Kivati: Fast Detection and Prevention of Atomicity Violations

Lee Chew David Lie

Department of Electrical and Computer Engineering
University of Toronto

Abstract

Bugs in concurrent programs are extremely difficult to find and fix during testing. In this paper, we propose Kivati, which can efficiently detect and prevent atomicity violation bugs. Kivati imposes an average run-time overhead of 19%, which makes it practical to deploy on software in production environments. The key attribute that allows Kivati to impose this low overhead is its use of hardware watchpoints, which can be found on most commodity processors. Kivati combines watchpoints with a simple static analysis that annotates regions of codes that likely need to be executed atomically. The watchpoints are then used to monitor these regions for interleaving accesses that may lead to an atomicity violation. When an atomicity violation is detected, Kivati dynamically reorders the access to prevent the violation from occurring. Kivati can be run in prevention mode, which optimizes for performance, or in bug-finding mode, which trades some performance for an enhanced ability to find bugs.

We implement and evaluate a prototype of Kivati that protects applications written in C on Linux/x86 platforms. We find that Kivati is able to detect and prevent atomicity violation bugs in real applications, and imposes very reasonable overheads when doing so.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Concurrency, Scheduling, Synchronization, Threads; D.4.5 [*Reliability*]: Fault-tolerance

General Terms Reliability

Keywords Kivati, Watchpoint, Atomicity Violation

1. Introduction

As the number of cores on a chip continue to increase, programmers will be further forced to write concurrent programs to take advantage of the additional cores. Unfortunately, such programs are prone to concurrency bugs, which are difficult to detect and fix. This is because concurrency bugs require a combination of two unlikely conditions to manifest. First, like regular non-threaded bugs, they require the right set of program inputs, which is exponential in number. Second, they also require the right thread interleaving, which is again exponential in number. As a result, concurrency bugs are likely to survive testing and remain in software when it is shipped to customers. Thus, a solution which can detect and prevent concurrency bugs with low enough overhead that can be deployed after testing would enable programmers to better take advantage of the trends toward multicore processors.

In this paper, we present Kivati, which detects and prevents atomicity violation bugs. An atomicity violation is an interleaving of memory accesses such that one memory access interleaves between another set of memory accesses that have to be executed atomically for correctness. Furthermore, the interleaving must be non-serializable – that is, there is no equivalent ordering of the accesses such that the set of memory accesses could have executed atomically. Figure 1 gives a simplified version of an atomicity violation in the Firefox browser. Here, the program is checking that `shared_ptr` is NULL (a read at line 3) before assigning it a new value (a write at line 4). For correctness, the read and write of `shared_ptr` must be performed together atomically. Otherwise, two threads could both pass the check and both assign a new value to `shared_ptr`, leading to a lost update. The bug exists because the developer neglected to enforce atomicity with a lock. While other classes of concurrency bugs exist, such as deadlocks or ordering violations, atomicity violations are a major class of concurrency bug and have been shown to account for approximately 65% of all concurrency bugs [14].

Previous proposals have tried to improve the ability of testing to find concurrency bugs [5, 8, 13, 16, 19, 22, 25]. These systems try to exhaustively explore thread interleavings to increase the likelihood of a concurrency bug manifesting itself. However, they impose an execution slowdown of 2.2x-72x, which makes them too slow for deployment on production machines. Other systems can perform run-time bug prevention more efficiently, but either require

EuroSys'10, April 13–16, 2010, Paris, France.
Copyright © 2010 ACM 978-1-60558-577-2/10/04...$10.00

```
void func(void)
{
  if (! shared_ptr )
     shared_ptr = another_function ();

        ...
```

Figure 1. Simplified version of an atomicity violation bug (#225525) in Firefox.

$read_{local}(A)$	$read_{local}(A)$
$write_{remote}(A)$	$write_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$
$write_{local}(A)$	$write_{local}(A)$
$write_{remote}(A)$	$read_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$

Figure 2. Non-serializable interleavings of accesses to the variable A.

specialized hardware that does not currently exist on commodity processors [15, 26], or are only applicable to deadlocks [10, 23].

In contrast, Kivati is able to detect and prevent atomicity violations on commodity hardware with low overhead. Kivati does not rely on programmer annotations. Instead, it uses static analysis to approximate the set of accesses it believes should be atomic. These accesses are then checked at run-time for actual atomicity violations. Kivati also dynamically detects and handles cases where an atomicity violation is intentional and required for correctness. To reduce overhead and false positives, Kivati can be configured to ignore accesses where atomicity is not required. However, even without this tuning Kivati has an average execution time overhead of 19% and a worst case overhead of 30%, which is orders of magnitude smaller than the existing atomicity violation detection systems [5, 8, 13, 22, 25] and comparable to run-time deadlock avoidance systems [10, 23].

The key to Kivati's performance is its use of hardware watchpoints, which can be found on all Intel and AMD x86 processors, as well as other major processor architectures. The watchpoint hardware allows Kivati to efficiently detect interleaved accesses during regions of code it believes should be atomic. When such an interleaved access is detected, Kivati prevents the violation by dynamically reordering the accesses to preserve atomicity. Kivati is implemented directly in the operating system kernel and the static annotator is written using the CIL program analysis framework [17]. This allows Kivati to efficiently protect almost any application written in C from atomicity violations.

Kivati supports two modes of usage. In *prevention mode*, Kivati detects and prevents atomicity violations with as little overhead as possible. When an atomicity violation is detected, Kivati records the thread IDs and locations of the accesses it made atomic, as well as the thread ID and location of the violating access. This information can be used by the software developer to determine if the violation is actually a bug and if necessary, fix it. At the cost of slightly more overhead, Kivati can also operate in *bug-finding mode*. In this mode, Kivati artificially increases the likelihood of an atomicity violation occurring by pausing threads when they are in a section of code Kivati believes should be atomic. Just as in prevention mode, all atomicity violations are prevented in bug-finding mode, so the only apparent difference to the end-user is the reduced performance. A scenario where bug-finding would be useful is during beta-testing, where users might be willing to accept reduced performance in order to help find and report bugs.

We make three contributions in this paper. First, we describe the design of Kivati, which is the first system we are aware of to provide fast detection and prevention of atomicity violations on commodity hardware. Second, we have implemented a Kivati prototype for Linux running on x86 processors that is able to protect applications written in C. Implementing a prototype on the x86 architecture was particularly challenging because the x86 watchpoint hardware raises a trap *after* the violating memory access has completed, meaning that its effects have been committed to the architectural state of the machine. Finally, we evaluate Kivati on a suite of 5 applications, and 11 known bugs in these applications. Kivati is able to detect and prevent all bugs while imposing very modest execution time overheads.

