Final Report

Project Title: A program analysis framework to increase the performance and effectiveness of asymmetric race detection and toleration

By Paruj Ratanaworabhan

February, 2015

Contract No. ……

Final Report

A program analysis framework to increase the performance and effectiveness of asymmetric race detection and toleration

Researcher	Institute
Paruj Ratanaworabhan	Kasetsart University

This project is supported by the Thailand Research Fund

Abstract

Project Code:

Project Title: A program analysis framework to increase the performance and effectiveness of asymmetric race detection and toleration

Investigator: Paruj Ratanaworabhan

E-mail Address: paruj.r@ku.ac.th

Project Period: 12 months

Abstract: As multicore processors become ubiquitous, parallel programs that exploit those extra cores are expected to be prevalent. However, parallel programming is not an easy undertaking. At present, programmers find that it is already hard enough to correctly program in conventional sequential mode. Parallel programming worsens the status quo as it introduces additional errors that are not found in sequential programming. These are, for example, deadlock, atomicity violation, and data races. This project will focus on data races, specifically asymmetric data races. In general, a race is defined as a condition where multiple threads access a shared memory location without synchronization and there is at least one write among the accesses. Asymmetric races occur when one thread correctly protects a shared variable using a lock while another thread accesses the same variable improperly due to a synchronization error (e.g., not taking a lock, taking the wrong lock, taking a lock late, etc.).

Asymmetric races are common and developers in software houses like Microsoft constantly have problems with them. There are two reasons for this. First, usually a programmer's local reasoning about concurrency, e.g., taking proper locks to protect shared variables, is correct. Errors due to taking wrong locks or no locks lie outside of the programmer's code, for example, in third party

libraries. Given that lock-based programs rely on convention, this phenomenon is understandable. The second reason has to do with legacy code. As software evolves, assumptions about a piece of code may be invalidated. For instance, a library may have been written assuming a single-threaded environment, but later the requirements change and multiple threads use it. An expedient response to this change is to demand that all clients wrap their calls to the library, acquiring locks before entry and releasing them on exit. Because this solution requires that all clients be changed, races can be introduced when clients fail to follow the proper locking discipline.

This project tackles asymmetric data races in locked-based parallel programs, specifically those written in unsafe languages such as C or C++ that use add-on libraries for threading and synchronization. At present, a large installed code base of such programs exists and programmers continue to write parallel code in this paradigm. The project aims to increase the efficiency and effectiveness of an asymmetric race detector and tolerator.

Keywords: Asymmetric race toleration and detection, static and dynamic program analysis, dynamic instrumentation

Executive Summary

Asymmetric races are data races caused when one thread accesses a shared variable guarded by a lock in a critical section and another thread accesses the same shared variable without holding the same lock. Asymmetric races are common and usually harmful. They often arise when well-tested code interacts with buggy legacy code or third-party libraries. Existing solutions for tolerating asymmetric races, whether based on software or hardware, have some limitations: they require either complier support, or application changes, or new hardware to be added to commercial hardware platforms.

This project proposes a consistent execution model for critical sections in lock-based multi-threaded programs. During the consistent execution of a critical section, two conditions are satisfied: (1) shared variables read in the critical section are not written outside and (2) shared variables written in the critical section are not read and written outside. As a result, asymmetric races can never occur. Based on this consistent execution model, we present a new software-based scheme, called ARace, to dynamically ensure that all critical sections are consistently executed by exploiting write buffering and shared variable protection. ARace can be directly applied to binary code and requires no additional compiler support or application changes. We have implemented ARace based on dynamic binary instrumentation and evaluated it with the applications from SPLASH-2 and Phoenix. Our results show that ARace guarantees the absence of asymmetric races while incurring only about 1x overhead on average.

Objectives

The main objective of this project is to build a software tool that is able to better detect and tolerate asymmetric data races. We achieve our main objective through three steps:

1. Develop a theory for and design a program dynamic analyzer, ARace, to increase the performance and effectiveness of asymmetric race detection and toleration.
2. Implement ARace based on Pin, a dynamic binary instrumentation framework from Intel.
3. Evaluate ARace against two widely adopted benchmarking suites, SPLASH-2 and Phoenix, from Stanford University

Research Methodology

Our research methodology follows the outline below:

- Surveying related work
- Developing the underlying theory for and designing the software tool that will satisfy our objectives
- Implementing the software tool based on the theory and design developed in the previous step
- Evaluating the effectiveness of our software tool using comprehensive benchmark programs

Related Work

The existence of data races makes multi-threaded programs error-prone. When two threads access a shared variable without any synchronization, where one of the accesses is a write, a data race happens. Data races may cause multi-threaded programs to exhibit undesired behaviors. Some data races escaping from in-house testing may be catastrophic in the real world, as is the case for the Northeastern U.S. electricity blackout.

