `P1`'s read request. During the delay, `y` leaves `P2`'s write buffer, and the value of `y` is made globally visible. When `P2` finally replies, and `P1` completes its access, `P1` correctly reads the value for `y` that `P2` wrote.

Note that GreCo only prevents fine-grained access interleavings *within the same cache line*. Fine-grained interleaving of accesses across different lines proceed normally. For example, if `P1` had read another variable `z` in a different cache line before reading `y`, that read to `z` would not have been delayed.

Atomicity Violations. Figure 4 shows how GreCo avoids the atomicity violation in Figure 2. In the example, `P1` writes an intermediate value of 1 to `x` that is not intended to be visible to other processors. However, the program does not reflect this intention, and `P2` can read `x` between `P1`'s writes to `x`. With GreCo, `P1` does not respond immediately to `P2`'s request because `x` is in `P1`'s access history. During this period of delay, `P1` finishes executing both writes to `x`. Later, after the delay finishes, `P1` responds to `P2`'s request with the correct value `x = 2`. GreCo prevents the interleaving that exposes the intermediate value by increasing the latency of `P1`'s response to `P2`'s request.

4. ARCHITECTURE

We have developed two different implementations of GreCo: *Explicit-History Greedy Coherence* (GreCo-Hist), and *Write-Buffer-Based Greedy Coherence* (GreCo-WB). GreCo-Hist probabilistically avoids both atomicity and SC violations with modest hardware extensions. GreCo-WB requires even fewer modifications to hardware than GreCo-Hist. However, GreCo-WB does not avoid atomicity violations as well as GreCo-Hist. We now describe the implementation of

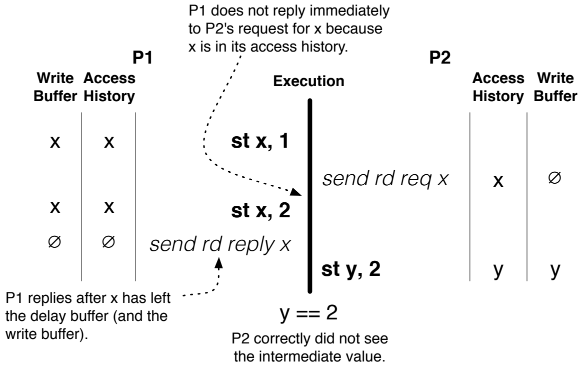

P1 does not reply immediately to P2's request for x because x is in its access history.
P1
Write Buffer
Access History
Execution
P2
Access History
Write Buffer
x
x
st x, 1
send rd req x
x
∅
x
x
st x, 2
∅
∅
send rd reply x
st y, 2
y
y
P1 replies after x has left the delay buffer (and the write buffer).
y == 2
P2 correctly did not see the intermediate value.

Figure 4: Greedy Coherence avoiding an atomicity violation by delaying `P1`'s response to `P2`.

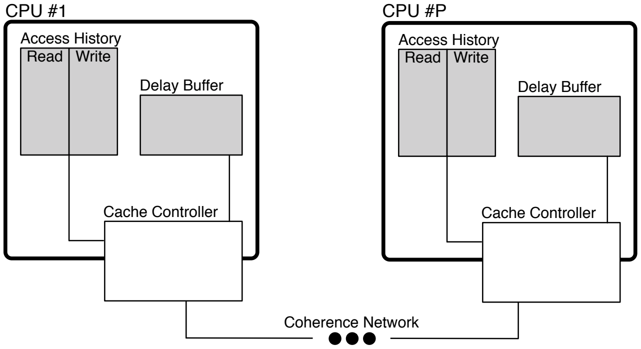


Figure 5: A block diagram of the GreCo-Hist architecture. Processor components not included in the diagram are unchanged. New components are shaded. Note that the access history and the delay buffer are both connected to the cache controller so that accesses can be monitored, and coherence replies can be delayed.