We begin by defining the problem Kivati solves and describing the design of Kivati in Section 2. Section 3 then provides details on the implementation of our prototype and we evaluate the ability to detect and prevent atomicity violations and the performance of our prototype in Section 4. We compare Kivati against related work in Section 5 and give our conclusions in Section 6.

2. Overview

2.1 Problem definition

Intuitively, an atomicity violation occurs when one thread violates the atomicity assumed by another thread. For example, if Thread 1 writes 5 to a memory location M and then immediately reads from M, it expects the value 5. However, if Thread 2 writes some value to M, say 10, in between these two accesses, this expectation is broken. More formally, an atomicity violation occurs when a memory access of one thread interleaves with several memory accesses of another thread in a non-serializable way – that is, there exists no serial execution of the accesses that gives the same results. Using the same example, in the interleaved execution Thread 1 would read 10 (Thread 2's write). However, in *any* serial execution Thread 1 would read 5 (its own write). Since the results are different, an atomicity violation has occurred. We call the thread that makes the violating access the *remote*

```
begin_atomic(1,&shared1, ...); /* AR 1 starts */
tmp1 = shared1; /* 1st access to shared1 */
begin_atomic(2,&shared2, ...); /* AR 2 starts */
tmp2 = shared2; /* 1st access to shared2 */
tmp1 = tmp1 + 1;
tmp2 = tmp2 − 1;
shared1 = tmp1; /* 2nd access to shared1 */
end_atomic(1, ...); /* AR 1 ends */
shared2 = tmp2; /* 2nd access to shared2 */
end_atomic(2, ...); /* AR 2 ends */
```

Figure 3. Annotation example with overlapping ARs.

```
begin_atomic(1,&shared, ...) ; /* AR 1 starts */
if (shared) {
  begin_atomic(2,&shared, ...) ; /* AR 2 starts */
  shared = 0;
  /* AR 1 ends here if shared != 0 */
  end_atomic(1, ...) ;
}
tmp = shared;
/* AR 1 ends here if shared == 0 */
end_atomic(1, ...) ;
/* AR 2 ends here, does nothing if shared == 0 */
end_atomic(2, ...) ;
```

Figure 4. Annotation example with control flow.

thread, and the thread that has its atomicity violated the *local* thread. Kivati aims to detect and prevent atomicity violations that occur when a remote thread makes a memory access to a shared variable that violates the atomicity of a pair of memory accesses made by the local thread to the same shared variable. This category of bugs has been the focus of a number of previous atomicity bug testing systems [13, 19]. Figure 2 lists the four interleavings that are non-serializable.

Atomicity violations can be either required, benign or buggy. Atomicity violations can be required for correctness, usually in the cases where inter-thread communication must happen in a certain order. For example, to communicate information between threads, a local thread may initialize a variable (a write), wait for a remote thread to modify it (another write), and then finally use it in some operation (a read). Other atomicity violations can be benign, meaning that program correctness is not affected regardless of whether a violating access occurs. Finally, atomicity violations are buggy if they lead to incorrect program behaviour. Buggy atomicity violations result from programming flaws where the programmer has neglected to enforce atomicity using synchronization. Whether a particular atomicity violation is required, benign or buggy depends on the semantics of the program. If the programmer has used synchronization to enforce a non-serializable interleaving in a required violation, Kivati can detect this and does not enforce atomicity on the violation. However, Kivati cannot differentiate between benign and buggy violations. Thus, to prevent buggy atomic violations, Kivati reorders the accesses of all atomic violations that it cannot identify as required to preserve atomicity. Since atomicity is only enforced if the violation is buggy or benign, Kivati never introduces new synchronization errors. However, Kivati incurs performance overhead from detecting required violations, as well as from unnecessarily enforcing atomicity on benign violations. This overhead is not necessary for Kivati to detect and prevent atomicity violations, and can be reduced by training Kivati to ignore pairs of memory accesses that do not need to be executed atomically.

2.2 The main idea

To detect and prevent atomicity violations, Kivati must be aware of all memory accesses that have the potential to cause violations. It is unreasonable to assume that programmers will correctly annotate these accesses, so Kivati must infer these accesses from the program itself. In addition, these accesses should be identified offline to minimize the run-time overhead of Kivati. Previous testing systems have used profiling runs to identify such memory accesses [13, 19]. However, profiling only exercises a limited set of program paths, and thus cannot exhaustively categorize all memory accesses that could occur in production use. As a result, Kivati uses static analysis on source code to annotate all local memory access pairs whose atomicity could potentially be violated, and uses training to eliminate pairs whose atomicity cannot be violated. This annotated code is then compiled using a standard compiler. At run-time, Kivati will use the annotations to detect and prevent atomicity violations.

Kivati's static analysis phase annotates each pair of memory operations that access the same shared variable. To do this, Kivati first generates a list of shared variables (LSV). Since statically identifying shared variables is not precise, there will be variables in the LSV which are not actually shared. In turn, this means there will be access pairs whose atomicity can never be violated, which results in extra performance overhead, but neither affects the correctness of the program nor the correctness of the atomicity violations that Kivati reports.

Using the LSV, Kivati performs a standard intra-procedural, path-insensitive data-flow analysis (DFA) to find, for each shared variable, all consecutive pairs of memory accesses to it. We call the region of code in between each pair an *atomic region (AR)*. The pair of accesses that define an AR are said to be local with respect to that AR, and the accesses that violate an AR are said to be remote with respect to that AR. For brevity, accesses will simply be referred to as local or remote when discussing an AR, except when it is unclear which AR it refers to. Each AR is associated with a shared variable, and is given a unique identifier. Kivati an-

Arch	Support	Number	Type
x86	Yes	4	After
SPARC	Yes	2	Before
MIPS	Yes	1	Before
ARM	Yes	2	Depends on inst.
PowerPC	Yes	1	After

Table 1. Survey of hardware watchpoint support. The "type" column indicates whether a trap is delivered before or after the instruction that accesses the watched address.

notates each AR with a *begin_atomic* annotation right before the first (local) access to the associated shared variable, and an *end_atomic* annotation right after the second (local) access. The simple example in Figure 3 shows two overlapping ARs to two different shared variables, `shared1` and `shared2`. The DFA would find one pair of accesses for the shared variable *shared1* (the read on line 2 and the write on line 7), and one pair of accesses for the shared variable *shared2* (the read on line 4 and the write on line 9). Kivati would then annotate it as shown in the figure. A more complex example is given in Figure 4. In this case, the DFA would find three pairs of accesses to *shared*: a) the read on line 2 and the write on line 4, b) the write on line 4 and the read on line 8, and c) the read on line 2 and the read on line 8. This illustrates how a memory access, such as the one on line 4, may be both the first access of one AR and the second access of another AR. In addition, it also demonstrates the effect of control flow. Depending on the value of shared, the *end_atomic* on line 12 may be executed without the *begin_atomic* for AR 2 ever being executed. Similarly, if we imagine that the program does not have the access at line 8 outside of the `if()` block, then Kivati would not place the *end_atomic* at line 10. In this case, it is also possible for the *begin_atomic* on line 1 to execute without its accompanying *end_atomic* ever executing. Such incomplete ARs are handled dynamically by Kivati's run-time detection and prevention mechanism.