There has been a plenty of research on dealing with data races. These research efforts fall into two categories: prior detection and post tolerance. The former detects and removes data races as aggressively as possible during in-house testing, while the latter tolerates data races in production runs. Despite extensive in-house testing, some data races still lurk around in released products. Thus, the latter techniques are invaluable in practice.

There is one class of data races, called asymmetric races, which occur at the time when one thread accesses a shared variable inside a critical section protected by a lock and another thread also accesses the same shared variable due to a synchronization error (e.g., outside any critical section or inside a critical section but not

protected by the same lock). The figure below illustrates an asymmetric race.

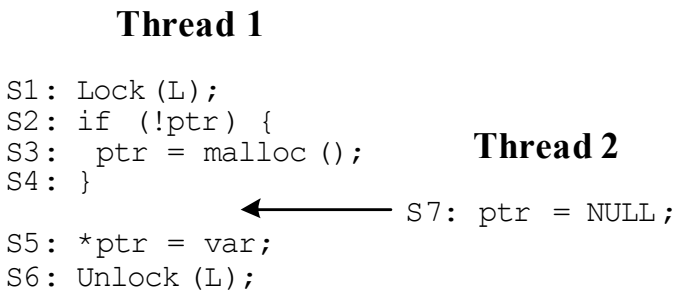


Figure 1. An asymmetric race

In this example, ptr is a shared variable. The two reads to ptr in thread 1 are inside a critical section protected by a lock L but the write to ptr in thread 2 is not inside any critical section. This race may lead to inconsistent results when reading ptr in thread 1 during different program executions. This happens when the write to ptr in thread 2 takes place between the first read to ptr at S2 and the second read to ptr at S5 in thread 1.

Asymmetric races are common in real applications and usually harmful. Among the harmful data races found, about 20% are estimated to be asymmetric races. The developer usually expects the accesses to shared variables to be made inside critical sections guarded by appropriate locks. Unfortunately, asymmetric races often arise when the developer's code interacts eventually with the other code from third-party libraries or legacy binaries. The latter code may be originally written only for single-threaded applications in mind. Thus, the presence of asymmetric races is often beyond the developer's control.

Although prior detection is useful for detecting and removing some asymmetric races, post tolerance can be more attractive. Many asymmetric races happen only when well-tested code interacts with legacy binaries or third-party libraries, whose source may be unavailable. In addition, due to their asymmetric nature, asymmetric races, which cannot be found during prior detection,

can be better prevented with post tolerance. A simple way to tolerate asymmetric races is to prevent another thread from accessing a shared variable if some thread is accessing it in a critical section, avoiding corrupting the shared variable.

We will now look at prior work related to the area of asymmetric races.

Asymmetric Races

ToleRace [1-4] is the first proposed software scheme for detecting and tolerating asymmetric races. ToleRace copies two shadows, v′ and v″, for each shared variable v accessed in a critical section when a thread T1 executes the critical section. Then T1 accesses v′ in the critical section. At the same time, another thread T2 can accesses v outside the critical section. After T1 has reached the end of the critical section, ToleRace compares the values of v and v″. Then ToleRace decides which value of v and v′ should be reserved as the new value of v: (1) if T1 can be serialized before T2, the value of v is reserved; (2) if T2 can be serialized before T1, the value of v′ is reserved; (3) if T1 and T2 cannot be serialized, ToleRace has to interrupt the execution of the program. ToleRace can tolerate asymmetric races in the former two cases but is inadequate in the last case ([5] illustrates one such example).

ISOLATOR [5] is another software scheme. At the beginning of a critical section, any page p that will be accessed in the critical section is copied to a shadow page p′. Then ISOLATOR protects p by making it inaccessible. The accesses to p in the critical section are redirected to p′. The accesses to p not in the critical section will cause page fault exceptions. At the end of the critical section, ISOLATOR copies the content from p′ to p, and unprotects p to be accessible. ISOLATOR needs compiler support or even application changes so that pages can be shadowed appropriately. Besides, for every shadow page, ISOLATOR uses a temporary page to copy it back. However, if there are multiple shadow pages, the atomicity of copying them back is not guaranteed in ISOLATOR.