GreCo-Hist; the following subsection describes the simpler hardware modifications required for GreCo-WB.

4.1 Explicit-History Greedy Coherence

There are two main components to the architecture support for GreCo: (1) support for collecting and maintaining an access history; and (2) support for detecting which incoming access requests should be delayed and carrying out those delays. GreCo-Hist adds two main structures to the satisfy these design goals: the *access history* and the *delay buffer*. Figure 5 shows a block diagram of the hardware extensions that implement GreCo-Hist.

4.1.1 Access History

The access history is a data structure that tracks recent memory accesses made by a processor. The processor uses its access history to determine when it should delay its reply to incoming memory access requests by other processors, as described in Section 3.

The access history is implemented as a searchable FIFO queue of line-granular memory addresses. When a processor executes a memory operation, it inserts the address that was accessed into the access history. The access history has a fixed size, and so inserting a new address may cause an address accessed earlier to be pushed out of the queue.

The access history structure is content-addressable so that it can be searched on every incoming coherence request. Requests for addresses in the history are delayed until the address leaves the history. The mechanism for implementing this delay is described in more detail in the next section.

The size of a processor's access history determines the length of time during which accesses to a given address are delayed following its own access to that memory location. Ideally, the number of entries in the access history would be close to the number of memory operations involved in the manifestation of errors. Prior work [13, 17, 21] has shown that this number is small, so we assume a modest default length of 128 entries.

In GreCo, processors keep an access history for read and write operations separately. Separating accesses by type is important because delays are only useful in situations that can lead to errors. Specifically, a processor should only delay an incoming read request if it has recently *written* the location—if it has recently read the location but not written it, then no delay is necessary. Keeping separate read and write histories allows the processor to identify cases involving only reads, and not delay. This behavior is essential to GreCo's low performance impact as it preserves delay-free read sharing.

It is possible for a processor to be delayed by another processor that is also being delayed. If two processors delay one another's access requests, a deadlock will result because neither processor can execute new memory accesses that push accesses from their histories (recall that delays last until the delay-causing access exits the history structure). GreCo prevents these deadlocks by limiting the amount of real time an access remains in a processor's access history while that processor is not executing new memory accesses. By guaranteeing that every access in an access history will eventually leave even if its host processor makes no further memory accesses, GreCo ensures that all delays will eventually end.

Specifically, the access history has a register associated with it that stores a *progress countdown timer*. The countdown timer is initialized to its maximum value. Each time an entry is enqueued into the access history, the register is reset to its maximum value. Each cycle, the register is decremented. When the register reaches zero, an empty placeholder entry is enqueued into the access history, pushing the oldest entry out of the history.

If a processor is involved in a deadlock, its access history (and that of all involved processors) will begin to empty as its progress countdown timer repeatedly reaches zero. Eventually, after one of the delay-causing entries leaves a processor's history, that processor will permit the delayed access to proceed, breaking the deadlock.

4.1.2 Delay Buffer

The delay buffer is a data structure that keeps track of incoming coherence requests to which a processor is delaying its reply. It consists of a set of registers, each of which contains a coherence request message (i.e., an address along with flags indicating the request type) together with a *reference count* indicating the number of times the address appears in the access history.

When an incoming request is delayed (i.e., it hits in the access history), it is placed in a free register in the delay buffer. The processor counts how many access-history entries refer to the address; the new delay buffer entry's reference count field is initialized to this value. No response is sent to the requesting processor, so the remote processor's memory operation does not complete.

Whenever an address exits the access history, the processor searches for entries in the delay buffer that refer to the address and decrements those entries' reference counts. If any reference count reaches zero, the processor replies to the corresponding delayed request, completing the GreCo delay.

The number of entries in the delay buffer is determined by the maximum number of outstanding requests a processor can handle. In a system with P processors, each with M miss-status holding registers (i.e., at most M outstanding memory requests), the delay buffer must have $(P - 1) \times M$ entries. The size of each entry is determined by the size of coherence requests and the size of the access history. Assuming requests consist of 48 address bits and 2 request type bits and the access history has 128 entries, each delay buffer register is 57 bits.

