Dynamically Detecting and Tolerating IF-Condition Data Races

Shanxiang Qi,[*] Abdullah A. Muzahid,[†] Wonsun Ahn, Josep Torrellas
University of Illinois at Urbana-Champaign
sqi@google.com muzahid@cs.utsa.edu dahn2,torrella@illinois.edu
http://iacoma.cs.uiuc.edu

Abstract

An IF-Condition Invariance Violation (ICIV) *occurs when, after a thread has computed the control expression of an IF statement and while it is executing the THEN or ELSE clauses, another thread updates variables in the IF's control expression. An ICIV can be easily detected, and is likely to be a sign of a concurrency bug in the code. Typically, the ICIV is caused by a data race, which we call* IF-Condition Data Race (ICR).

*In this paper, we analyze the data races reported in the bug databases of popular software systems and show that ICRs occur relatively often. Then, we present two techniques to handle ICRs dynamically. They rely on simple code transformations and, in one case, additional hardware help. One of them (*SW-IF*) detects the races, while the other (*HW-IF*) detects and prevents them. We evaluate* SW-IF *and* HW-IF *using a variety of applications. We show that these new techniques are effective at finding new data race bugs and run with low overhead. Specifically,* HW-IF *finds 5 new (unreported) race bugs and* SW-IF *finds 3 of them. In addition, 8-threaded executions of SPLASH-2 codes show that, on average,* SW-IF *adds 2% execution overhead, while* HW-IF *adds less than 1%.*

1. Introduction

Parallel programming often leads to spending substantial time trying to find and fix concurrency bugs. Examples of such bugs include deadlocks, data races, atomicity violations, and ordering violations. It is therefore important to find new approaches to find these bugs in an effective and inexpensive manner.

One approach that has been used in the past is to look for code patterns that both are easy to identify in the source code and have a high probability of being involved in a bug. Such patterns are often related to synchronization operations. For example, grabbing two locks in different orders in different locations in the code is often a sign of a deadlock bug [5]. As another example, grabbing the same lock multiple times in the same function is often a sign of an atomicity violation bug [24].

In this paper, we focus on IF statements and apply a similar approach. IF statements are very easy to recognize in the source code. In addition, we note that, after a thread has computed the control expression of the IF statement and is executing the THEN or ELSE clauses, no other thread should update the variables in the control expression. If this happens, it is likely to be a bug. The rationale is that the statements in the THEN and ELSE clauses are control dependent on the IF control expression. Hence, the program is likely to malfunction if the value of the expression does not hold until the completion of the IF statement. In other words, it is likely that the programmer implicitly assumed that the value of the control expression remains invariant while the THEN and ELSE clauses execute. An update to a location accessed in the control expression breaks such invariance. Consequently, we define an *IF-Condition Invariance Violation (ICIV)* as "a pattern where, after a thread (*T1*) has computed the control expression of an IF statement and while it is executing the THEN or ELSE clauses, another thread (*T2*) updates variables in the IF's control expression".

The typical way for thread *T2* to update variables in the IF's control expression is through a data race on the variables — although this is not the only way, as we will see. Consequently, we define an *IF-Condition Data Race (ICR)* as "a data race where a memory location accessed by a thread (*T1*) in the control expression of an IF statement is updated by a racy write from *T2* while *T1* is executing the THEN or ELSE clauses". Note that *T1* may or may not access the location again in the THEN or ELSE clauses. Figure 1 shows an example of ICR.

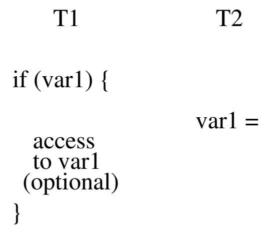


Figure 1: Example of an IF-condition data race (ICR), where time progresses from top to bottom. Note that *T1* may or may not access the racy location again in the THEN or ELSE clauses.

If one considers the IF's control expression to be the "Check" and the THEN or ELSE clauses to be the "Action" of the IF statement, ICRs share a similar pattern with what are called TOCTTOU (time of check to time of use) [36, 38] races. The difference is that while TOCTTOU races apply to conditions in the file system between competing processes, ICRs apply to memory accesses of multi-threaded programs.

ICR detection can be done in an extremely efficient and lightweight way. Since ICRs are associated with a particular program structure, they are easy to spot and check with a fast and specific test. There is no need for profiling or training runs to find the targets for the checks — a simple compiler analysis of the code structure is typically enough. In addition, these races are likely to be especially harmful races because, intuitively, they break the logic of the IF statement and are unlikely to be

[*] Shanxiang Qi is now with Google Inc., Mountain View, CA.
[†] Abdullah A. Muzahid is now with the University of Texas, San Antonio, TX.

placed by the programmer on purpose.

In this paper, we analyze the data races reported in the bug databases of popular software systems and find that ICRs occur relatively often. We present two techniques to handle them. One technique is entirely software-based and detects ICRs during the code testing phase (*SW-IF*). The other relies on additional hardware and is able both to detect and prevent ICRs during production runs (*HW-IF*).

Our techniques use the compiler to identify IF statements with control expressions that contain accesses to shared locations. Then, in *SW-IF*, the compiler inserts code to check that the value of the expression has not been changed by another thread. The check is performed in the THEN and ELSE clauses, either before the first local write to one of the locations in the control expression or, if there is no such write, at the end of the clauses. In *HW-IF*, the compiler inserts code to "watch" the shared locations that participate in an IF's control expression. During the IF's execution, the local processor can access the watched locations, but any remote processor that attempts to do it gets a failed access signal and has to retry. This simple mechanism prevents other threads from causing ICRs.

We evaluate *SW-IF* and *HW-IF* using a variety of applications. We show that these techniques are effective at finding new data race bugs with very few (or harmless) false positives, and run with low overhead. Specifically, *HW-IF* finds 5 new (unreported) ICR bugs in the Cherokee web server and the Pbzip2 application; *SW-IF* finds 3 of them. In addition, 8-threaded executions of the SPLASH-2 applications show that, on average, *SW-IF* adds 2% execution overhead, while *HW-IF* adds less than 1%.

The contributions of this paper are: (1) defining a new invariant whose violation is likely to be a bug (IF-Condition Invariance Violation or ICIV) and identifying the new, common type of data race that violates it (IF-Condition Data Race or ICR); (2) proposing two new techniques to detect and prevent ICR bugs; and (3) showing that these techniques find several new bugs with very few (or harmless) false positives, and little execution overhead.

This paper is organized as follows: Section 2 provides a background; Section 3 examines ICIVs in detail and their relation to races; Section 4 characterizes ICRs; Sections 5 and 6 present *SW-IF* and *HW-IF*; Section 7 shows the potential of these techniques; Section 8 evaluates them; Section 9 discusses related work; and Section 10 concludes.

2. Background on Race Detection

Data races are one of the most common concurrency bugs. A data race occurs when two threads access the same memory location without any intervening synchronization and at least one of the accesses is a write. Data race debugging can be very hard and, therefore, the topic has received much attention (e.g., [2, 7, 8, 9, 17, 20, 21, 23, 25, 26, 27, 31, 32, 34, 37, 43, 44]). The work includes the proposal for software tools for race detection (e.g., [2, 31, 32, 34, 43]) and special hardware structures in the machine (e.g., [8, 17, 19, 26, 27, 44]).

In general, there are two approaches to find data races: lockset algorithms, such as in Eraser [32], and happened-before algorithms, such as in Thread Checker [2]. The lockset approach is based on the idea that all accesses to a given variable should be protected by a common set of locks. This approach tracks the set of locks held while accessing each variable. It reports a violation when the currently-held set of locks (lockset) at two different accesses to the same variable have a null intersection.

The happened-before approach relies on identifying concurrent epochs. An epoch is a thread's execution between two consecutive synchronization operations. Each thread uses a vector clock to order its epochs relative to the other threads' epochs. In addition, each variable has a timestamp that records at which epoch it was last accessed. When a thread accesses the variable, it compares the variable's timestamp to its own clock, to determine the relationship between the two corresponding epochs: either one happened before the other, or the two overlap. In the latter case, we have a race.