Pacman [6] also aims to asymmetric races. The main difference between Pacman and above two schemes is that Pacman is based on hardware. Pacman exploits cache coherence hardware to protect cache lines that contain variables accessed in a critical section. If instructions not in the critical section try to access these cache lines, they will fail and have to wait. Pacman needs additional hardware support to exploit cache coherence. Besides, Pacman has no knowledge about critical sections. That is because critical sections have no difference with normal code from a hardware perspective. Compared with software-based schemes, Pacman is unintrusive and has negligible execution overhead. Nevertheless, it is not yet supported by current computer platforms.

Transactional Memory

Transactional Memory (TM) is another way to provide atomicity for lock-free data structures. In TM, an atomic region is considered as a transaction and the transaction is executed speculatively. At the end of the transaction, TM checks whether there is conflict. If yes, TM aborts the transaction and rolls back to re-execute the transaction. Otherwise, the transaction is committed. TM needs to handle side effect operations effectively during rollback, which is still an open problem. TM can be implemented based on hardware [7], software [8], or hybrid [9].

Data Race Detection

There is a large body of research focusing on data race detection, both static and dynamic. Static detections use program analysis techniques, like type-based checking [10], static flow analysis [11], or lockset analysis [12]. One inherent drawback of static detections is that a lot of false positives are reported. Dynamic detections are mainly based on the lock-set algorithm [13], happens-before analysis [14] or hybrid of the two [15]. Although dynamic detections have fewer false positives than static detections, they have the challenge of coverage.

If we focus on the three systems most closely related to ours, ToleRace, ISOLATOR, and Pacman, we see the following shortcomings:

- ToleRace cannot tolerate a case of asymmetric race where the two executing threads T1 and T2 cannot be serialized, and, hence, needs to interrupt the execution of the program.

- ISOLATOR requires compiler support or even application changes so that pages can be shadowed appropriately. This scheme is ineffective unless the source of a program is available. Unfortunately, asymmetric races are often triggered when well-tested code interacts with legacy binaries or third-party libraries.

- Pacman is a hardware-based scheme that is uninstrusive and induces negligible slowdown. However, it is not yet supported by current computer platforms.

To overcome the imitations inherent in the three aforementioned schemes, this project proposes a consistent execution model for critical sections in lock-based multi-threaded programs. During the consistent execution of a critical section, two conditions are satisfied: (1) shared variables read in the critical section are not written outside and (2) shared variables written in the critical section are not read and written outside. As a result, asymmetric races can never occur. Based on this consistent execution model, we present a new software-based scheme, called ARace, to dynamically ensure that all critical sections are consistently executed by exploiting write buffering and shared variable protection. ARace works on-the-fly, requires no additional compiler support or application changes, and can be deployed even when the source code of a program is not available. We have implemented ARace based on dynamic binary instrumentation. Our results show that ARace guarantees the absence of asymmetric races with acceptable performance overhead.

There are fundamental differences between ARace and Transactional Memory (TM). Even though both use write buffers, ARace does not need to detect versions and conflicts during the execution of a critical section, because it protects the shared variables read in the critical section. When there are conflicts, TM must abort a transaction and rollback. During rollback, TM needs to handle side effect operations effectively, which is still an open problem. In contrast, there is no notion of abort-and-rollback in ARace, because its program executions are not speculative.

1. P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and Tolerating Asymmetric Races. In PPoPP, 2009.
2. P. Ratanaworabhan, D. Kirovski, and R. Nagpal. Efficient Runtime Detection and Toleration of Asymmetric Races. In IEEE Trans. on Comput., Vol. 61, No. 4, 2012.
3. P. Ratanaworabhan, M. Burtscher, D. Kirovshi, and B. Zorn. Hardware Supprot for Enforcing Isolation in Lock-Based Parallel Programs. In ICS, 2012.
4. D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. An Oracle for Tolerating and Detecting Asymmetric Races. Microsoft Research Technical Report MSR-TR-2007-122, 2007.
5. S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dyamically Ensuring Isolation in Concurrent Programs. In ASPLOS, 2009.
6. S. Qi, N. Otsuki, L. O. Nogueira, A. Muzahid, and J. Torrellas. Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware. In HPCA, 2012.
7. L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In ISCA, 2008.
8. V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: A Software Transactional Memory for Many-Cores. In EuroSys, 2012.
9. C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In ISCA, 2007.

10.C. Boyapati and M. C. Rinarad. A Parameterized Type System for Race-Free Java Programs. In OOPSLA, 2001.
11.D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In SOSP, 2003.
12.P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In PLDI, 2006.
13.S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Andersom. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In ACM Trans. Comput. Syst., 1997.
14.E. Schonberg. On-the-fly Detection of Access Anomalies. In PLDI, 1989.
15.A. Muzahid, D. S. Gracia, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In ISCA, 2009.