Kivati detects if a remote access actually interleaves with an annotated local pair in a non-serializable order at runtime. If an access is remote with respect to one AR but is the first local access with respect to another AR, then it is already annotated (because our system would have inserted a *begin_atomic* before it) and thus easily detected by Kivati. While Kivati could also annotate all remote accesses that do not start ARs, this will result in unnecessary annotations and extra overhead. Similarly, Kivati could use the memory management unit to cause traps whenever an access is made to a page that contains a shared variable in an AR, but this will also incur severe performance overhead [3]. Instead, we make novel use of hardware watchpoint support to detect and prevent violations. When entering an AR with a *begin_atomic*, Kivati configures a hardware watchpoint to trap into the operating system if the shared variable the AR is

	Local thread		Remote thread
1	**begin_atomic**(1,&shared ...) ;		
2	shared = 0;		
3	flag = 1;		
4	/* *wait for remote* */		
5	**while** (flag == 1);	1	/* *wait for local* */
6		2	**while**(flag != 1);
7		3	shared = &val;
8		4	flag = 0;
9			
10	/* *read remote value* */		
11	tmp = *shared;		
12	**end_atomic**(1, ...) ;		

Figure 5. Example of a required atomicity violation.

associated with is accessed by another thread. Kivati uses the hardware watchpoint to continuously monitor the variable until the AR completes with a matching *end_atomic*. Since the watchpoints are implemented in hardware, they impose very little or no overhead at all. Hardware watchpoints are supported on most major processor architectures as shown in Table 1.

When Kivati detects a remote access, whether via a watchpoint or a *begin_atomic*, it does not know if an non-serializable interleaving will occur until the second local access occurs and the AR terminates. Kivati conservatively delays all remote accesses that interrupt an AR until after the AR terminates. To do this, Kivati suspends the remote thread before it is about to make its remote access and allows the local thread to execute until the AR completes. Only then does Kivati allow the remote thread to resume execution and perform its memory access. If the current AR is overlapped with another AR, such as AR 1 and AR 2 at line 4 in Figure 4, then the remote thread remains suspended until the shared variable it is accessing is not in any AR. If at the end of the AR, Kivati determines that a non-serializable interleaving has occurred, it records the thread IDs, address of the shared variable and program counters of the memory accesses involved in the interleaving.

As shown in the example in Figure 4, it is both possible for an *end_atomic* to occur without a matching *begin_atomic* and a *begin_atomic* to occur without a matching *end_atomic*. In the former case, Kivati simply ignores the *end_atomic*. The latter case can happen due to two reasons. The first reason is due to control flow, where a *begin_atomic* may be executed, but its accompanying *end_atomic* is not executed. The second is a bit more subtle and is illustrated by the required atomicity violation in Figure 5. Here, the programmer has used `flag` as a synchronization variable to ensure that the local thread waits until the remote thread sets `shared` to a valid value before dereferencing it. Kivati will detect the non-serializable interleaving and suspend the remote thread at line 3 until the *end_atomic* at line 12 in the local thread is executed. Unfortunately, the *end_atomic* can never be executed because the remote thread can never set `flag` to zero and release the local thread from its loop. Both these cases are covered by two mechanisms that Kivati implements. The first is a timeout on the remote thread, which if it expires, resumes the remote thread regardless of whether the AR has completed or not. The second is a *clear_ar* annotation, which is placed at the end of every subroutine and terminates any outstanding ARs that were started in the context of the subroutine. In both cases, prematurely terminating an AR resumes any suspended threads and removes any watchpoints associated with the terminated ARs. If the matching *end_atomic* eventually executes after the timeout, we still record the violation, but note that it was not prevented.

Both Intel and AMD x86 processors support four watchpoint registers, the abundance of which allows Kivati to monitor more ARs simultaneously. However, both AMD and Intel x86 processors trap *after* the memory access has occurred, meaning that the shared variable has already been written to or read from. As a result, to prevent an atomicity violation, Kivati must undo the effects of the instruction making the memory access in order to move it after the AR has completed. Processors that trap before the access simplify the implementation of Kivati since the effects of the memory access do not need to be undone. We will give details on Kivati's mechanism for undoing memory accesses in Section 3.3.

2.3 Prevention mode and bug-finding mode

Up to now, we have described Kivati in prevention mode, where it detects and prevents buggy atomicity violations. At the cost of some additional performance overhead, Kivati can be run in bug-finding mode, which pauses the local thread when it calls *begin_atomic* at the start of an AR. This artificially increases the length of the AR and increases the likelihood that another remote thread will interleave between the *begin_atomic* and *end_atomic* of the AR. This might, for example, be used by a software developer during beta-testing to help find more bugs on realistic workloads.

3. Implementation

In this section we provide details on our Kivati prototype, which supports the x86 family of processors. We will begin by describing the implementation of Kivati's static annotator. Then, we describe Kivati's prevention engine, which is implemented as a set of kernel modifications. The x86 hardware watchpoint registers can only be accessed from ring 0, which means that Kivati's detection component must be implemented in the operating system kernel. Finally, we outline several optimizations which improve the performance of Kivati and describe its limitations.

3.1 Static annotator

Our prototype's static annotator is built using the CIL program analysis framework [17]. The static annotator annotates a given program in two steps. First, it must build the list of shared variables (LSV). Then, it performs data-flow analysis (DFA) to annotate all local pairs of accesses to variables in the LSV.

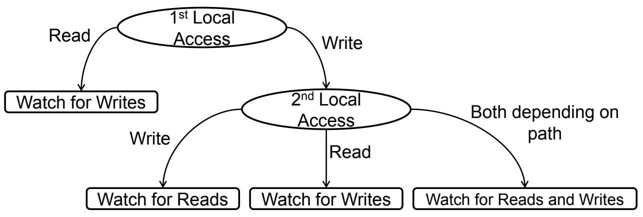

Figure 6. Logic for determining what type of remote access to watch for.

Since Kivati only annotates ARs where both local accesses are in the same subroutine, Kivati constructs an LSV for each subroutine. Kivati begins by seeding each LSV with all global variables. To this it adds any arguments passed in by reference to the subroutine and any pointers returned from a called subroutine. Then, a DFA is performed that adds to the LSV any variable that is data-flow dependent on a shared variable already in the LSV. After the DFA has iterated to a steady state, all variables in the LSV are considered shared variables for the subroutine. As mentioned in Section 2.2, the LSV is an approximation. However, variables in the LSV which are actually non-shared will only be monitored but will never incur an atomicity violation at run-time.