Thanks to all this work, the state of the art in data race debugging has made giant strides in the last decade. Unfortunately, commercial race-detection tools (e.g., [2, 34]) still suffer from several limitations. Two of the most vexing ones are the lack of specificity and the high runtime overhead.

The first issue refers to the lack of focus on the key data races. If we run a commercial race-detection tool on a large software system, we typically obtain a long list of data races. While it can be argued that all data races are undesirable, for productivity reasons, it is imperative to focus the human's attention on those races that truly cause program malfunctioning — at least first.

The high runtime overhead — often 100x or more — burdens the program developer, who needs to run the race detector multiple times during development. The overhead results from the tool's desire to provide a complete analysis. Recently, there have been proposals for race detectors that use program sampling (e.g., [4, 16]) or hardware support (e.g., [8, 19, 44]). These are promising approaches, although they come with shortcomings in race detection ability or hardware cost.

In this paper, we focus on a type of data race that is very easy to find, since it can be detected by looking for IF statements in the source code. In addition, it is likely to be a especially harmful race since, intuitively, it breaks the logic of the IF statement and is unlikely to be placed by the programmer on purpose.

More discussion on related work is presented in Section 9.

3. IF-Condition Invariance Violations & Races

We define an *IF-Condition Invariance Violation (ICIV)* as "a pattern where, after a thread (*T1*) has computed the control expression of an IF statement and while it is executing the THEN or ELSE clauses, another thread (*T2*) updates variables in the IF's control expression". We consider this pattern a likely bug deserving of programmer attention because the THEN and ELSE clauses are control dependent on the IF control expression. As a result, it is likely that the programmer implicitly assumes that the condition tested by the IF statement remains unchanged while these clauses execute.

Application	# Reported Data Races	Bug ID	# IF-Condition Data Races	Language
Apache	26	25520,21287,45605, 49986-1, 49986-2, 49985, 47158, 48790, 1507, 31018, 45608, 44178, 254653, 49972, 40681, 40167, 728, 41659, 37458, 36594, 37529, 40170, 46211, 44402, 46215, 50731	20	C, C++, Java
MySQL	13	644, 791, 2011, 3596, 12848, 52356, 19938, 24721, 48930, 42101, 59464, 56324, 45058	8	C, C++
Mozilla	11	622691, 341323, 13377, 225525, 342577, 52111, 224911, 325521, 270689, 73761, 124923	8	C, C++
Redhat (glibc)	2	2644,11449	0	C
Java SDK	1	6633229	1	Java
Pbzip2	1	Data race from [40]	1	C++
Total	54	—	38	—

Table 1: Data races studied. They are mostly obtained from the bug databases of popular software systems.

In Section 1, we said that the obvious way to induce an ICIV is through a data race, which we call *IF-Condition Data Race (ICR)*. However, there are also non-racy ICIVs. They appear when the IF control expression is enclosed in a critical section, *T1* releases the lock after computing the expression and, before *T1* completes the THEN or ELSE clauses, another thread (*T2*) acquires the lock and updates variables in the expression.

Figure 2 shows two examples of non-racy ICIVs. In Figure 2(a), *T2* changes the value inside a critical section while *T1* is still executing the IF statement; Figure 2(b) is like (a) except that, in addition, *T1* reads the changed value. In neither case is there a data race.

```
        T1              T2
  if (var1) {
    unlock(L)
                      lock(L)
                      var1 = !var1
                      unlock(L)



  }
              (a)
```

```
        T1              T2
  if (var1) {
    unlock(L)
                      lock(L)
                      var1 = !var1
                      unlock(L)
    lock(L)
        = var1
    unlock(L)
  }
              (b)
```

Figure 2: Examples of non-racy ICIVs.

While these patterns are likely indications of bugs, in this paper, we focus on *racy* ICIVs, namely ICRs. We experimentally find and characterize ICRs.

ICIVs are not the same as what is conventionally referred to as atomicity violations. Specifically, a single-variable atomicity violation requires that, in between two accesses to the same variable by a thread *T1*, a second thread *T2* accesses the variable. Many ICIVs are not single-variable atomicity violations because, as shown in Figure 1, *T1* may not access the variable for the second time. Consider now multi-variable atomicity violations. For example, a two-variable atomicity violation requires that *T1* accesses two variables (say with *rd x* and *rd y*) and, logically in between, another thread accesses these variables. For example, if *T2* only executes *wr x* in between, this is not an atomicity violation because *wr x* can be logically moved after *T1*'s *rd y*. Instead, *T2* has to execute *wr x* and *wr y* in between for it to be a two-variable atomicity violation. Consequenty, ICIVs are not necessarily multi-variable atomicity violations either. Conversely, many atomicity violations are not ICIVs because ICIVs require a particular pattern associated with an IF statement.

4. An Analysis of IF-Condition Data Races

To get an idea of whether or not ICRs are common, we collected and analyzed data race bugs. Specifically, we took the 9 data races from Yu's list [40], and also mined the bug report databases of Apache, MySQL, Mozilla, the glibc library of Redhat, and JAVA SDK. From the bug report databases, we did our best effort to collect the bugs that met the following conditions: 1) programmers used the words "race condition" in the description of the bug and 2) it was relatively easy to pinpoint the race in the source code according to the description. The resulting 54 data race bugs obtained are listed in Table 1.

From this list, we identify those that are *IF-Condition Data Races (ICRs)*. An ICR is "a data race that occurs when a memory location accessed by a thread (*T1*) in the control expression of an IF statement is updated by a racy write from *T2* while *T1* is executing the THEN or ELSE clauses". Note that *T1* may or may not access the location again in the THEN or ELSE clauses. Figure 1 shows an example.

As shown in Table 1, we find that, of all these races, 38 (or 70%) are ICRs. This percentage should not be taken as representing an average that is valid across a wide range of applications — since we did not exhaustively analyze all the data races in the databases. Instead, it is an indication that ICR bugs are common and, therefore, deserve to be targeted by (inexpensive) debugging techniques. In addition, 6 out of these 38 ICR bugs (or 16%) involve 2 racing variables. These bugs are all double-checked lock (DCL) [33] races.

We now present two new techniques, *SW-IF* and *HW-IF*, which are best efforts to detect and prevent ICRs. Strictly speaking, *SW-IF* and *HW-IF* detect and prevent ICIVs but, in this paper, we are interested in racy ICIVs — that is, ICRs. Both *SW-IF* and *HW-IF* rely on a compiler pass that identifies IF statements with control expressions that include shared variables.

5. SW-IF: Detecting IF-Condition Data Races

5.1. Main Idea

SW-IF attempts to find ICRs by inserting code at the end of the THEN and ELSE clauses of the IF statement, that checks whether the value of the control expression has changed. If it has, it is a very good indication that another thread wrote to the variables in the expression. Sometimes, the compiler is forced to insert the check earlier, before the THEN or ELSE ends, because local writes modify or can modify the value of the control expression. We call the locations in the program where the compiler inserts these checks *Confirmation* points.