Theory and Design

Overview

ARace exploits two techniques to ensure that the execution Θ of a critical section Ξ is consistent, where $\Xi = <\Lambda, \Phi, \Gamma>$. The first is **Write Buffer**. The writes to any ϕ in Φ during Θ are redirected to the write buffer. The write buffer is written back to original shared variables when the last instruction in Θ is executed. By this way, the intermediate statuses of any ϕ in Φ generated by instructions in Θ are hidden, and instructions not in Θ can only see the final result of ϕ after Θ is finished.

Another technique utilized by ARace is **Shared Variable Protection**. Any ϕ in Φ read by instructions in Θ is protected to be read-only. When Θ is executed, if an instruction not in Θ tries to modify ϕ after instructions in Θ have read ϕ, it will fail. Then it has to wait for the finish of Θ. Any protected ϕ is unprotected to be writeable when the last instruction in Θ is executed.

To prohibit inconsistent statuses of shared variables, ARace forbids two critical sections that access same shared variables from being executed concurrently. For two critical sections $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$ and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, if $\Phi_1 \cap \Phi_2 \neq \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 are not allowed to be executed concurrently. Otherwise, if $\Phi_1 \cap$

$\Phi_2 = \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 can be executed concurrently. Note, if Ξ_1 and Ξ_2 are protected by the same lock, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 will not be executed concurrently even if $\Phi_1 \cap \Phi_2 = \emptyset$.

Figure 2 illustrates the main steps of ARace. The numbers in the ring manifest the happen-before order of the steps. In this example, X and Y are shared variables accessed in the critical section. S1 indicates that this is a critical section. When S2 is executed, the shared variable X is protected to be read-only firstly (1). Then S2 can read the value of X (2). When S3 is executed, a new write buffer item, Y′ is allocated to cache the writes to Y (see details in next subsection) (3). When S4 is executed, Y′ in the write buffer is written back to Y (4), and X is unprotected to be writeable (5).

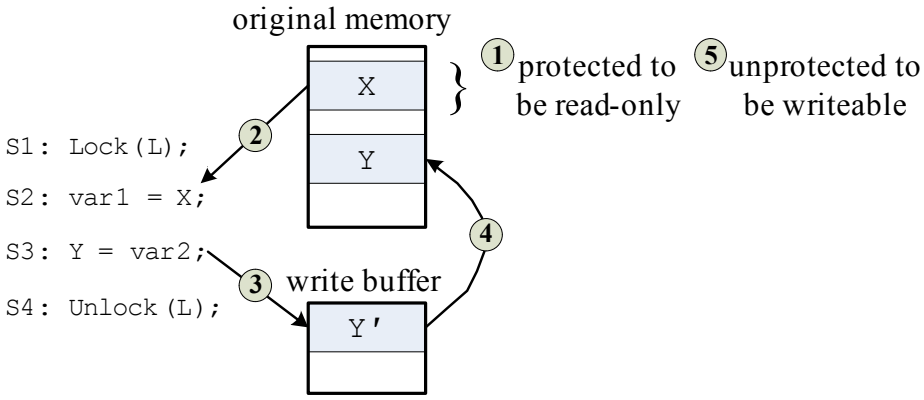


Figure 2. Main steps of ARace

Write Buffer

The write buffer is a thread private storage, allocated at thread starting and freed at thread exiting. It is constructed by write buffer items and is indexed by the memory addresses of shared variables. The size of each write buffer item is not fixed, and depends on the access size of instructions in Θ. Each ϕ in Φ written by instructions in Θ is mapped to a unique write buffer item. The write buffer item corresponding to ϕ is allocated at the first time that ϕ is written by some instruction γ in Θ. The size of the firstly allocated item is the same as the access size to ϕ in γ. In some programming languages, for example **C/C++**, it is allowed to access some bits of variables. Hence, ϕ may not be accommodated in the firstly allocated item. To address this problem, when the access size to ϕ

in instructions after γ is bigger than the size of previous allocated item, ARace will allocate a new write buffer item to accommodate the bigger size and copy the content from the old item to the new item. Then, the following accesses to ϕ are redirected to the new item.

Atomicity of Writing Back The write buffer item corresponding to ϕ is written back to ϕ when the last instruction in Θ is executed. If the process of writing back is not atomic, an inconsistent execution will be introduced. Figure 3 illustrates this situation. S3 and S4 read shared variable X and Y when S1 and S2 write back new values to X and Y. After this execution interleaving, var1 and var2 are respectively 1 and 0, which violates sequential consistence. To guarantee the atomicity of writing back, ARace protects all corresponding shared variables to be *unreadable* and *unwriteable* at the beginning of writing back, i.e. X and Y are protected to be unreadable and unwriteable before S1 and S2 are executed in this example.