To annotate local pairs as ARs, Kivati constructs a CFG of each subroutine and performs a path-insensitive DFA on the CFG, tracking the program statement and type of each access to variables in the LSV. At the end of each DFA iteration, it forms intra-procedural local access pairs by matching each shared variable access with another access to the same variable that precedes it in the DFA. The operation is conceptually similar to a reaching-definition analysis except that Kivati considers all preceding accesses, not just definitions (i.e., it considers preceding reads as well).

Each pair is then labelled with a *begin_atomic* and an *end_atomic*, which are subroutines that the annotated program will call at run-time before it enters an AR and after it exits an AR respectively. *begin_atomic* takes 5 arguments: a globally unique AR ID, the address of the shared variable, the size of the shared variable, the type of remote access to watch for (read or write) and the type of the first local access. The AR ID is used to identify which atomic region suffered a violation as well as to match *begin_atomic*s and *end_atomic*s. The next three arguments: the address and size of the shared variable, and the type of remote access, are used to configure the hardware watchpoint. The type of remote access to watch for is determined by the type of the two local accesses as shown in Figure 6. In cases where the first local access is paired with a second read and a second write along different paths, Kivati must monitor for both remote reads and remote writes (bottom right in Figure 6). Kivati records the first access type so that in these cases, when an intervening remote access was detected, it can determine whether the remote access actually caused a non-serializable interleaving when it arrives at the second local access and learns which path was taken. *end_atomic* takes two parameters: the second local access type and an AR ID, which will be the same as its paired *begin_atomic*. Finally, a call to *clear_ar* is inserted at every subroutine exit.

3.2 Detecting violations

We will first describe how Kivati detects violations and leave Kivati's method for preventing them to Section 3.3. To detect remote accesses that occur during an AR, Kivati uses hardware watchpoint registers. Both AMD and Intel x86 processors provide four watchpoints, meaning that Kivati can track and detect accesses to four different memory words simultaneously. The watchpoint registers must be configured with the address to watch, the size of accesses to watch for (8, 16, 32 or 64 bits), and the type of access to trap on (e.g., writes).

Kivati requires two new data structures to be added to the kernel. First, Kivati maintains a per-thread *AR table* that records the active ARs that a thread is currently executing in. Second, Kivati maintains *hardware watchpoint metadata*, which records which ARs are using each watchpoint, as well as a list of remote threads that have been suspended as a result of making accesses to addresses being monitored by the watchpoints.

Each core has a set of watchpoint registers, which can only detect violating accesses from the core itself. As a result, the state of the watchpoint registers must be kept consistent across all cores. While one may implement this using inter-processor synchronization, it would impose unnecessary performance overhead. When a core modifies its hardware watchpoint register state due to a thread calling *begin_atomic*, all other cores must update their watchpoint registers to match before the first core can enter its AR. Instead of stalling the first core while other cores update their state, Kivati sets the hardware register on the first core and then causes the thread calling the *begin_atomic* to block, allowing the core to run other threads. In addition, rather than interrupt other cores, the thread remains blocked while other cores opportunistically update their state when dropping into the kernel due to a system call or interrupt. When all other cores have updated their registers, the blocked thread is woken up and can enter its AR.

Both *begin_atomic*s and *end_atomic*s are implemented as system calls that trap into the kernel. On a *begin_atomic*, the kernel component of Kivati first checks if the address is already being monitored by another AR in the same thread or not by examining the watchpoint metadata. If so, it checks the remote access type and size against the watchpoint's metadata, updates them if necessary and adds this AR ID to the list of ARs using the watchpoint. The watchpoint hardware is always set to the most aggressive settings of all ARs using the watchpoint – i.e., the union of all read and write requirements and the largest size requirement. The location and type of access of the *begin_atomic* are also recorded for later use. If no watchpoint register is monitoring the address of the *begin_atomic*, then it checks if there is a free watchpoint, and if so, records the AR ID in the watchpoint's metadata and uses it to monitor the address of the *begin_atomic*. If all watchpoint registers are already used by other threads, then the system call simply exits and Kivati logs that it was not able to monitor the AR due to a lack of watchpoint registers. If Kivati is running in bug-finding mode, *begin_atomic* also pauses the local thread for a configurable period of time to increase the likelihood of an atomicity violation occurring.

If a trap occurs due to a watchpoint, Kivati records the identity of the remote thread making the triggering access in the watchpoint's metadata. However, at this point, the remote access is not an atomicity violation until the matching *end_atomic* of the AR is executed. Thus, Kivati also notes down whether the triggering access was a read or a write.

When an *end_atomic* is executed, Kivati first checks if the watchpoint metadata contains an AR with the same ID as the one passed in by the *end_atomic*. This tells Kivati whether a corresponding *begin_atomic* has been called. If there is no configured watchpoint, Kivati simply returns back to the user program and the *end_atomic* has no effect. If matching AR is found, Kivati checks if there were any watchpoint traps recorded, and if so, compares the type of remote access against the types of the two accesses of each AR to see if a non-serializable interleaving is formed. If so, Kivati logs the information about the violation mentioned in Section 2.2. The AR corresponding to the *end_atomic* is removed from the list of ARs using the watchpoint. If there are no more ARs using the hardware watchpoint after this, it is disabled and marked free. Otherwise, the watchpoint is reconfigured to match the type and size of the remaining ARs using the watchpoint. A *clear_ar* is similar to an *end_atomic* except that it causes all ARs allocated within the current subroutine to be removed from all existing watchpoints. All triggering accesses associated with the removed ARs are also removed. Thus, no atomicity violations can be detected on ARs removed due to a *clear_ar*.

To improve performance and reduce the number of false positives, Kivati can be configured to stop monitoring ARs that have been determined to only have benign atomicity violations. On application startup, Kivati loads an *AR whitelist* from a file that contains a list of benign AR IDs. The contents of this file are stored in memory and checked on every *begin_atomic* and *end_atomic*. If the AR ID is in the whitelist, the *begin_atomic* or *end_atomic* simply returns without entering the kernel. The whitelist file is periodically checked and re-read for updates during execution so that a software developer can send patches to customers to update whitelists for long running processes.

3.3 Preventing violations

To prevent atomicity violations, Kivati reorders remote accesses that cause watchpoint traps to occur after all ARs on the watchpoint have completed. However, this is complicated by the fact that x86 watchpoint traps occur *after* the triggering instruction has completed. As a result, the effects of the remote access will have been committed to the architectural state of the processor by the time Kivati is invoked. To preserve the atomicity of the AR and move the remote access after the AR, Kivati undoes the effects of the remote access and re-executes it after the ARs have completed. To implement prevention, the following actions are added to the actions required for detection on *begin_atomic*s, *end_atomic*s and triggered watchpoints.