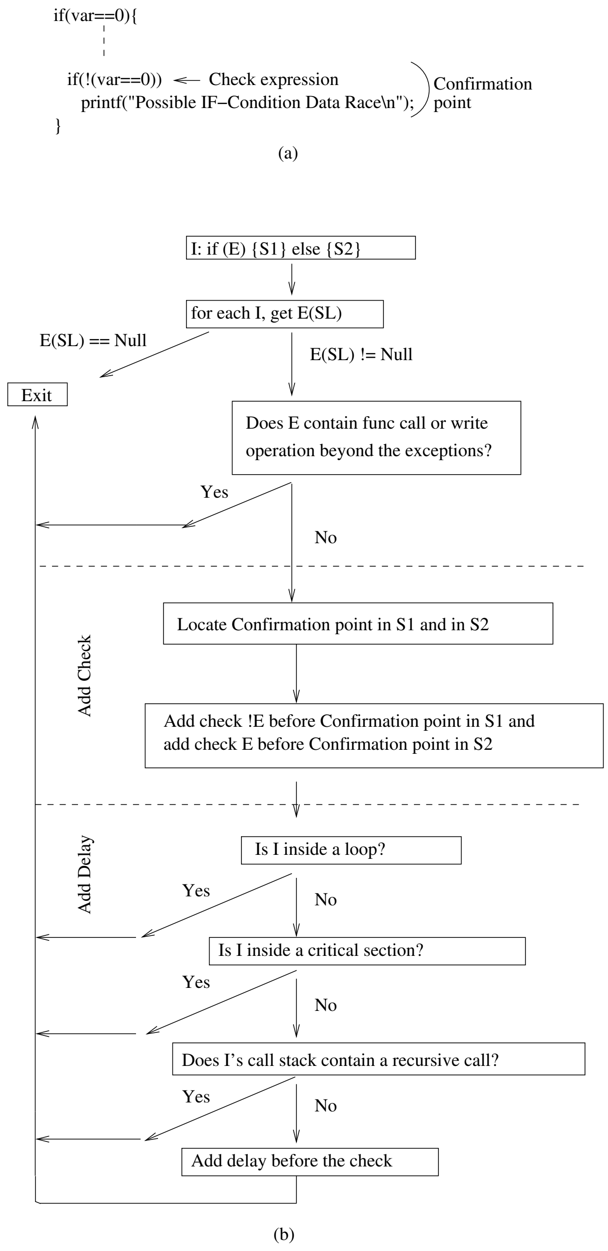
More formally, for a given IF statement, call *E* the control expression, *E(SL)* the set of potentially-shared locations accessed in *E*, and *E(L)* the set of all locations accessed in *E*. We only analyze an IF statement if its *E(SL)* is not empty. In this case, the compiler sequentially searches each of the THEN and ELSE clauses for any statement that might potentially perform a write to *E(L)*. If it finds any, the Confirmation point is placed before the first such write. If the compiler cannot find a candidate write, the Confirmation point is placed at the end of the clause. There is at most one Confirmation point *in each* of the THEN and ELSE clauses. This is so to reduce overhead.

At a Confirmation point, the compiler inserts code that recomputes *E*. If the logic value of *E* is different than the logic value *E* had when it was first executed in the IF's control expression, then a race bug is declared. Figure 3(a) shows the idea. Note that, to reduce overhead, we do not insert synchronization operations around the recomputation of *E*.

Like DataCollider [10], this approach is a best-effort one. This scheme may miss an ICR because the Confirmation point is placed earlier than at the end of the THEN or ELSE clause, and the data race happens between the two points. It can also miss the race because, although some variables in *E* changed, expression *E* returns the same logic value. In the following, we present the algorithm, and then discuss the limitations.

5.2. Algorithm

We use the Cetus source-to-source compiler [13] to analyze and transform the code. The structure of our transformation algorithm, called *SW-IF*, is shown in Figure 3(b). The algorithm is performed in two steps: "Add Check" and "Add Delay". "Add Check" identifies the Confirmation points and inserts the checks. This is the only step used if we want to run *SW-IF* with minimal overhead. "Add Delay" is a pass that is useful if we want to force different program interleavings and we can tolerate more overhead.



```
if(var==0){
  ...
  if(!(var==0))
    printf("Possible IF-Condition Data Race\n");
}
```

Check expression
Confirmation point
(a)
I: if (E) {S1} else {S2}
for each I, get E(SL)
E(SL) == Null
E(SL) != Null
Exit
Does E contain func call or write operation beyond the exceptions?
Yes
No
Add Check
Locate Confirmation point in S1 and in S2
Add check !E before Confirmation point in S1 and add check E before Confirmation point in S2
Add Delay
Is I inside a loop?
Yes
No
Is I inside a critical section?
Yes
No
Does I's call stack contain a recursive call?
Yes
No
Add delay before the check
(b)

Figure 3: High-level structure of the *SW-IF* algorithm.

5.2.1. Adding Checks. In the first step, the compiler needs to identify all the IF statements that will be augmented with Confirmation points. We call this set the *Monitor* set. Such a set starts with all the IF statements that have control expressions that potentially access shared locations. Cetus is able to tell when accesses reference provably private locations. IF statements where *E* contains only accesses to provably private locations are not part of the Monitor set.

SW-IF removes from the Monitor set those IF statements where *E* includes writes. *SW-IF* does not support writes because, otherwise, when it recomputes *E* at a Confirmation point, it would have side effects. The only exception to this rule is when *E* only contains the ++ or - - operator, and Cetus can prove that there is no aliasing. In this case, the IF statement is kept in the Monitor list, and *E* will be recomputed at the Confirmation point without the ++ or - - operation.

SW-IF also removes from the Monitor set those IF statements where *E* contains a function call. *SW-IF* does this to avoid unwanted side effects at Confirmation points, or the need to perform expensive interprocedural analysis to analyze function side effects. The exceptions are C standard functions that do not write variables, such as string compare (strcmp and strncmp) and absolute (abs). In general, neither writes nor procedure calls are common in control expressions.

For each IF statement in the Monitor set, *SW-IF* finds the Confirmation points of the THEN and ELSE clauses. Then, it inserts there the recomputation of *!E* and *E*, respectively. Finding the Confirmation point boils down to finding the first potential write to *E* within the THEN or ELSE clause and for this, we rely on the alias analysis provided by Cetus. We make an exception for function call statements however. Limitations of Cetus in analyzing the side effects of function calls cause the placement

```
for(i=0; i<MAX; i++){

   if (var==0){
      Do not add delay here.
      if (!(var==0)) ...
   }
}
         (a)
```

```
Lock (l)
  if (var==0){
      Do not add delay here.
       if (!(var==0)) ...
  }
 Unlock (l)
         (b)
```

```
foo (n){            bar(){
 bar();               if (var==0){
 foo(n−1);               Do not add delay here.
}                        if (!(var==0)) ...
                      }
                    }
         (c)
```

Figure 4: Three types of IF statements where *SW-IF* does not insert delays.

Mode	False Negatives (Failure to Signal IF-Condition Race)	False Positives (Incorrectly Signal IF-Condition Race)
SW-IF	Occasional: • Racing writes update E but do not change the overall logic value of E • Potential local writes to E force Confirmation point early and race happens later • Unsupported writes or functions in E prevent insertion of Confirmation point and race happens	Rare: • Before the Confirmation point, the local thread modifies the logic value of E without the compiler being able to analyze it. • Certain non-racy IF-Condition Invariance Violations
HW-IF	Rare: • Inability to watch the side effects of functions called in E and race happens there • Simplified support for monitoring nested IFs • Race under special conditions: breaking a deadlock and preempting a thread	Harmless: • Due to simplified AWT hardware: "False sharing" data races and external read conflicts • Non-racy IF-Condition Invariance Violations

Table 2: False positives and false negatives in *SW-IF* and *HW-IF*.

of Confirmation points to be too conservative if placed before every call that might potentially update *E*. Hence, we insert a Confirmation point before only the function calls that have a high probability of updating *E*, that is, functions with arguments that contain addresses of or references to variables in *E*. Also, if the IF statement includes a loop and the loop contains a potential write to *E*, *SW-IF* places the Confirmation point before the loop. If no potential writes are found, the Confirmation is inserted at the end of the THEN or ELSE clause.

5.2.2. Adding Delays. In a development scenario, we may want to precede each Confirmation point check with a small sleeping delay, so that we can potentially force a different interleaving and uncover a bug. Consequently, in the "Add Delay" step, *SW-IF* selects the IF statements within the Monitor set that can additionally be instrumented with delays. We call the resulting set the *Delay&Monitor* set. To obtain the Delay&Monitor set, *SW-IF* removes three types of IF statements from the Monitor set (Figure 4). Adding delays in these IF statements could potentially add too much overhead.

The first two types of IF statements to remove are those inside loops or critical sections. We use Cetus' intermediate representation tree to check interprocedurally if there are loops or critical sections enclosing the IF statement. The third type of IF statements to remove are those whose call stack contains a recursive function. This case is identified using the strongly-connected component analysis in Cetus.