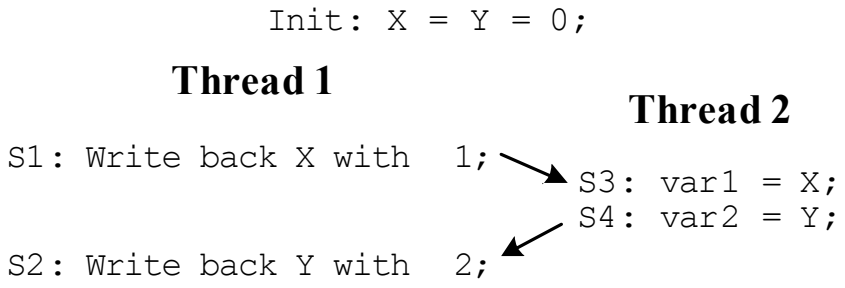


Figure 3. An example of writing back

After above protecting, ARace cannot write back write buffer items to corresponding shared variables directly. Fortunately, most modern operating systems, like Windows, UNIX, or Linux, support mapping the same physical memory at multiple virtual pages in a process's address space. To write back a write buffer item to corresponding shared variable ϕ, ARace allocates a new virtual page, called **swap page**, to map the physical page of original virtual page that contains ϕ. The swap page is both readable and writeable. ARace writes back the write buffer item corresponding to ϕ to the swap page with the same offset of ϕ in original virtual page. Actually, with the help of one swap page, ARace can write

back items whose corresponding shared variables lie in the same page, which is more efficient than writing back items one by one.

The protected shared variables are unprotected to be readable and writeable after the writing back process finishes. Then instructions not in Θ will read consistent status of shared variables. Besides, after the writing back process, the write buffer items allocated during Θ are freed for following executions of critical sections.

Shared Variable Protecting

To prevent instructions not in Θ from corrupting shared variable ϕ read by instructions in Θ, ϕ is protected to be read-only. When the last instruction in Θ is executed, ϕ is unprotected to be writeable. In most modern operating systems, memory is protected at a page granularity. Thus ARace has to protect the whole page that contains ϕ when it needs to protect ϕ. If ϕ lies in two pages, all of these two pages are protected. And the protected pages are unprotected to be writeable at the end of Θ.

False Sharing For two different critical sections $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$ and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, if $\Phi_1 \cap \Phi_2 = \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 can be executed concurrently. However, shared variable ϕ_1 in Φ_1 read by instructions in Θ_1 and ϕ_2 in Φ_2 read by instructions in Θ_2 may be allocated in the same page, called *p*. If Θ_1 and Θ_2 are executed by two different threads T_1 and T_2 concurrently, *p* will be protected repeatedly. More to the point, assuming T_1 finishes Θ_1 before T_2 finishes Θ_2, if T_1 unprotects *p* to be writeable at the end of Θ_1, Θ_2 will be at the risk of inconsistent.

To solve above false sharing problem, ARace uses a global shared structure, called *globalPage*, to record which pages have been protected to be read-only so far. Each protected page has a thread list *L* to record which threads have read shared variables in this page in critical sections. In addition, every thread in ARace has a local storage *S* to record the pages that contains shared variables it has read in critical sections. Algorithm 1 and Algorithm 2 respectively illustrate the processes of shared variable protecting and share variable unprotecting.

Algorithm 1. protect_sv (*t*, *v*, *size*)

Input: thread *t*, shared variable *v* read by *t*, and the *size* of *v*
Output: none

```
1: P = pages (v, size);
2: for each p in P do
3:   Lock (globalPage.Lock);
4:   if (p is not in globalPage) then
5:     protect p;
6:     add p to globalPage;
7:   end if
8:   add t to p.L;
9:   Unlock (globalPage.Lock);
10:  add p to t.S;
11: end for
```

Algorithm 2. unprotect_sv (*t*)

Input: thread *t* to exit a critical section
Output: none

```
1: for each p in t.S do
2:   Lock (globalPage.Lock);
3:   delete t from p.L;
4:   if p.L is empty then
5:     unprotect p;
6:     delete p from globalPage;
7:   end if
8:   Unlock (globalPage.Lock);
9:   delete p from t.S;
10: end for
```

Algorithm 3. redirect_access (*ins*, *t*)