When a *begin_atomic* occurs, there are two possibilities. First, if the address of the *begin_atomic* is being watched by one or more ARs in another thread then the current thread is a remote thread that is about to make an access to the same shared variable of the other thread's AR. Kivati suspends the current thread during its *begin_atomic*, thus delaying the first access of this thread's AR until the other thread's ARs have completed. The other possibility is that no other thread is watching the address of the *begin_atomic*. Kivati then enters the AR and monitors the address as described in the preceding section. In this case, Kivati must record the value of the shared variable after the first local access so that it may undo the effects of any interleaving remote accesses. If the *begin_atomic* precedes a read access, Kivati records the value of the shared variable as it enters the AR. If it is a write access, Kivati waits until after the write occurs and then records the value. This is necessary because undoing a remote write requires rolling back to the value before it occurred, which is the value after the first local write of the AR. This is accomplished by setting the watchpoint to trigger on a write access and then recording the value of the shared variable when the local thread's write triggers the watchpoint.

If a remote access causes a watchpoint trap, to prevent the violation, Kivati undoes the effects of the remote access and then suspends the execution of the remote thread until the all local ARs have completed. To undo the effects of the instruction, Kivati moves the program counter back to the instruction that caused the remote access, and then undoes any effects on memory. Because x86 instructions are variable length, Kivati cannot simply move the program counter back a fixed amount. Instead, a pre-processing pass on the binary is used to identify all instructions that access memory and thus could cause a remote access. The program counters of these instructions as well as the program counter of instructions that immediately follow them are recorded in a lookup table and used by Kivati at run-time to move the program counter back to the instruction that caused the watchpoint triggering remote access. A special case is the subroutine `call` instruction, which can cause a remote read access if the argument is an indirect pointer in memory. In this case, the program counter will not point to the instruction immediately following the `call`. To handle this case, we also record the first instruction of every subroutine in a separate list, and if the program counter points to one of these after a watchpoint trap, we know the previous instruction was a `call` instruction and can be found by examining the value stored at the top of the stack and moving back by the size of a `call` instruction.

Kivati also undoes any effects the remote access had on the state of memory. If the remote access is a write, Kivati undoes the write by changing the value of the shared variable back to the value that it recorded after the first local access. If the remote access is a read, then Kivati first determines whether the shared variable was read into a register or another memory location by disassembling the remote access instruction. If it is a register, then we allow the inconsistent value to remain in the register as it will be overwritten with the correct value when the remote access is re-executed after the ARs have completed. However, if it is another memory location, we must ensure that this incorrect value is not "leaked" to another thread. Kivati implements this by configuring another watchpoint register to watch this location. If there are no hardware watchpoints left, then Kivati allows the remote thread to continue and logs that it was unable to reorder this remote access. Finally, any instruction-dependent side effects are also undone. For example, instructions such as `push`, `pop` or `call` also affect the stack register value and this effect is undone accordingly.

After the effects of the remote access have been undone, Kivati suspends the remote thread until all ARs in the local thread complete. However, as mentioned in Section 2.2, Kivati implements a 10 ms timeout in order to avoid deadlocks due to an *end_atomic* which never executes. If the timeout expires before all ARs complete, then the suspended threads are made runnable again. In addition, all ARs using the watchpoint register that timed-out are removed and the watchpoint is freed.

When an *end_atomic* occurs, Kivati will try to restart any threads that are suspended due to remote accesses. Kivati checks if there are any remaining active ARs on the watchpoint. If there are none left, the watchpoint is freed as described earlier and threads are permitted to proceed. Kivati preferentially schedules threads that were blocked due to watchpoint traps before threads that were blocked because they tried to enter their own AR.

3.4 Optimizations

While using the hardware watchpoint support requires domain crossings into the kernel, these crossings are expensive so we would like to minimize their frequency. Transitions into the kernel happen on each *begin_atomic* and *end_atomic* annotation, as well as whenever a watchpoint trap occurs. In this section, we describe four optimizations we have added to Kivati that decrease the number of crossings into the kernel.

First, we do as much pre-processing as possible in user space, and only enter the kernel if we need to modify a hardware watchpoint register. To do this, we replicate both the AR table and watchpoint register metadata in a user space library, which *begin_atomic* and *end_atomic* call instead of directly dropping into the kernel. This allows us to avoid trips into the kernel on a *begin_atomic* when there are no free hardware watchpoints, or if there is a hardware watchpoint already configured with the same address, size and access type as the current *begin_atomic*. On an *end_atomic*, we can also avoid trips into the kernel if a matching *begin_atomic* was not previously executed, or when the hardware watchpoint state doesn't need to be changed because there are still active ARs using the watchpoint to monitor for the same size and types of accesses.

Second, we note that while the remaining trips into *begin_atomic* are necessary to activate the watchpoint monitoring, trips into the kernel due to *end_atomic*s can be completely eliminated. When an *end_atomic* removes the last AR on a watchpoint, the usual procedure is to drop into the kernel and disable the hardware watchpoint. Instead, we simply let the hardware watchpoint continue to watch the address, but note in the user space copy that the watchpoint is no longer active. If another thread does trigger the watchpoint, we drop into the kernel due to the watchpoint trap, learn from the user space copy that the hardware watchpoint should have been freed, and disable the watchpoint at that time. No violation is logged since Kivati is aware that the AR should have been terminated already. However, in a significant number of times, a thread will execute a *begin_atomic* before the watchpoint is triggered and the *begin_atomic* will drop into the kernel anyways. At this point, Kivati will free the hardware watchpoint and make it consistent with the user space copy. The same optimization is applied in cases where an *end_atomic* requires a change to a hardware watchpoint's size or access type. Thus, we save a trip into the kernel whenever a *begin_atomic* occurs after an *end_atomic* but *before* another thread triggers the hardware watchpoint.

The third optimization further reduces the number of unnecessary trips into the kernel by disabling the hardware watchpoints during execution of the local thread that owns the AR using the register. This eliminates the traps that would have otherwise occurred for each AR due to local accesses. However, the optimization introduces a problem for the (local write)-(remote write)-(local read) interleaving. Recall that we relied on the watchpoint to trap into the kernel so that Kivati can record the value of the shared variable after the first local write. With the watchpoint disabled, this trap will no longer occur. Instead, we use our annotation pass to replicate the first local write to also save a copy to a page that is shared between the user space Kivati library and the Kivati kernel component. If and when a remote write causes a watchpoint trap, Kivati will use this copy to undo the remote write.

The fourth and final optimization improves performance by noting that unnecessary traps into the kernel are caused by benign atomicity violations. While we cannot statically identify benign atomicity violations in general, if we assume that the implementation of synchronization functions are correct (i.e. locks, conditional waits, etc...), atomicity violations on synchronization variables are always benign or required. Thus, we can add all synchronization variables to the whitelist (e.g., lock variables, flags).