5.3. Limitations

SW-IF is attractive because, while being an entirely software-only solution, the impact on execution speed is almost negligible, allowing it to be used extensively with many input sets and test cases. It can be left on during the entire testing phase with the programmer barely noticing any slowdown. However, it may suffer from false negatives (i.e., miss an ICR) and, in unusual cases, false positives (i.e., incorrectly declare an ICR). The top row of Table 2 summarizes the sources of ICR false negatives and false positives for *SW-IF*.

There are three scenarios where *SW-IF* can suffer false negatives. One is when racing writes update one or more locations in *E* but the overall logic value of *E* remains the same. *SW-IF* chooses to check the logic value of *E* only because it assumes that the programmer only relies on it being invariant. The second scenario is when a potential local modification of *E* forces *SW-IF* to place the Confirmation point early, and the data race happens between that point and the THEN/ELSE clause end. The final scenario is when unsupported writes or function calls in *E* prevent *SW-IF* from inserting a Confirmation point and a data race happens. False negatives caused *SW-IF* to miss some ICR bugs in the applications we tested.

False positives occur in two cases. First, they occur when the thread executing the IF statement updates the logic value of *E* before the place in the code where *SW-IF* puts the Confirmation check. In this case, the check will incorrectly declare a data race. This case can occur due to function side effects in function calls before which Cetus chose not to insert a Confirmation point. Figure 5 shows such an example, where F modifies *p as a side effect, and the check in the end signals an ICR. The second case occurs in certain non-racy ICIVs, such as in those shown in Figure 2. Overall, we found these two cases to be very rare: in our experiments with real applications, *SW-IF* did not experience any false positives.

```
If (*p ==q){
   F(); // *p =!q in F
   Confirmation point
}
```

Figure 5: Example of false positive in *SW-IF*.

Note that, since *SW-IF* does not place synchronizations in Confirmation points, it can insert a racing read in non-racy ICIVs, such as those in Figure 2. Finally, *SW-IF* is also limited in that it can only detect, not prevent, ICRs. By the time it detects the race at the Confirmation point, the race has already happened. *SW-IF* can only raise an exception or print an error message. To address the limitations of *SW-IF*, we now propose *HW-IF*, which augments it with some hardware support.

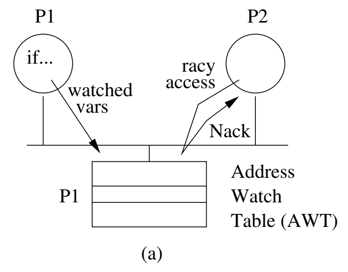
6. HW-IF: Detecting & Preventing ICRs

6.1. Main Idea

In *HW-IF*, the compiler searches for IF statements that can cause ICRs as before. But this time, instead of Confirmation points, the compiler inserts, right before the IF statement, code to "watch" all the shared locations that are accessed in the control expression. The hardware envisioned has a functionality that extends the current watchpoints provided by x86 processors [11]. Specifically, we envision that the local processor can still access the watched locations. However, any remote processor that attempts to access them will get a failed memory access signal that will prompt the hardware to retry the access — transparently to the software. At the end of the IF statement, the compiler inserts code to stop watching the memory locations. At that point, the accesses from the other processors will succeed. In this way, *HW-IF* detects and prevents ICRs.

6.2. Operation of HW-IF

A simple design of *HW-IF* is shown in Figure 6(a). For simplicity, we consider a bus-based multicore, although more scalable organizations can be designed. As part of the bus controller, there is a hardware table called the Address Watch Table (AWT). Each AWT entry contains information about one watched memory location: it contains the address of the cache line containing the watched location and the ID of the "owner" processor.



```
Watch(shared var1 in Exp)
Watch(shared var2 in Exp)
if (Exp){
    stmt1
} else {
    stmt2
}
Unwatch()
```

(b)

Figure 6: Hardware needed for *HW-IF* (a) and code augmented by the compiler (b).

The compiler augments the code as in Figure 6(b). For each shared location accessed in the control expression, the compiler emits a *Watch* instruction. If the expression has a function call, the Watch instructions are for the addresses passed as arguments to the function. A Watch instruction for an address issues a bus access that allocates an entry in the AWT with the address. The bus access has the same effect as a bus write by the issuing processor in an invalidation-based protocol. Specifically, any cache (other than the issuing one) with a copy of the cache line has to write it back to memory (if its state was Dirty) and invalidate it (irrespective of the state it was in).

This operation does not add much execution overhead. The reason is that the control expressions in IF statements typically access few shared locations. For our applications, a control expression accesses on average 1.6 shared locations.

From then on, during the execution of the IF statement, when a processor other than the owner tries to reference a location being watched, it misses in its cache and issues a bus access. As shown in Figure 6(a), the AWT observes the access. If the requested address matches an address in one of AWT's entries and the requester is not the owner of the entry, the AWT returns a failed-access transaction (called negative-acknowledge or Nack). A Nack prompts the requesting processor's hardware to automatically retry the access, transparently to the software. This will continue until the owner processor removes the address from the AWT. The owner processor can always access the watched locations, either from its cache or from memory. Its bus access is not Nacked by the AWT.

At the end of the IF statement, the owner processor issues an *Unwatch* instruction, which clears all the AWT entries that it owns. A subsequent access from any processor to the previously-watched locations now succeeds.

Watch and Unwatch have acquire and release semantics, respectively. This means that no access past the Watch can proceed until Watch completes, and all accesses prior to the Unwatch have to be completed before Unwatch can proceed.

For simplicity, we flatten out nested IF statements, which means that the watched locations of all the nesting levels have equal status, and the first Unwatch clears all the entries of the owner processor. As a result, the remaining parts of the outer IF nests lose the ability to watch their addresses.

Overall, the proposed *HW-IF* design emphasizes hardware simplicity. We can improve its efficiency at the expense of more complicated hardware. For example, we can improve IF nesting support by including hardware counters in the AWT that are incremented and decremented at each monitored IF entry and exit. Similarly, we can eliminate the continuous Nack and retry with hardware support. However, our evaluation shows that this *HW-IF* design is already highly effective.

6.3. Discussion

HW-IF is attractive because it has an overhead low enough to be used on-the-fly, and can both detect and prevent ICRs. This means that *HW-IF* can prevent bugs from manifesting in a production environment. In addition, *HW-IF* suffers very few false negatives (i.e., failures to signal ICRs) and the false positives that it may have (i.e., incorrect signaling of ICRs) are usually harmless. The bottom row of Table 2 summarizes the sources of false negatives and false positives for *HW-IF*.

HW-IF has very few false negatives because it has none of those present in *SW-IF*: (i) since it constantly monitors all accesses to shared locations in *E*, it detects racy writes even if their combined effect does not change *E*'s logic value; (ii) it monitors IF statements from beginning to end, and so has no false negatives due to premature Confirmations; and (iii) it monitors IF statements even when *E* includes write operations and function calls, since *E* is not re-evaluated at any Confirmation point.

The only false negatives can occur due to the inability to watch the side effects of functions called in *E*, the simplified support for nested-IF monitoring, and certain special conditions related to deadlocks and thread preemption. We consider this third class in Section 6.4.

False positives can still happen, but they typically cause nothing but a slight delay in a cache line access. There are three sources of false positives. The first one results from the fact that the AWT deals with cache line addresses, and false sharing can cause it to signal a data race even when there is none. This issue can be avoided by designing an AWT that works at word or byte granularity. The second source of false positives is external reads to addresses accessed in *E*. These accesses do not cause ICRs, but induce Nacks from the AWT. This issue can be easily avoided by building the AWT to allow external read accesses. The third source of false positives is non-racy ICIVs. Overall, since false positives are inexpensive, we tolerate these cases.

Finally, note that *HW-IF*, as an extension of current watchpoints, provides a good hardware primitive to use in a broad range of debugging and security uses.