4.2 Write-Buffer-Based Greedy Coherence

In GreCo-Hist we add searchable FIFOs to implement GreCo's access history. However, modern processors implicitly track their write access history in write buffers. Write buffers hold memory writes between when they are retired and when their effects are visible in the memory system. To decrease GreCo's design complexity, we can reuse the existing write buffer as the access history. A write buffer is typically organized as a searchable FIFO, precisely what is needed to implement an access history.

4.2.1 Implementing GreCo-WB

In GreCo-WB, the write buffer is used as the access history. The delay buffer is identical to the structure in GreCo-Hist. When a processor receives a coherence request, it searches its content-addressable write buffer for the requested address.

To ensure forward progress, GreCo-Hist requires the addition of a progress countdown timer to drain the access history, even if the processor does not enqueue new accesses. In typical write buffer implementations, accesses are guaranteed to *eventually* leave the buffer and become visible to other processors, obviating the need for a progress countdown timer.

4.2.2 Benefits of GreCo-WB

There are two main benefits to using the write buffer as an access history. First, design complexity is reduced because a separate access history is unnecessary. Second, detection of SC violations is more precise than in GreCo-Hist. The write buffer contains precisely the set of accesses that are relevant in avoiding SC violations. If a processor delays all accesses to data in its write buffer until that data leaves the write buffer, no violations of SC are possible, assuming a processor implementing an x86-TSO-like memory model [22]. Due to the fact that GreCo delays coherence requests to prevent SC violations, GreCo cannot delay *all* accesses to data in write buffers: GreCo is unable to avoid SC violations involving cache hits, which do not require coherence.

There are two main limitations posed by the use of the write buffer as an access history. Firstly, the write buffer is limited in size. This size limitation reduces the window over which atomicity violation bugs can be detected and prevented. Secondly, the write buffer only records write accesses, not reads. Thus, GreCo-WB cannot prevent any errors by delaying a remote write access after a local read access. Note that these limitations do not impede GreCo's ability to prevent SC violations.

5. CORRECTNESS

In this section we show how GreCo preserves the guarantees of coherence (Section 5.1) without inducing deadlock (Section 5.2). We describe which sequential consistency violations (Section 5.3) and atomicity violations (Section 5.4) GreCo is able to avoid. Throughout, we assume a broadcast-based cache coherence implementation using a split-transaction bus.

5.1 Preservation of Coherence

GreCo preserves the coherence guarantee provided by the cache coherence protocol: for each cache line in the shared address space, there exists some global total order of accesses to that line that allows each processor's observed order of accesses to that line. (Enforced read-read ordering is typically partial, but all other component orderings are total.) Coherence protocols often implement this guarantee by enforcing a global total order on coherence messages concerning a single line. Processors respond to requests for a line in the order they are received. Furthermore, when a processor makes a request, it waits for all responses before itself responding to any subsequent requests from other processors for the same line.

Delayed coherence replies are possible in our assumed base coherence implementation. A processor assumes no bound on the time between when it sends a coherence message and when it receives a reply from the other processors in the system. Coherence protocols are resilient to such latency variations in order to remain decoupled from the particular implementation (e.g., one that ensures replies always arrive in a fixed interval). GreCo works by inserting additional, bounded delays before coherence replies are sent. From the perspective of the coherence protocol, GreCo is transparent—it only affects the latency of request completion.

5.2 Deadlock Freedom and Forward Progress

During a program's execution, two or more processors may issue a series of coherence requests (for different lines) that cause the processors to wait for one another to reply before either can make progress. This situation represents a *cycle* of coherence requests. In a system without GreCo, cycles are inconsequential because every processor responds to every request as soon as it can.