Input: instruction *ins* in a critical section, thread *t* executing *ins*
Output: if *ins* accesses shared variable, memory address after redirecting

```
1: type = instruction_type (ins);
2: if ((type is Read_SV) or (type is Write_SV)) then
3:   addr = shared_variable_address (ins);
4:   size = shared_variable_size (ins);
5:   if (addr is in t.writebuffer) then
6:     if (size > t.writebuffer(addr).size) then
7:       allocate a new item in t.writebuffer for (addr, size);
8:       copy the content from old item to new item and free old item;
9:     end if
10:    return &t.writebuffer(addr);
11:  end if
12:  if ((type is Read_SV) and (type is not Write_SV)) then
13:    protect_sv (t, addr, size);
14:    return &addr;
15:  end if
16:  if ((type is not Read_SV) and (type is Write_SV)) then
17:    allocate a new item in t.writebuffer for (addr, size);
18:    return &t.writebuffer(addr);
19:  end if
20:  if ((type is Read_SV) and (type is Write_SV)) then
21:    protect_sv (t, addr, size);
22:    allocate a new item in t.writebuffer for (addr, size);
23:    copy the content from addr to new item;
24:    return &t.writebuffer(addr);
25:  end if
26: end if
```

When the page that contains ϕ is protected, instructions not in Θ can only read the content in this page. If there is an instruction not in Θ that tries to modify any content in this page, it will receive page fault exception. Then ARace suspends the thread in page fault handler. The suspended thread will resume its execution when the page is writeable.

Lazy Unprotecting If Θ is executed frequently, the page *p* containing ϕ is also protected and unprotected frequently. Actually, except instructions in Θ, if there is no instruction modifying any content in *p*, it does not need to unprotect *p* at the end of Θ. To utilize this feature, **ARace-LU** is proposed. ARace-LU is ARace with Lazy Unprotecting (LU). LU puts off unprotecting *p* until there is an instruction not in Θ that modifies contents in *p*. During this process, although Θ is executed multiple times, *p* is protected and unprotected only once.

Although LU will decrease the number of unnecessary protecting and unprotecting of *p*, it may also introduce additional page fault exceptions on *p*. For example, there are instructions not in Θ modifying contents in *p* after every Θ. The performance evaluation of ARace and ARace-LU will be presented later in this report.

Access Redirecting

ARace examines each instruction γ in Θ to check whether it accesses some shared variable ϕ in Φ. If yes, ARace will redirect the access. Algorithm 3 illustrates the process of access redirecting.

For most **RISC** architectures, like MIPS or Alpha, instructions have only two memory access types: reading and writing. But for **CISC** architectures, it is different. For example, instructions in IA-32 have three memory access types: reading, writing, and readwriting. The last access type means one instruction can read and then write the memory. The redirecting algorithm in ARace supports all access types in these architectures.

Lock Variable Mapping

Lock variables, like λ in Λ**,** are used to implement lock synchronizations. In most current popular programming languages, including **C/C++**, **Java**, and **C#**, programmers can define lock variables like normal variables. From the view of the compiler, lock variables have no difference with normal variables. Therefore, lock variable λ in Λ may be allocated in the same page with shared variable ϕ in Φ. If instructions in Θ read ϕ, ARace needs to protect

the page that contains ϕ to be read-only. Thus, λ is also protected to be read-only. Figure 4 illustrates this case.

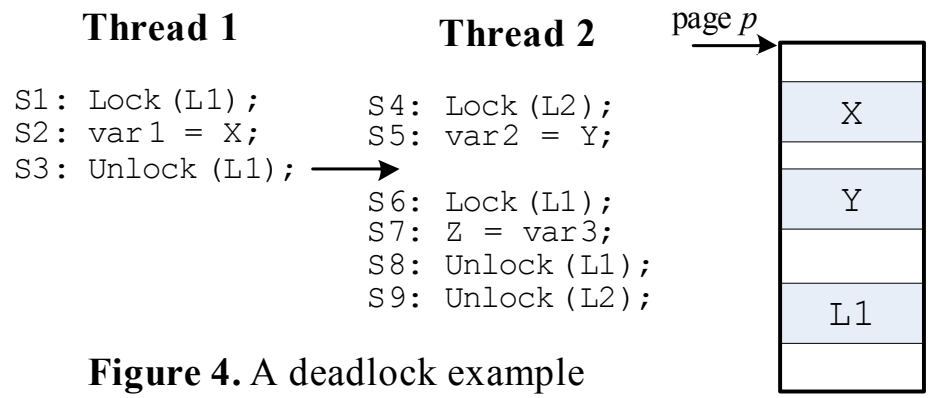


Figure 4. A deadlock example

In this example, $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$, where $\Lambda_1 = \{L1\}$, $\Phi_1 = \{X\}$, $\Gamma_1 = \{S1, S2, S3\}$, and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, where $\Lambda_2 = \{L1, L2\}$, $\Phi_2 = \{Y, Z\}$, $\Gamma_2 = \{S4, S5, S6, S7, S8, S9\}$. Because there is no branch type instruction in Γ_1 and Γ_2, Ξ_1 and Ξ_2 both have only sequential executions. Suppose they are respectively Θ_1 executed by T_1 and Θ_2 executed by T_2. Θ_1 and Θ_2 can be executed concurrently because $\Phi_1 \cap \Phi_2 = \emptyset$.