6.4. HW-IF Implementation Issues

6.4.1. Deadlock Effects. Whenever there is a mechanism for one processor to stall a second one, as in the case of *HW-IF*, one must watch for possible deadlocks. In *HW-IF*, a deadlock may occur in two cases, which may lead to false negatives and are easily handled. The first one appears when two processors are executing IF statements, both allocate AWT entries, and the timing is such that each ends up waiting for the other. Specifically, processor *P1* references a variable that is being watched by *P2* (it is in AWT's *P2* entry) and gets Nacked, while *P2* references a variable watched by *P1* and gets Nacked. This may occur, for example, when the control expressions of the two IF statements have common variables, or when the variables are different but share the same cache line (false sharing).

This case is solved by adding a Cycle Detection (CD) vector to the AWT controller. The CD vector has one entry per processor, and each entry contains the ID(s) of the processor(s) that are being Nacked by that particular processor, if there exists any. When a Nacking processor executes *Unwatch* and clears all its AWT entries, its CD entry is also cleared. On each Nack, the CD vector hardware attempts to detect a cycle by starting from the entry of the Nacking processor, reading the IDs in the entry, and using the IDs to read the corresponding entries in the CD. This process repeats until either there are no more Nacked processors or we end up in the entry of the original processor after two or more steps. In this case, a deadlock has been detected. If this happens, the AWT controller simply lets one of the bus accesses proceed instead of Nacking it. In the worst case, we are allowing a data race (which becomes a false negative); in the most likely case, the deadlock was due to false sharing.

The second case is when a thread *T1* executing an IF statement that is stalling a second one (*T2*) ends up spinning on a synchronization variable owned by *T2*. This case is solved by a small modification to the synchronization library. Specifically, after a processor has spun on a synchronization variable for a long time, the library simply clears the processor's AWT entries by executing *Unwatch*. If the spinning was due to the scenario described, the deadlock is broken by allowing a potential data race (which is a false negative). Custom user synchronizations can be instrumented in a similar fashion, after identifying them using methods proposed by Tian *et al.* [35]

6.4.2. Other Effects. There are other issues related to the practical implementation of the *HW-IF* hardware. The first one has to do with thread preemption and context switching. When a thread executing an IF statement with an entry in the AWT is preempted from its processor, the OS clears the processor's AWT entries using *Unwatch*. This avoids the possibility of long spins by other threads. While more advanced solutions are possible that rely on associating process IDs to AWT entries, they are unlikely to be cost-effective.

The second issue is support for multithreaded processors. Multithreaded processors have multiple hardware contexts and run multiple threads at a time. It is possible that different threads executing on different contexts of the same processor concurrently execute different IF statements. In this case, *HW-IF* can use an approach similar to the one described by Pacman [28]. The approach requires an extension where the messages sent by processors to the AWT include both the processor ID and the hardware context ID within the processor. Similarly, AWT entries would have both a PID and a context ID field. However, since the AWT is connected to the network, it can only observe data sharing across processors — not across contexts in a processor. Consequently, for *HW-IF* to work, a parallel program can only use one context per processor — although multiple programs can use multiple contexts of a processor. To allow a program to use multiple contexts from a processor, bigger changes would be needed.

A final issue has to do with the scalability of the AWT. We have assumed a centralized AWT, which is reasonable for a snoopy protocol. However, in a directory-based protocol, we need to distribute the AWT across different directory modules. Fortunately, like the directory, the AWT can be designed for a distributed environment, which is partitioned based on address ranges. Hence, each directory module has an associated AWT module, which is in charge of the range of physical addresses corresponding to that directory module.

7. Potential: Detect Existing ICR Bugs

We take the 38 ICR bugs from Table 1 and characterize whether they can potentially be detected with *SW-IF* and/or detected and prevented with *HW-IF*. Our approach is to manually inspect the source code of these bugs and, from it, decide whether they can be handled by our schemes. We estimate that *SW-IF* can detect 47% of these bugs. On the other hand, *HW-IF* can detect and prevent all of them. Table 3 shows the data broken down on a per-application basis.

Application	# IF-condition Data Races	# Detected by *SW-IF*	# Detected and Prevented by *HW-IF*
Apache	20	7	20
MySQL	8	6	8
Mozilla	8	5	8
Redhat (glibc)	0	0	0
Java SDK	1	0	1
Pbzip2	1	0	1
Total	38	18	38

Table 3: Bugs potentially handled on a per-application basis.

```
T1                                   T2

void ...print_thd (..) {             bool do_command(...) {
  if (thd−>proc_info) {
    putc(' ', f);
                                       thd−>proc_info = 0;

    fputs(thd−>proc_info, f);        }
  }
}
```

(a) MySQL #3596

```
T1                                        T2

if (apr_atomic_casptr(
    &(qi−>rp), new_recycle,
      new_recycle−>next)==                new_recycle−>next = qi−>rp
      new_recycle−>next){

        break;
}
```

(b) Apache #44402

Figure 7: Examples of IF-condition data races (ICRs).

To understand how we reached these conclusions, Figure 7 shows examples of two ICR bugs from Table 1. Figure 7(a) is a typical ICR that can be detected by *SW-IF*. In the bug, thread *T1* tests *thd→proc_info* and then uses it. Meanwhile, *T2* sets *thd→proc_info* to 0.

Figure 7(b) shows an ICR that cannot be detected by *SW-IF* but can be handled by *HW-IF*. There is an IF statement in *T1* and a write statement in *T2*. The control expression in the IF contains a call to *apr_atomic_casptr*, which is an atomic compare-and-swap. It swaps *&(qi→rp)* and *new_recyle* if *&(qi→rp)* is equal to *new_recycle→next*. In addition, *apr_atomic_casptr* returns the old value of *&(qi→rp)*. Between the return of *apr_atomic_casptr* and the comparison to *new_recycle→next*, *T2* intervenes and pollutes *new_recycle→next*, causing the bug. Detecting the bug in *SW-IF* is problematic because recomputing the control expression before leaving the THEN clause can cause side-effects due to the compare-and-swap. However, *HW-IF* can easily have the hardware watch all the shared locations accessed in the control expression and prevent the bug.

HW-IF also prevents a prevalent class of data race bugs called double-checked locking (DCL) [33]. In Section 4, we showed that 16% of all reported ICR bugs were DCL bugs. Figure 8 shows a simplified form of a bug from Apache (Bug #47158). In the example, a thread (*T1*) finds *x* to be null. Then, it updates *x.m* (as part of *Object()*) and *x*. The compiler or hardware can reorder the updates so that *x* gets updated before *x.m*. After the update to *x* but before the update to *x.m*, another thread (*T2*) executes the same code. *T2* finds that *x* is not null and goes on to use *x.m*, which is still uninitialized. If we use *HW-IF*, we place *Watch(x)* and *Unwatch()* around the innermost IF. This prevents *T2* from interleaving with *T1* during the initialization process and, therefore, avoids this bug. Recall that Unwatch has release semantics (Section 6.2) and, therefore, it only executes after all prior accesses have completed, including the updates to *x* and *x.m*. *SW-IF* is unable to detect this bug.

```
if (x == NULL){ // start of DCL
  synchronized (this) {
    if (x == NULL){
      x = new Object();  // initializes x.l, x.m, etc
    }
  }
} // end of DCL
y = x.m;
```

Figure 8: Example of a double-checked lock (DCL) bug.

8. Evaluation

In this section, we evaluate *SW-IF* and *HW-IF*. We start by describing our experimental setup (Section 8.1). Then, we characterize the IF statements in the applications (Section 8.2), and evaluate the execution overhead of our algorithms (Section 8.3), their effectiveness at detecting new data race bugs (Section 8.4), and the sensitivity of *SW-IF* to the delay inserted at Confirmation points (Section 8.5).

8.1. Experimental Setup

We use the Cetus source-to-source compiler [13] to analyze and transform applications for *SW-IF* and *HW-IF*. Cetus uses its intermediate representation tree and call graph to find the IF statements that access shared locations. It then instruments them with either Confirmation points or Watch/Unwatch instructions.