3.5 Limitations

Our Kivati prototype has several shortcomings due to limitations of our hardware and static analysis. First, older processors (Intel Pentium III or earlier) do not report violations due to `REP MOVS/STOS` instructions until the end of the repetition after the repetition in which the access occurred. As a result, we will not be able to accurately undo and reorder remote accesses caused by these instructions. In addition, all Pentium processors do not report data breakpoints for repeated `INS` and `OUTS` instructions until after the iteration in which the memory was accessed. These two are I/O instructions, and thus should not have any impact on our system as they would not appear in user-level programs.

Second, while x86 processors provide more hardware watchpoints than other architectures we surveyed, they are still insufficient to cover every AR. Our evaluation in the next section shows that Kivati is unable to monitor approximately 5% of ARs for atomicity violations because all of the available hardware watchpoints were already in use. Increasing the number of hardware watchpoints or implementing fine-grained memory protection, such as that proposed in the Mondrian system [24] would help to alleviate this limitation.

Finally, our prototype uses only simple static analysis in its annotator. More precise static analysis can help Kivati detect and prevent more atomicity violations, as well as improve its run-time performance. For example, Kivati could be enhanced to perform inter-procedural analysis to detect ARs that span subroutines, allowing it to detect atomicity violations on such ARs as well. In addition, pointer analysis could be used to better identify shared variables. Better precision in this regard would remove unnecessary *begin_atomic* and *end_atomic* annotations on non-shared variables, reducing performance overhead, as well as making better use of the limited number of hardware watchpoints. In addition, our current analysis only identifies local accesses as belonging to the same shared variable if they use the same variable name. Similarly, instead of labelling individual elements of an array as shared or unshared, we treat an entire array as shared if any element appears to be shared. Pointer analysis will allow us to also identify ARs involving local accesses to the same shared variable that occur due to an alias, as well as produce finer-grain labelling of shared elements in

Application	Workload	Description
NSS 3.12.4 (Firefox)	Included testsuite	Network Security
VLC 1.0.2	Transcoding a video using the x264 codec	Multimedia task
Apache 2.2.13	Webstone 2.5	Web server throughput
Apache 2.2.13 & MySQL 5.1.39	TPC-W	E-commerce website
SPEC2001 OMP	Included inputs	Computational workload

Table 2. Applications and workloads.

arrays. We note that eliminating annotations on non-shared variables does not decrease the number of false positives produced by Kivati, since non-shared variables cannot cause violations at run-time. Reducing false positives can only be reduced by not annotating shared variable accesses whose atomicity can safely be violated. Our system currently uses a whitelist generated from two sources of such knowledge: manual identification of synchronization variables as mentioned in Section 3.4, and training runs as discussed later in Section 4.2. While better static analysis can only improve the performance and violation detection and prevention ability of Kivati, our evaluation in the next section shows that even with the simple static analysis in our prototype, Kivati has reasonable performance and good bug detection and prevention ability.

4. Evaluation

We evaluate two major characteristics of Kivati. First, we evaluate the performance overhead Kivati imposes across a set of threaded workloads. Second, we evaluate how effective Kivati is at detecting and preventing atomicity violations. All experiments are performed on a machine with an Intel 2.13 GHz Core 2 Duo processor, 2 GB of RAM, a 7200 RPM Serial-ATA disk and a Gigabit Ethernet network card. The system was running Ubuntu 8.10 with an SMP-enabled Linux kernel (2.6.27) that has been modified to implement Kivati. The applications and workloads we used are given in Table 2. These include the NSS module in the Mozilla Firefox web browser, the VLC media player, the Apache web server, the MySQL database and the SPEC2001 OpenMP (OMP) benchmark suite. When testing each application, both cores were used. TPC-W is a website simulation that includes both the Apache web server and the MySQL database. In our performance experiments, we had Kivati simultaneously protect both of these applications from atomicity violations. Both server benchmarks, Webstone and TPC-W, measure throughput.

4.1 Performance

To evaluate the performance we ran the workloads described in Table 2 and present the results in Table 3. Bug-finding mode is run with a pause time of 20 ms introduced at every *begin_atomic*. Runtime gives the base execution time of an unmodified application running on a vanilla kernel. The

App	Base	SyncVars	Optimized
NSS	1403	1183 (16%)	821 (41%)
VLC	730	629 (14%)	492 (33%)
Webstone	1114	925 (17%)	608 (45%)
TPC-W	2359	1890 (20%)	1220 (48%)
SPEC OMP	1315	1143 (13%)	788 (40%)

Table 4. Number of domain crossings (*begin_atomic* system calls, *end_atomic* system calls and remote traps) in Kivati. All measurements are given in thousands of system calls per second. A percentage reduction in system calls versus the base implementation is also given.

App	Vanilla	Prevention	Bug
Webstone	492	525 (6.7%)	538 (9.3%)
TPC-W	1000	1112 (11.2%)	1161 (16.1%)

Table 5. Effect of Kivati on latency of requests to server applications. All times are given in milliseconds and percentage overhead is given relative to vanilla.

remaining columns give both overhead of prevention mode (first number) and bug-finding mode (second number) versus the vanilla performance. Base gives overhead of the basic system where every *begin_atomic* and *end_atomic* results in a crossing into the kernel. Null syscall gives overhead with each *begin_atomic* and *end_atomic* modified to return immediately back to the process. SyncVars gives the performance when synchronization variables are added to the whitelist. Finally, optimized is overhead with all optimizations described in Section 3.4 enabled.

The majority of the run-time overhead can be attributed to entering the kernel during *begin_atomic* and *end_atomic*. The Null syscall experiment eliminates all other overhead from tracking state, suspending threads and using the watchpoint hardware. Yet this only reduces overhead by less than 6% in most cases. The largest reduction occurs when all optimizations are applied, which reduces the geometric mean of the overhead from 30% to 19%. The experiments also show that bug-finding mode adds an average of 2.5% overhead with all optimizations enabled. This is due to stalls that threads experience when they execute a *begin_atomic*.

Application	Runtime (s)	Base (%)		Null syscall (%)		SyncVars (%)		Optimized (%)	
NSS	1298	32.4	35.9	25.3	28.4	28.1	32.0	19.7	22.0
VLC	1510	18.0	19.9	14.3	16.1	15.9	17.3	13.0	14.1
Webstone	3000	27.9	29.1	22.6	25.2	25.3	26.3	16.5	19.1
TPC-W	1800	53.7	58.2	40.9	46.3	43.5	49.5	29.5	34.7
SPEC OMP	4800	30.0	33.5	24.6	27.7	27.5	30.3	19.0	21.9

Table 3. Performance of Kivati in both prevention and bug-finding mode. All percentages refer to overhead over a vanilla system. The left number in each column is overhead in prevention mode and the right number is overhead in bug-finding mode.