In a system with GreCo, such a situation is complicated if the lines being accessed by the processors are in one another's access histories. In this case, GreCo causes each processor to delay its reply to the other. Progress is hampered by such a mutual delay. However, as described in Section 4, GreCo automatically empties access histories in order to avoid deadlock.

During a mutual delay, the progress timeout mechanism in each access history continues to push old accesses out. Eventually, one processor's delay-causing history entry leaves the access history. The processor replies to the delayed coherence request, permitting the other processor to make progress. The amount of time that can be wasted resolving a delay cycle is bounded by the time required to completely empty the access history. Note that if the interleaving the delays tried to avoid will not be avoided in this case.

In a system with GreCo, at least one processor is always able to make forward progress, in spite of coherence request cycles. However, this does not guarantee the progress of individual threads. If a processor repeatedly accesses the same line, that line may never be absent from that processor's access history. In such a situation, all other processor's requests for the line will be delayed indefinitely, potentially starving other processors. In the worst case scenario, this effect can serialize threads' executions. In real programs, however, repeated accesses to the same line don't often occur with sufficiently high frequency to cause starvation. More often, threads have program dependences that must be fulfilled by other processors, permitting accesses to leave their history. Furthermore, under GreCo-WB, and assuming no false sharing (we discuss false sharing in Section 5.5), only racy accesses can be delayed, because the write buffer is flushed at synchronization.

5.3 Sequential Consistency

An execution is sequentially consistent if there exists a global total order on memory operations (such that every read gets the value from the most recent write to the same address) that allows the orders observed by each processor. Architectures with write buffers can violate SC when a processor buffers a write and retires a subsequent read before the buffered write is globally visible, effectively reordering the operations. x86-TSO [22] defines an abstract memory model in which the reordering of reads with buffered writes is the only source of SC violations. Under TSO, GreCo can prevent some SC violations by preventing read misses from reading memory locations while a processor has a write buffered.

Without GreCo, if a processor p_1 buffers a write to a line l, and another processor p_2 takes a read miss for l, then p_2 may get a stale value from memory, possibly leading to an SC violation. With GreCo, p_2's read is delayed until p_1's write leaves its write buffer and is globally visible. Therefore, p_2 reads the value of p_1's write to l, which is the most recent, so SC is preserved. Assuming that all reads miss, GreCo forces every read to get the value from the globally most recent write, enforcing SC.

However, GreCo does not prevent SC violations involving read hits. If p_2 has l in the shared state in its cache when it executes its read, then it need not send a coherence request to access l. Without a coherence request to delay, GreCo cannot prevent p_2 from reading the stale value of l. SC may be violated in this case. GreCo's avoidance guarantees depend on the state of the caches. On write-buffer-based TSO architectures, GreCo-WB prevents all SC violations that involve only cache misses.

5.4 Atomicity

GreCo prevents potential single-variable atomicity violations that satisfy the following conditions. First, if the intended atomic section is in progress on processor p, the potential interleaving access must generate a cache miss. This condition ensures that the potential interleaving access sends a coherence request to which processor p can delay its response in order to avoid the atomicity violation. Second, if the intended atomic section on line l is in progress on processor p, at least one *conflicting* access to line l must be present in processor p's access history at all times until completion of the atomic section. (Two accesses *conflict* if they access the same memory location and at least one is a write.) This ensures that processor p continues to delay its response to the coherence request for the potential interleaving access until the intended atomic section has completed and the access will not violate its atomicity. A useful proxy for this condition is that accesses to line l within the intended atomic section be separated by fewer operations than the size of the access history *and* that processor p not be involved in any deadlocks that may drain its access history.

5.5 False Sharing

GreCo may exacerbate the performance impact of false sharing. GreCo tracks accesses at a cache line granularity and may cause delays even if no bytes in the line are actually shared. To GreCo, accesses to different bytes of the same line are indistinguishable from accesses to the same byte. The false sharing problem is not unique to GreCo. False sharing is a known performance pitfall that programmers and compilers often attempt to minimize already.