For *SW-IF*, we run the applications with 8 threads on a desktop with 4 2-context Intel Xeon cores running at 2.33 GHz. For *HW-IF*, since there is no hardware that implements the AWT watchpoint table described in Section 6.2, we instrument the application code with PIN and call an execution-driven, cycle-level architectural simulator. The simulator models a chip multiprocessor (CMP) with 4 or 8 processors and a memory subsystem. The simulator intercepts the Watch and Unwatch instructions and emulates the AWT.

Each AWT entry is 62-bit long, and contains the line address, processor-ID, and Valid bit. Although a watched IF statement only accesses 1.6 locations on average, we conservatively have 100 entries in the AWT. In our experiments, the AWT never gets full. If it did, we would simply discard entries. When we add delay at Confirmation points, we add 15μs. The default parameters are shown in Table 4.

Architecture	CMP with 4 or 8 processors
Coherence protocol	Snoopy-based MESI on a 64byte bus
Processor type	2-issue, in-order, 1GHz
Private L1 cache	32Kbytes, 4-way assoc., 64byte lines
Private L2 cache	512Kbytes, 8-way assoc., 64byte lines
L1 and L2 hit latency	Min. 2 and 8 cycles round trip, respect.
L2 miss latency	Min. 30 cycles round trip to other L2s and 250 cycles round trip to main memory
Watch/Unwatch instr.	Min. 500 cycles (includes main mem. access)
AWT size	100 entries; each entry is 62 bits
Delay added (optional)	15 μs

Table 4: Default architectural parameters.

For the evaluation, we use the following applications. To characterize the IF statements in programs and to quantify the performance overhead of our algorithms, we use the SPLASH-2 applications. The reason is that these codes have a well-defined standard input set, which is useful for measuring performance. As a reference, we also measure the performance overhead of running the popular Valgrind-3.6.1 [22] debugging tool. For Valgrid, we only turn-on its race detection part (Helgrind).

SW-IF and *HW-IF* do not find any ICRs in the SPLASH-2 codes. Therefore, we also run our algorithms on Cherokee-0.9.2 [1] and Pbzip2-0.9.4 [3]. Cherokee is a web server written in C, and Pbzip is a parallel data compression application that can be compiled by Cetus after disabling a few macros. Since Cetus can only analyze C programs, we cannot run any of the other applications from Table 1 that are written in C++ or Java. The glibc library is written in C. However, testing glibc in a stand-alone manner is not representative. For this reason, we do not do it. We also use Cherokee and Pbzip to explore the sensitivity of the race detection capabilities of *SW-IF* to the length of delay at Confirmation points.

We ran Cherokee and Pbzip using *SW-IF* and *HW-IF* 6 times each. As we will see, they found 8 ICRs. In addition, these are the only ICRs found by Helgrind. This indicates that our techiques are robust.

8.2. IF Statement Characterization

We first try to understand the structure of the IF statements. For this experiment, we use *SW-IF* with delay insertion. Table 5 lists the total number of IF statements in the codes and the number of checked (i.e., monitored with Confirmation points) IF statements — both the static number seen in the source code and the dynamic number observed at runtime. We see that about a third of all the dynamic IF statements are checked on average.

Apps	# IF Statements		# Checked IF Statements	
	Static	Dynamic	Static	Dynamic
Radiosity	357	12246807	216	3225037
Water_nsquared	63	13210272	39	13134311
Water_spatial	111	8995819	48	4955930
Ocean_con	518	141680	313	16794
Ocean_non	302	141676	235	8170
Cholesky	283	834469	200	715804
FFT	60	804	55	35
LU_cont	92	13620	79	12387
LU_non	64	449	58	285
Radix	51	121	31	45
Barnes	90	1686857	62	777343
FMM	308	33421676	238	1432271
Raytrace	354	8530683	159	3128360
Average	204	6094225	133	2108213

Table 5: Static and dynamic number of IF statements.

Figure 9 characterizes the dynamic IF statements further. It breaks the number of dynamic IF statements into *Checked*, *SharedNoCheck*, and *PrivateOnly*. *Checked* are those that are instrumented with Confirmation points; *SharedNoCheck* are those that have a shared location access in the condition expression but are not instrumented due to compiler limitations; *PrivateOnly* are those that only have accesses to private locations in the condition expression and thus do not need checks. The figure shows that *SW-IF* checks almost all of the IF statements that have shared accesses. It misses only 3% of the cases.

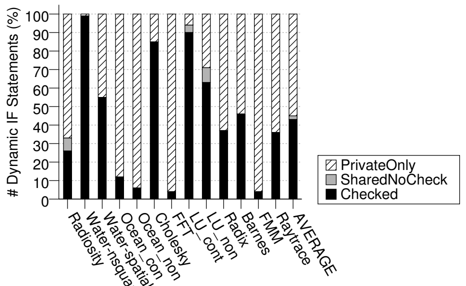


Figure 9: Fraction of dynamic IF statements that are checked.

Figure 10 shows the fraction of checked dynamic IF statements that have delays inserted in them. The figure breaks the number of checked dynamic IF statements into those that receive delays (*Delayed*) and those that do not. The *Loop*, *Recursive*, and *Lock* categories show the cases where the IF statement did not receive a delay because it was (i) in a loop, (ii) in a recursive call but not in a loop, and (iii) in a critical section but not in any of the prior categories. The figure shows that *SW-IF* inserts delays in about 20% of the dynamic checks on average. The main reason why this number is not higher is due to IF statements inside loops.

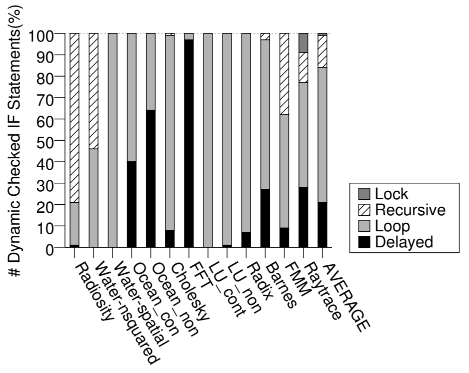


Figure 10: Checked dynamic IF statements that receive delays.

8.3. Performance Overhead

We now evaluate the performance overhead of *SW-IF* and *HW-IF*. To evaluate *SW-IF*, we run the instrumented applications natively using 8 threads. We consider two scenarios: binary instrumented with Confirmation points without delays (*SW-IF*) and with delays for the appropriate Confirmation points (*SW-IFdelay*). We also run the original uninstrumented binary (*Original*). As a reference, we also show the execution time of the applications running on Helgrind, which is the race detection part of Valgrind. While Helgrind is a much more general data race detector than *SW-IF*, it gives a reference data point. The results are shown in Figure 11 where, for each application, the bars are normalized to *Original*.

From the figure, we see that the average performance overhead of *SW-IF* is 2%. This is a very low overhead, which shows that *SW-IF* could even be used in a production environment. *SW-IFdelay* has an average performance overhead of 6%, which is still small enough for a debugging and testing environment. We also see that Helgrind has a much higher performance overhead. This is because it runs sequentially and uses a general algorithm

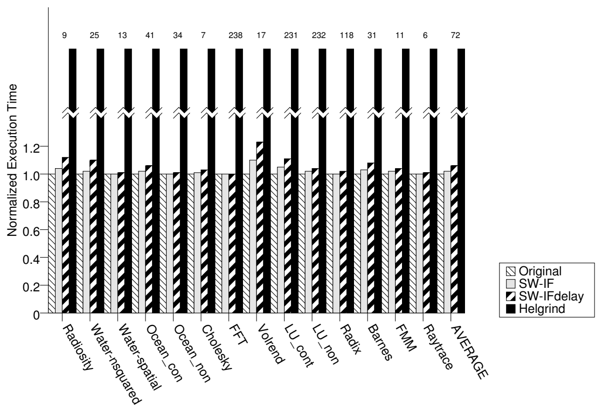


Figure 11: Execution time of the applications under *SW-IF*.

that can find many classes of races.