App	Bug ID	Prev	Bug(20ms)	Bug(50ms)
Apache	44402	66:59	8:01	8:23
	21287	-	13:30	17:20
	25520	-	4:49	7:33
NSS	341323	12:25	2:59	2:05
	329072	1:40	0:16	0:17
	225525	4:41	2:21	3:09
	270689	2:00	0:33	0:56
	169296	-	10:19	7:40
	201134	52:45	9:27	7:33
MySQL	19938	8:53	1:50	1:26
	25306	11:15	2:44	3:20

Table 6. Bug detection and prevention in Kivati. Times are given in minutes and seconds for Kivati to detect and prevent atomicity violations in prevention and bug-finding mode with a 20 ms pause and a 50 ms pause. A "-" indicates that the bug did not manifest after 90 minutes of testing.

Table 4 gives the number of kernel entries in thousands per second under the three levels of optimization. Kernel entries occur from *begin_atomic* and *end_atomic* system calls and remote traps, although the system calls account for over 99.9% of all entries. We can see that the optimizations reduce the number of kernel entries by an average of 41%.

We also tabulate the increase in latency of requests for the two server benchmarks, Webstone and TPC-W, in Table 5. We can see that Kivati increases the latency of each request slightly. This effect is more pronounced in bug-finding mode since each thread is stalled for a period of time whenever it executes a *begin_atomic*.

4.2 Identifying and preventing violations

To evaluate the effectiveness of Kivati at detecting and preventing atomicity violations we constructed a corpus of atomicity violation bugs by searching the bug databases of our open source applications. For each bug, we then ran the application in Kivati and repeatedly applied the inputs that would trigger the bug. Because we do not control the interleavings, it would take several attempts for the bug to manifest. We repeated this for Kivati in prevention mode, as well as bug-finding mode with both a 20 ms and 50 ms pause time. The results of this experiment are tabulated in Table 6.

App	Prevention		Bug-finding	
	FP	Traps/s	FP	Traps/s
NSS	8	16.5	11	19.1
VLC	4	9.9	5	12.0
Webstone	12	21.1	14	23.5
TPC-W	19	30.0	24	32.7
SPEC OMP	5	5.9	7	7.5

Table 7. Number of false positives and rate of watchpoint traps per second.

App	Base	SyncVars
NSS	46.7 (5.7%)	41.8 (5.1%)
VLC	29.4 (6.0%)	25.6 (5.2%)
Webstone	34.8 (5.7%)	29.7 (4.9%)
TPC-W	146.1 (12.0%)	110.9 (9.1%)
SPEC OMP	41.9 (5.3%)	37.7 (4.8%)

Table 8. Thousands of missed ARs per second due to insufficient watchpoint hardware. Also shown is the number of missed ARs as a percentage of the total number of ARs executed.

Kivati was able to detect and prevent every bug when it occurred. Bugs were always found faster in bug-finding mode than in prevention mode, and bug-finding mode was also able to find bugs that did not manifest in prevention mode after 90 minutes of testing. Interestingly, increasing the length of the pause time from 20 ms to 50 ms actually increases the time required to find the bug in just over half the cases. This is because while increasing the pause time increases the likelihood of a violating interleaving, it also slows down the execution of the application.

We also ran Kivati with the performance workloads and recorded the number of false positives that Kivati triggers. The number of false positives is presented in Table 7 along with the rate of watchpoint traps the workloads experience. We considered a false positive to be a unique atomic region that has had at least one violation. This means that even if an atomic region participated in multiple violations, it would only be counted as a single false positive. The false positives are present because Kivati cannot differentiate between

App	Number of Registers Available										
	2	3	4	5	6	7	8	9	10	11	12
NSS	57%	39%	5.7%	3.6%	1.4%	0.32%	0.0007%	0.0001%	0.0001%	0%	
VLC	34%	15%	5.2%	1.6%	0.01%	0.0006%	0%				
Webstone	51%	29%	4.9%	3.0%	0.58%	0.42%	0.027%	0%			
TPC-W	59%	44%	9.1%	6.1%	1.8%	1.0%	0.39%	0.02%	0.001%	0.00008%	0%
SPEC OMP	66%	53%	4.8%	3.5%	1.3%	0.022%	0.001%	0.0006%	0%		

Table 9. Percentage of missed ARs depending on the number of hardware watchpoint registers available.

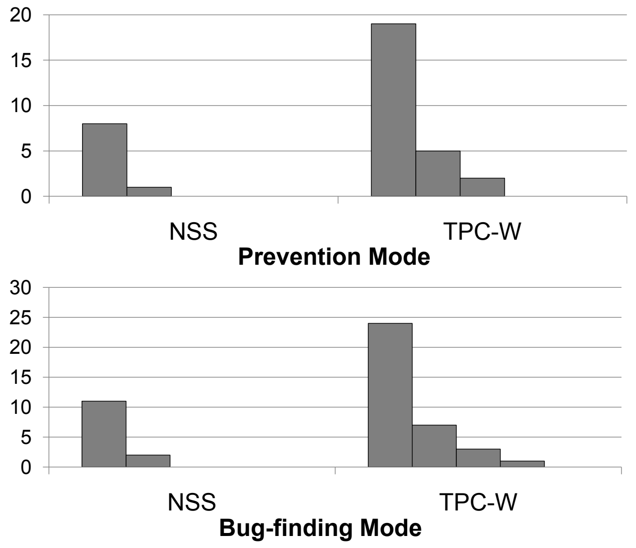

Figure 7. False positives on successive training iterations in prevention and bug-finding mode.

benign and non-benign atomicity violations. As we can see, the number of false positives is manageable. The number of watchpoint traps due a remote access during an AR is also given. Note that the vast majority of these traps do not result in atomicity violations, either because they are serializable or because the *end_atomic* does not occur. However, as compared to the rate of *begin_atomic*s and *end_atomic*s given in Table 4, we see that only a very small number of ARs ever experience a remote access at all. This leads us to believe that Kivati's overhead could be further reduced with more precise identification of shared variables.

Because the x86 processor has only four hardware watchpoints, there are instances where Kivati cannot monitor an AR because all watchpoints are already used by other active ARs. As a result, any violation of such an AR will be missed. Table 8 shows the number of such "missed ARs" as a rate of thousands per second and as a percentage of ARs executed. While the absolute number seems high, it is actually a small percentage of the total number of ARs executed. When we implement measures that reduce the number of ARs that Kivati must monitor, such as whitelisting synchronization variables or increasing the precision of our static analysis, the number of missed ARs decreases. Other optimizations that reduce the number of transitions into the kernel, but not the number of ARs, have no effect.