Assume that X, Y and L1 are allocated in the same page *p* as illustrated in Figure 4. Consider the following execution interleaving between Θ_1 and Θ_2: S1 is executed between S4 and S6. Then S6 has to wait for S3 to acquire lock L1. Due to the end of Θ_1, before S3 is executed, T_1 tries to unprotect *p* to be writeable. However, because of T_2, the thread list *L* of *p* is not null after erasing T_1. Thus *p* is still read-only when S3 is executed. L1 will not be released successfully until *p* is writeable, which means T_2 have finished Θ_2. However, if L1 cannot be acquired at S6, T_2 will not finish Θ_2. Therefore, a deadlock status happens.

To avoid this unintended deadlock status, ARace exploits a **Lock Variable Mapping Table** (**LVMT**) to map every lock variable λ in Λ to a new lock variable λ′, where $\lambda' \notin \Lambda$. λ′ has the same memory size with λ, and is in an independent memory region, which is always readable and writeable. LVMT is a one-to-one mapping table illustrated in Figure 5. Each term of LVMT has information for mapping: memory addresses of λ and λ′. When Lock/Unlock

instruction in Θ accesses λ, the memory address of λ is used to search LVMT to find λ′. Then λ is replaced by λ′, and the probability of deadlock status is eliminated.

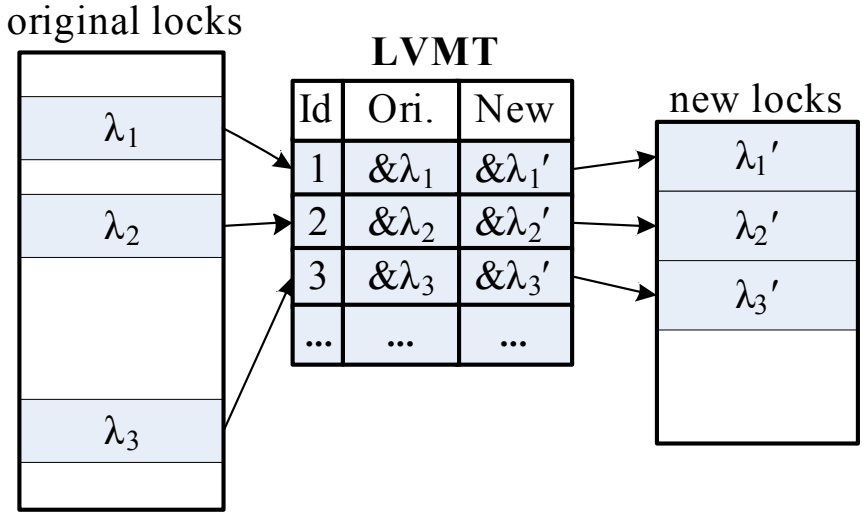


Figure 5. Lock Variable Mapping Table

Ad Hoc Synchronizations

In many multi-threaded programs, ad hoc synchronizations are widely used by developers. If one of the synchronization pairs is in a critical section, the ad hoc synchronization itself constructs an asymmetric race. Figure 6 is an example of this case. In this example, S3 and S6 construct an asymmetric race: AR(S3, S6).

Under ARace, AR(S3, S6) will not be triggered. But, thread 1 will never exit the loop if it executes S3 before thread 2 executes S6. That is because syncFlag belongs to the shared variable set of the critical section, and if thread 1 reads different values from syncFlag, the execution of the critical section will be inconsistent. Actually, shared variables like syncFlag are only used for ad hoc synchronizations. Thus there is no need to guarantee the consistent statuses of these variables in critical sections. ARace utilizes techniques proposed in a related work by W. Xiong et al. “Ad Hoc Synchronization Considered Harmful”, OSDI, 2010, to detect shared variables like syncFlag accessed in a critical section, and deletes them from the shared variable set of the critical section.