6. EVALUATION

We evaluated our system to determine (1) how well GreCo avoids failures and (2) how much performance overhead GreCo introduces over conventional execution.

6.1 Methodology

We developed a simulated implementation of both GreCo-Hist and GreCo-WB using the Pin binary instrumentation infrastructure [19]. The simulator includes a model of an 8-core processor. Each core models a 4-way associative 32KB L1 data cache, a delay buffer, and a 128-entry access history with a 50 cycle progress countdown timer. The simulated core includes a write buffer model that affects access costs, determines potential SC violations and, in GreCo-WB, replaces the access history. We use a simple instruction cost model derived from a specification for the Intel 80486 [24]. Cache hits cost 1 simulated cycle, cache misses cost 100 simulated cycles. Threads are assigned to a simulated core when they are spawned in round-robin fashion. The simulator executes one cycle from each thread during each global time step, whether executing a single- or multi-cycle instruction or a delay due to GreCo.

Benchmarks. We used a set of atomicity violation bug kernel benchmarks to evaluate and characterize the avoidance capability and efficiency of GreCo. The bug kernels are variants of programs used in prior work on hardware support for concurrency error detection [3, 17]. The programs range from about 50 to 300 lines of code. Each creates 8 threads to execute code that results in an atomicity violation under some interleavings. We also used the PARSEC benchmark suite to further evaluate the performance characteristics of GreCo.

6.2 Synthetic Benchmark Experiments

Figures 6 and 7 show the fraction of failures avoided, the total multithreaded performance overhead compared to a system without GreCo, and the aggregate number of cycles spent in delay by all threads as a fraction of the total multithreaded runtime for GreCo-Hist and GreCo-WB running our synthetic benchmark programs.

The data in Figure 6 show that GreCo-Hist eliminates virtually all dynamic manifestations of the atomicity violations in all kernel benchmarks. The number of delays is often quite high, varying from around 50% of the non-GreCo execution time to around 150%. In spite of the high delay frequency, the performance overhead is often disproportionately low—in all but `homemade`, `mysql`, and `unc`, the overhead is negligible. These data suggest that, in applications with lower overhead, the delayed threads are not on the application's critical path—during the delay, another thread can proceed with useful computation. The result is that the delays do not increase the total multithreaded execution time substantially. In the applications with higher overhead, it is likely the delays are on the critical path, hindering the execution's progress.

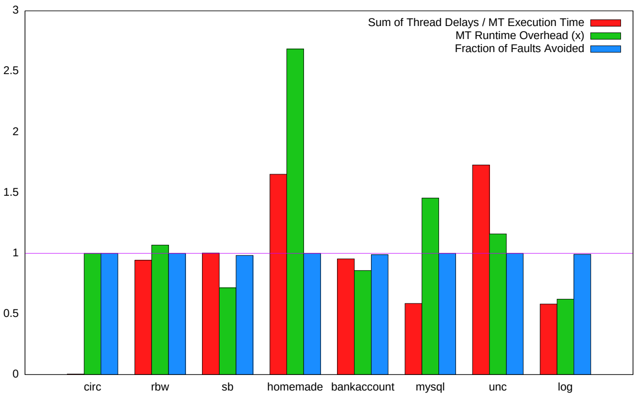


Figure 6: Failure avoidance, performance overhead, and delay characteristics of GreCo-Hist running bug kernels.

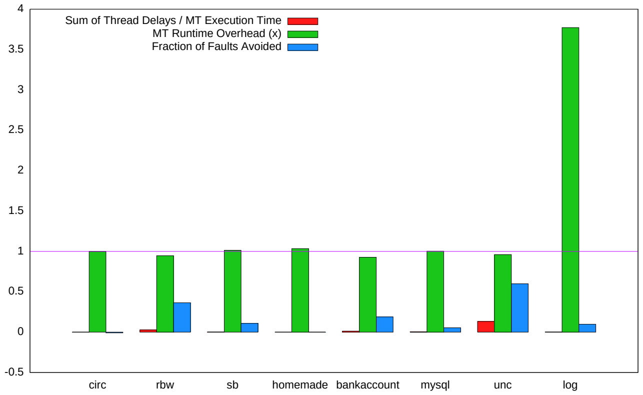


Figure 7: Failure avoidance, performance overhead, and delay characteristics of GreCo-WB running bug kernels.