To evaluate *HW-IF*, we run 4- and 8-threaded applications on the architecture simulator described in Section 8.1. The results are shown in Figure 12. The bars show the execution time overhead of running the application on a machine with *HW-IF* hardware over one without *HW-IF* hardware. We see that, on average, the execution overhead is less than 1% for both 4 and 8 threads. This makes *HW-IF* perfectly suitable for on-the-fly production environments.

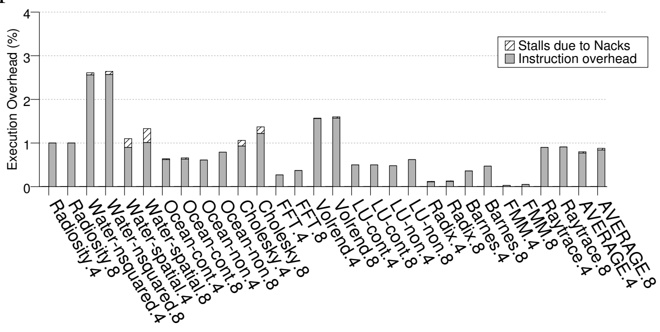


Figure 12: Execution time overhead of the programs under *HW-IF*. The figure shows data for 4- and 8-threaded runs.

The execution time overhead shown in Figure 12 is broken down into two components: *Stalls due to Nacks* and *Instruction overhead*. The *Stall* overhead is due to delays incurred by processors when they are issuing bus requests that are Nacked by the AWT. The *instruction overhead* refers to slowdowns incurred by processors when they are executing Watch and Unwatch, or when they suffer additional cache misses due to them. We can see from the results that the stalls due to Nacks cause negligible overhead; the overhead primarily comes from the Watch and Unwatch execution, and misses.

There is no standard input set for Cherokee and Pbzip2. However, if we use the same input set as Yu *et al.* [41], the measured execution time overhead for Cherokee and Pbzip2 is 3.2% and 1.4%, respectively, for *SW-IFdelay*; 1.1% and 0.6%, respectively, for *SW-IF*; and 0.8% and 0.4%, respectively, for *HW-IF*. These are small overheads.

8.4. Detecting New IF-condition Data Race Bugs

SW-IF without delays and *HW-IF* do not find any ICRs in the SPLASH-2 codes. However, they find several in Cherokee and Pbzip, and some of them are *new, unreported bugs*. Table 6 shows the ICRs that these algorithms find. The table assigns an ID to each bug, shows whether it is a new bug, lists the source code locations of the ICR, and shows whether *SW-IF* or *HW-IF* can find it.

Bug ID	New Bug?	Locations of the ICR	Found by	
			SW-IF	*HW-IF*
Cherokee-1	Yes	thread.c:327 – server.c:1676	No	Yes
Cherokee-2	Yes	thread.c:57 – bogotime.c:114	Yes	Yes
Cherokee-3	Yes	thread.c:1890 – server.c:1132	No	Yes
Cherokee-4	Yes	thread.c:1945 – server.c:275	Yes	Yes
Cherokee-5	No	bogotime.c:114 – bogotime.c:109	No	Yes
Cherokee-6	No	buffer.c:92 – buffer.c:187	No	Yes
Pbzip2-1	Yes	pbzip2.cpp:704 – pbzip2.cpp:966	Yes	Yes
Pbzip2-2	No	pbzip2.cpp:1044 – pbzip2.cpp:889	No	Yes

Table 6: ICRs found.

As shown in the table, our algorithms find 6 ICRs in Cherokee and 2 in Pbzip2, of which 5 are new, unreported bugs. We have reported these bugs to the software developers. *SW-IF* without delays detects 3 of them while *HW-IF* detects and prevents all of them.

To understand the new bugs in detail, Figure 13 displays the source code and the buggy interleaving for each new bug.

BugID	Source Code T1	Source Code T2

Cherokee−1	if (thread−>conns_num >0)
	thread−>conns_num−−;
		conns_num += THREAD(thread)
		−>conns_num;

Cherokee−2	if(thd−>bogo_now
	== cherokee_bogonow_now)
	return;
		cherokee_bogonow_now = newtime;

Cherokee−3	if(unlikely(srv−>wanna_exit))
	thd−>exit = true;
		srv−>wanna_exit= true;

Cherokee−4	if ((thd−>exit==false)&& ...)){
	step_MULTI_THREAD_block
	(thd, ...);
	}
		THREAD(i)−>exit = true;

Pbzip2−1	if(OutputBuffer[currBlock]
	.bufsize<1 ||...){
	usleep(50000);
	}
		OutputBuffer[blockNum]
		.bufSize=outSize;

Figure 13: Description of the new ICRs found by *SW-IF* and *HW-IF*.

In Cherokee-1, *T2* updates shared variable *conns_num*, which could be aliased to the variable that *T1* reads and writes. *HW-IF* can detect and protect against this bug. However, *SW-IF* cannot because *T2* makes *conns_num* bigger and, therefore, if we test the *thread→conns_num>0* condition later, it will still be true.

In Cherokee-2, *T2* may change *cherokee_bogonow_now* before *T1* executes the return in the THEN clause of the IF statement. If so, the rest of the function that contains this IF statement may be incorrectly skipped. This bug can be detected by both *SW-IF* and *HW-IF*.

In Cherokee-3, *T2* can change *srv→wanna_exit* after *T1* has used it in an IF control expression. Since the latter contains the function call *unlikely* with potential side-effects, *SW-IF* cannot be used. Hence, only the *HW-IF* scheme can detect this bug.

Cherokee-4 and Pbzip2-1 are similar to Cherokee-2 in that *T2* can change the value of the IF control expression while *T1* is executing the THEN clause. For Cherokee-4, the result is an unnecessary block; for Pbzip2-1, the result is an unnecessary sleep. Both bugs can be detected with *SW-IF* and *HW-IF*.

It is important to note that while *SW-IF* missed some ICRs (i.e., it had false negatives) due to the limitations described in Table 2, it did not suffer any false positives: all ICRs reported by *SW-IF* were actual races. On the other hand, *HW-IF* detected and prevented all the ICRs that were reported by Helgrind (i.e., it had no false negatives). While *HW-IF* did suffer from the occasional false positive due to false sharing and due to external read conflicts in the AWT (as described in Table 2), false positives only result in Nacks. These Nacks had negligible impact on performance as we saw in Section 8.3.

Helgrind finds the data races in Table 6. However, when we ran Helgrind, we obtained messages for hundreds or thousands of data races — many of which have low importance. It took us several days to analyze the log.

8.5. Sensitivity of SW-IF to Delays

Previous work [10] showed that adding delays at strategic points is beneficial when trying to expose data races. Hence, we perform the same experiments as in the previous section, but used *SW-IF* with delays. Our goal is to extend the time that *SW-IF* is able to detect the races.

We find that *SW-IFdelay* finds exactly the same number of data races as *SW-IF*, namely those of Table 6. It seems, therefore, that delays are not important for exposing ICRs for the particular applications we use.

9. Related Work

9.1. Specialized Data Race Detectors

In the area of data race debugging, we see race specialization and low-overhead techniques as promising. One approach to reduce overhead is to use sampling, as in LiteRace [16] and DataCollider [10]. LiteRace looks for data races in infrequently-exercised code regions. DataCollider looks for data races in the kernel. It places hardware watchpoints on some variables (or repeatedly reads them) to find if a remote thread modified their values in between. When DataCollider places a watchpoint, a processor interrupts all other processors and atomically updates their watchpoint registers. Theoretically, putting all the addresses from the IF control expression in watchpoints could deliver a functionality like *HW-IF*. However, it would have a very large overhead, and be incompatible with production runs.