Another factor that affects the number of missed ARs is the number of hardware watchpoints. We simulate the number of missed ARs for an arbitrary number of hardware watchpoints by varying the number of entries in the hardware watchpoint metadata between 2 and 12. Table 9 shows that a minimum of 4 hardware watchpoints is desirable, as the number of missed ARs increases rapidly with less than that. However, an architecture with 8 hardware watchpoints would be able to monitor all but 1% of all ARs. A long tail in one of the benchmarks causes the number of required hardware watchpoints to monitor all ARs in our benchmarks to be 12.

Finally, we evaluated the effect of training on the number of false positives. We iteratively ran the NSS and TPC-W workloads. After each iteration, we took the false positives recorded and added them to the whitelist for the next iteration. Figure 7 compares the results in both prevention and bug-finding mode. Each bar represents an iteration and its height represents the number of false positives found during that iteration. We see that it took an extra iteration for TPC-W to reach zero false positives in bug-finding mode, but overall we converge to a small number of false positives with very little training. In addition, the results naturally suggest that bug-finding mode should be used for training since it detects and eliminates more false positives than prevention mode – 13 vs 9 for NSS and 35 vs 26 for TPC-W.

5. Related Work

A number of systems focus on finding and detecting atomicity violations during testing, so that the software developer may fix them before releasing the code. Some systems do not execute the code and just perform static analysis [4, 7]. However, a good majority have some dynamic component. For example, Atomizer [5], automated type analysis [21] and Velodrome [8] detect potential atomicity violations by using Lipton's theory of reduction [12] or happens-before relations [11]. However, these systems execute with 2.2x-72x slowdown. In addition, they only provide limited information on how the bug manifests, making it difficult for a software developer to reproduce the bug and correct it. Kivati is able to provide a detailed trace with the thread IDs, address of the shared variable and program counters of the instructions involved.

Other testing systems are also able to provide exact information about the interleaving required to produce a bug. For example, SVD [25], AVIO [13], CTrigger [19], CHESS [16], Racefuzzer [22] and FastTrack [6] all execute programs and check interleavings during execution for violations. The last three systems focus on data-race errors and not atomicity violations. However, all of these systems rely on expensive instrumentation to detect atomicity violations, which cause them to have worst-case overheads of 15x-65x without the addition of specialized hardware support. Unlike Kivati, these systems do not prevent the atomicity violation from occurring once detected. In general, Kivati complements concurrency bug testing systems. If a certain sequence of accesses has been tested and shown not to have any atomicity violations, it can be placed in Kivati's whitelist, which reduces the overhead and false positives that Kivati will experience during run-time. On the other hand, no testing system will find all bugs, so Kivati can prevent the ones that the testing systems do not catch.

Recently, there have been a number of systems that can dynamically detect and prevent concurrency bugs. For example, Gadara [23] and Dimmunix [10] both detect and prevent deadlocks, but not atomicity violations. Rx [20] can detect and prevent a wider range of bugs, but does so probabilistically by repeatedly executing the buggy section of code until the bug does not manifest. In contrast, Kivati can prevent every atomicity violation it encounters.

Other systems take a different approach and try to constrain execution to interleavings that are known to be safe. Constrained interleaving [26] profiles the application initially to find a set of "safe" interleavings, and then only allows these interleavings to be executed at run-time. When implemented without hardware support constrained interleaving has overheads of 100x-200x. Other systems, such as Kendo [18] and DMP [2] constrain programs to execute according to a pre-determined interleaving, thus removing the non-determinism that makes it difficult to find bugs. DMP requires specialized hardware but Kendo, which is implemented in software, has overheads comparable to Kivati. However, Kendo is restricted only to programs that use locks on all accesses to shared variables. Such programs would not benefit from Kivati since they are likely to already be free of atomicity violations.

Finally, other systems that require specialized hardware support can provide protection similar to Kivati's with lower overhead. Atom-Aid [15] requires hardware that supports bulk execution [1] and can only probabilistically remove atomicity violations. However, this hardware is not available on commodity processors. Transactional memory hardware [9] also prevents atomicity violations by ensuring that all instructions in a transaction execute atomically. However, this requires that the software developer correctly insert transactions into their code, while Kivati does not rely on any programmer annotations. Kivati's ARs are conceptually similar to transactions, but much simpler to implement. A transactional memory system has to monitor every memory location accessed in a transaction for conflicts, while Kivati only has to track the address specified by the *begin_atomic* that starts the AR.

6. Conclusion

Kivati's application of static analysis to identify ARs, use of hardware watchpoints to monitor ARs and implementation in the kernel enable Kivati to detect and prevent atomicity violations efficiently for applications written in C and executed on Linux/x86 platforms. The dominant source of overhead in Kivati are transitions into the kernel, which occur when Kivati needs to reconfigure the watchpoint hardware. As a result, optimizations that enable Kivati to avoid trips into the kernel by performing more processing in a user space library result in the greatest performance improvements.

Kivati has two complementary modes – prevention mode, which has low overhead, and bug-finding mode, which imposes more overhead, but can find more bugs and find them in less time. Bug-finding mode is particularly useful during training, because it is able to find and remove more false positives. An interesting result is that increasing the pause times in bug-finding mode does not always result in faster bug detection since it also slows down application execution.

Finally, we note that Kivati could benefit from a more sophisticated static analysis than the simplistic one used in our prototype. A static analysis that can more precisely identify shared variables will remove ARs for non-shared variables. A smaller number of ARs benefits Kivati by reducing its performance overhead, reducing the number of false positives, and reducing the number of instances where it exhausts hardware watchpoint resources. Despite this, we have found that Kivati's simple static analysis is still adequate to produce good performance, good bug detection and prevention, and a low number of false positives.

Acknowledgements

We would like to thank our shepherd Michael Hohmuth and the anonymous reviewers for their insightful comments and guidance. This work also benefited from suggestions by Ashvin Goel, Michael Stumm, Angela Demke Brown, Stan Kvasov, and Tom Hart. This work was supported in part by an NSERC Discovery Grant and a Bell University Grant. Lee Chew was supported by an Ontario Graduate Scholarship.

References

[1] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 278–289, June 2007.

[2] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, Mar. 2009.

[3] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, pages 121–130, Mar. 2007.

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, Oct. 2003.

[5] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, Jan. 2004.

[6] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, June 2009.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the 2003 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, June 2003.

[8] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 293–303, June 2008.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 289–300, May 1993.

[10] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 295–308, Dec. 2008.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] R. J. Lipton. Reduction: a new method of proving properties of systems of processes. In *Proceedings of the 2nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 78–86, Jan. 1975.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, Oct. 2006.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, Mar. 2008.

[15] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 277–288, June 2008.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Dec. 2008.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *CC'02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[18] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, Mar. 2009.

[19] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36, Mar. 2009.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies — a safe method to survive software failure. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 235–248, Oct. 2005.

[21] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 83–94, June 2005.

[22] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, June 2008.

[23] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 281–294, Dec. 2008.

[24] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316. ACM, 2002.

[25] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, June 2005.

[26] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 325–336, July 2009.