The data in Figure 7 show that GreCo-WB is able to avoid between 0% and 50% of the atomicity violation manifestations in our kernel benchmarks. This result suggests that, while a substantial improvement in failure avoidance comes with the addition of a read history (as in GreCo-Hist), the simpler GreCo-WB can also avoid some atomicity violations. Figure 7 also shows that GreCo-WB rarely leads to delays and hence has negligible performance overhead over our baseline system. The exception to this trend is `log`, which incurred a performance overhead greater than a factor of 3.

Dekker's Algorithm Microbenchmark. We wrote a microbenchmark implementing Dekker's algorithm for mutual exclusion, which can fail to provide mutual exclusion if its execution is not sequentially consistent. Two threads spin in a loop to enter a critical section 1000 times. The goal of this experiment is to show that GreCo avoids potential SC violations with low overhead. We consider an access a *potential SC violation* if it accesses data written by a write in another processor's write buffer.

Running without GreCo, we saw 5997 potential SC violations. With GreCo-Hist, there was 1 potential SC violation and the performance overhead was 5%. Running with GreCo-WB, there were 0 potential SC violations and the performance overhead was 13%. GreCo-WB imposed a slightly higher overhead than GreCo-Hist but avoided all potential SC violations. In contrast, GreCo-Hist failed to avoid all potential SC violations but imposed lower overhead than GreCo-WB. In spite of the differences, the performance overheads of both schemes are acceptably low.

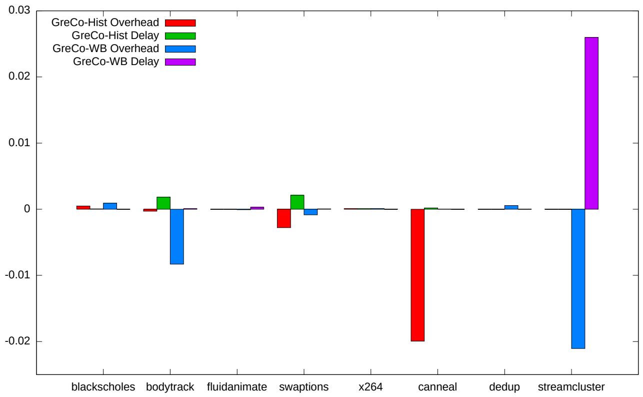


Figure 8: Performance overhead and delay characteristics of GreCo-Hist and GreCo-WB running PARSEC.

6.3 Performance

Our performance and avoidance results for GreCo-Hist and GreCo-WB using these kernel benchmarks are promising because the kernel programs we used are *pathological* to our system; they execute buggy code in a tight loop with high frequency. Such programs trigger delays and experience failures more often than real applications. Figure 8 shows the performance overhead of GreCo-Hist and GreCo-WB running applications from PARSEC, a benchmark suite representative of real-world workloads [1].

Across the board, the overhead for either scheme never degrades performance by more than a few percent (`streamcluster`) and, in some cases, GreCo slightly improves the applications' performance (`canneal`, `bodytrack`). Such small variations in either direction indicate that the performance impact of GreCo is negligible for these applications. These data also show that real applications tend to share data less frequently than bug kernels (Figures 6 and 7) because GreCo delays memory accesses less frequently during executions of PARSEC applications than kernels. The sharing pattern in these applications is therefore synergistic with the design goals of GreCo: most of the time, the application is not sharing and GreCo does not impose on the execution; occasionally, the application engages in high-frequency sharing and GreCo imposes with low overhead to prevent that sharing from manifesting failures. These results show that GreCo is well-suited for use in deployed systems, where high performance is essential.