Some techniques try to avoid data races at runtime (e.g., [6, 28, 29, 30, 42]). In particular, ToleRace [30] and ISOLATOR [29] focus on asymmetric races, where a well-synchronized thread inside a critical section has a race with a second thread lacking proper synchronization. They avoid this by isolating the well-synchronized thread. Pacman [28] supports fine-grain isolation in hardware. It avoids asymmetric races by using hardware cache coherence mechanisms to prevent other processors from accessing variables that the well-synchronized thread is accessing inside the critical section. Pacman proposes a hardware primitive similar to our Watch in *HW-IF*. However, it differs from *HW-IF* in several ways: (i) it has a different goal, (ii) it is more costly, since it needs to protect all the addresses accessed in a critical section, (iii) it does not use any compiler pass, and (iv) it is implemented with more elaborate signature-based hardware.

9.2. Hardware Transactional Memory

One could think of using Hardware Transactional Memory (HTM) in place of the AWT for *HW-IF* to guarantee atomicity. However, in its most popular form, HTM is not the best primitive for ICRs. Using a transaction prevents accesses from other threads to any variable inside the IF statement. This is not what we intend. We want to prevent only accesses to the variables that are inside the IF control expression. Table 7 shows an example where HTM does not work as we want: there is no ICR and the transaction fails.

T1	T2
`begin_transaction;` `if (x) {` `lock L;` `y` `unlock L;` `}` `end_transaction;`	`lock L;` `...`

Table 7: HTM does not work as we want for ICRs.

However, if HTM gives the program the ability to selectively mark which accesses set the speculative cache bits, then HTM could be used to protect only the variables in the IF-condition. Still, we would have to overcome two HTM limitations. First, false sharing would result in squashes with HTM, rather than stalls as with the AWT. Second, in large IF statements, HTM would be subject to squashes due to cache overflow.

9.3. Atomic Region Detection and Violation

A related concurrency bug is atomicity violations. Many works have focused on these bugs, such as SVD [39], AVIO [14], AtomAid [15], AtomTracker [18], LifeTx [42], and AFix [12]. SVD [39] identifies atomic regions called *CU*s on-the-fly by looking at the pattern of reads/writes. It uses heuristics to build CUs: (i) a CU needs to start with a read, and then a write, and (ii) a CU should not contain two independent computations. SVD works by instrumenting an executable. SVD can be slow (up to 65x slowdown) and require a post-pass analysis to fix the CUs.

AtomAid [15] and LifeTx [42] prevent illegal interleavings from manifesting by using transactional memory to protect code regions. AVIO [14], AtomTracker [18], LifeTx [42] and AFix [12] employ some form of training mechanism. They try to learn some semantic invariants from correct training runs, and use this information to detect/fix bugs that violate such invariants. One drawback of the training approach is that, unless the training is done with all kinds of interleavings and inputs, the scheme may not be effective at detecting violations.

10. Conclusions

This paper introduced the *IF-Condition Invariance Violation (ICIV)* as a code pattern that is likely to be a sign of a concurrency bug, and the *IF-Condition Data Race (ICR)* as an ICIV caused by a data race. An analysis of the data races reported in bug databases showed that ICRs occur relatively often. ICRs are typically bugs because of the implicit invariance of the IF condition implied by IF statements. Moreover, their obvious structure allows the implementation of a very efficient data race detector. This paper introduced how such a detector can be built purely in software (*SW-IF*) or with the help of some hardware (*HW-IF*). *HW-IF* can be used to both detect and prevent ICRs.

We evaluated *SW-IF* and *HW-IF* using a variety of applications. We showed that these new techniques are effective at finding new data race bugs and run with low overhead. Specifically, *HW-IF* found 5 new (unreported) ICR bugs in the Cherokee web server and the Pbzip2 application; *SW-IF* found 3 of them. In addition, 8-threaded executions of the SPLASH-2 applications showed that, on average, *SW-IF* added 2% execution overhead, while *HW-IF* added less than 1%. These minuscule overheads point to the use of both *SW-IF* and *HW-IF* as lightweight race detectors. *SW-IF* can speed-up the process of code development and testing, while *HW-IF* can be used to avoid races in production runs.

Acknowledgments

We thank the anonymous referees, who helped improve this work substantially. This work is supported in part by NSF under grants CCF-1012759 and CNS-1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).

References

[1] "Cherokee Web Server," http://www.cherokee-project.com/.

[2] "Intel Corporation. Intel Thread Checker," http://www.intel.com/support/performancetools/threadchecker.

[3] "Parallel BZIP2," http://compression.ca/pbzip2/.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional Detection of Data Races," in *PLDI*, 2010.

[5] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," in *OOPSLA*, November 2002.

[6] L. Chew and D. Lie, "Kivati: Fast Detection and Prevention of Atomicity Violations," in *EuroSys*, 2010.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," in *PLDI*, June 2002.

[8] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "RADISH: Always-on Sound and Complete Race Detection in Software and Hardware," in *ISCA*, 2012.

[9] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *SOSP*, October 2003.

[10] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-Race Detection for the Kernel," in *OSDI*, 2010.

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part II*, Intel Corporation.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated Atomicity-Violation Fixing," in *PLDI*, 2011.

[13] S. Lee, T. A. Johnson, and R. Eigenmann, "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation," in *LCPC*, 2003.

[14] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," in *ASPLOS*, 2006.

[15] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations," in *ISCA*, June 2008.

[16] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-Race Detection," in *PLDI*, 2009.

[17] S. L. Min and J.-D. Choi, "An Efficient Cache-Based Access Anomaly Detection Scheme," in *ASPLOS*, April 1991.

[18] A. Muzahid, N. Otsuki, and J. Torrellas, "AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection," in *MICRO*, 2010.

[19] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas, "SigRace: Signature-Based Data Race Detection," in *ISCA*, June 2009.

[20] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *PLDI*, 2006.

[21] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," in *PLDI*, June 2007.

[22] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI*, 2007.

[23] R. Netzer and B. Miller, "Detecting Data Races in Parallel Program Executions," in *LCPC*, 1990.

[24] C.-S. Park and K. Sen, "Randomized Active Atomicity Violation Detection in Concurrent Programs," in *FSE*, November 2008.

[25] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Context-Sensitive Correlation Analysis for Race Detection," in *PLDI*, June 2006.

[26] M. Prvulovic, "CORD: Cost-Effective (and Nearly Overhead-Free) Order-Recording and Data Race Detection," in *HPCA*, June 2006.

[27] M. Prvulovic and J. Torrellas, "ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes," in *ISCA*, June 2003.

[28] S. Qi, N. Otsuki, A. Muzahid, and J. Torrellas, "Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware," in *HPCA*, 2012.

[29] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs," in *ASPLOS*, March 2009.

[30] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman, "Detecting and Tolerating Asymmetric Races," in *PPOPP*, February 2009.

[31] M. Ronsse and K. D. Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," *ACM Trans. Comp. Syst.*, May 1999.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Comput. Syst.*, November 1997.

[33] D. C. Schmidt and T. Harrison, "Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-Safe Objects," in *PLOP*, 1996.

[34] "Sun Microsystems. Sun Studio Thread Analyzer," http://docs.sun.com/app/docs/doc/820-0619, Sun Microsystems.

[35] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, "Dynamic Recognition of Synchronization Operations for Improved Data Race Detection," in *ISSTA*, 2008.

[36] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably Solving File TOCTTOU Races with Hardness Amplification," in *FAST*, 2008.

[37] C. von Praun and T. R. Gross, "Object Race Detection," in *SIGPLAN Not.*, November 2001.

[38] J. Wei and C. Pu, "TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study," in *FAST*, 2005.

[39] M. Xu, R. Bodik, and M. D. Hill, "A Serializability Violation Detector for Shared-Memory Server Programs," in *PLDI*, June 2005.

[40] J. Yu, "Collection of Concurrency Bugs," 2009, http://web.eecs.umich.edu/∼jieyu/bugs.html.

[41] J. Yu and S. Narayanasamy, "A Case for an Interleaving Constrained Shared-Memory Multi-Processor," in *ISCA*, 2009.

[42] ——, "Tolerating Concurrency Bugs Using Transactions as Lifeguards," in *MICRO*, 2010.

[43] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," in *SOSP*, October 2005.

[44] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: Hardware-Assisted Lockset-Based Race Detection," in *HPCA*, February 2007.