7. RELATED WORK

Research related to GreCo includes a variety of work on using hardware [9, 15–17] and software [11] support to avoid failures in concurrent programs, work aimed at identifying the cause of concurrency errors [4,8,18,23,26], often using hardware support [9,13,14, 27], and work developing hardware and software-only systems that avoid concurrency errors by forcing executions to adhere to tested thread interleavings [6, 25].

Prior techniques for avoiding concurrency errors focus exclusively on data races [9, 16] or atomicity violations [15, 17]. In addition, their hardware complexity is very high—most [9, 15, 17] require support for speculative execution. In contrast, GreCo focuses on both atomicity violations *and* SC violations, using one unified mechanism. Furthermore, the mechanisms required by GreCo have low implementation complexity and do not require costly speculation support. Unlike software approaches like AFix [11], GreCo does not require source code or synchronization modifications; instead, GreCo transparently changes the timing of certain operations to avoid potential errors.

Systems built to identify the cause of concurrency errors [4,7,8,13, 14,18,23,26,27] contribute to increasing software reliability by informing developers why failures occurred. These systems are complementary to GreCo, as GreCo does not answer questions about why a failure occurred; it instead focuses on avoiding the failure altogether. Software systems such as ConSeq [26], Falcon [23], Recon [18], and FastTrack [7] impose considerable overheads on an execution and are only suitable for use during development. In contrast, GreCo does not have a significant impact on the performance of the application, so it is applicable to deployed systems. Hardware approaches [13, 14] more closely match the performance characteristics of GreCo but do not avoid failures.

Some prior work has proposed systems that rely on a database of tested thread interactions to avoid errors by avoiding untested interactions [6, 25]. Using a strategy of rote learning, these systems avoid many errors without false positives in inferring likely errors. In contrast, GreCo is more general. GreCo uses a short history of operations observed during an execution to avoid failures. Because it considers fine-grained sharing hazardous, GreCo's inductive bias is stronger than rote learning. As a result, GreCo is effective even in untested code. Additionally, our experiments show that GreCo's approach does not lead to frequent false positives, as delays are rare and overheads are negligible for PARSEC.

8. CONCLUSION

Greedy Coherence is a technique that employs simple hardware extensions to avoid concurrency errors. By strategically delaying replies to some coherence requests, GreCo perturbs the interleaving of threads' memory operations to coarsen fine-grained sharing. The result is that GreCo avoids the manifestation of some atomicity violations and violations of sequential consistency. GreCo can be implemented with low complexity, using simple additions to the memory system that are off the critical path; in the case of GreCo-WB, the write buffer already present in many architectures can be reused. Furthermore, GreCo does not require modifications to the coherence protocol. We have shown that GreCo can avoid failures in kernel benchmarks and imposes negligible performance overhead in real-world benchmark programs. With low complexity and high performance, GreCo can increase the reliability of deployed systems with no additional programmer effort.

9. REFERENCES

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[3] L. Ceze, C. von Praun, C. Caşcaval, P. Montesinos, and J. Torrellas. Concurrency control with data coloring. In *MSPC*, 2008.

[4] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[5] Corensic Inc. Jinx, September 2011. `http://www.corensic.com/Products.aspx`.

[6] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[7] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[8] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[9] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *SPAA*, 1991.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[11] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[14] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[15] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, 2010.

[16] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, 2010.

[17] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[18] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI*, 2011.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[20] M. Musuvathi. Systematic concurrency testing using CHESS. In *PADTAD*, 2008.

[21] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO*, 2010.

[22] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[23] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[24] Z. Smith. The Intel 8086 instruction set. `http://home.comcast.net/~fbui/intel.html`.

[25] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[26] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[27] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: hardware-assisted lockset-based race detection. In *HPCA*, 2007.