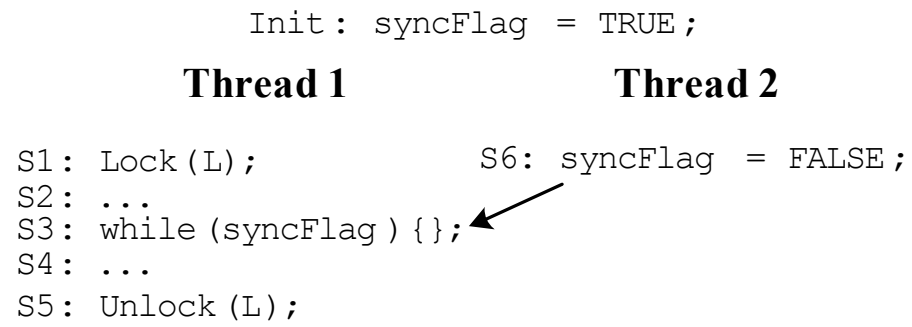


Figure 6. An asymmetric race with ad hoc synchronization

Implementation

We choose **Pin** to implement ARace. Pin is a dynamic binary instrumentation framework from Intel. The targets of Pin are the IA-32 and x86-64 instruction set architectures. It is extensively used in a plenty of research. Pin instruments programs at run time. Thus it needs no recompiling of programs.

ARace is implemented as a Pintool, including two main components: instrumentation engine and analysis engine. The instrumentation engine is used to instrument instructions and routines. The analysis engine contains access redirecting, write buffer, shared variable protecting, and lock variable mapping. Figure 7 illustrates the framework of the implementation.

The target multi-threaded programs are compiled on IA-32 architecture with pthreads library. The pthreads library is a widely used multi-threaded library. Although the platform and multi-threaded library are specific in our implementation, we believe that ARace scheme is general enough for other platforms and multi-threaded libraries.

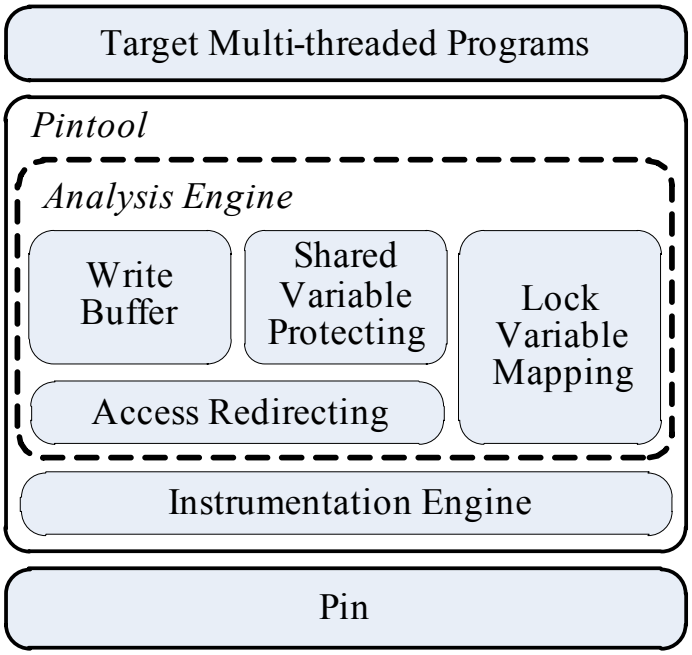


Figure 7. Implementation framework

Shared Variables

Because we have no any prior knowledge about that which variable is shared variable, a conservative policy is adopted: regarding all non-stack variables as shared variables. Although this policy may introduce some false positives, it does not affect the accuracy. In addition, this policy is more efficient than determining if a variable is a shared variable at run time.

Critical Sections & Lock Variables

In pthreads library, the points of entering and exiting a critical section are indicated by calling *pthread_mutex_lock* and *pthread_mutex_unlock* routines. For *pthread_mutex_trylock* routine, if the calling thread acquires the lock successfully, we also consider the following instructions are executed in a critical section.

Lock variables are those arguments passed to above routines with *pthread_mutex_t* structure in pthreads. The original lock variables passed to above routines are replaced by new lock variables via LVMT. So it is not the original lock variables but the new lock variables are really accessed in these routines. In our implementation, above three routines are all instrumented.

Moreover, current implementation of ARace utilizes techniques proposed by A. Jannesari and W. F. Tichy in “Identifying Ad-hoc Synchronization for Enhanced Race Detection.”, in IPDPS, 2010, to identify critical sections enclosed by user-defined Lock/Unlock calls.

Conditional Variables

Besides lock variables, conditional variables are another important class of synchronizations. Conditional variables are generally accessed in critical sections. Figure 8 is a typical example using conditional variable from application *radix* in SPLASH-2 [28]. In this example, the accesses to conditional variable C is protected by lock L. This creates an illusion that the critical section protected by the same lock can be executed concurrently.

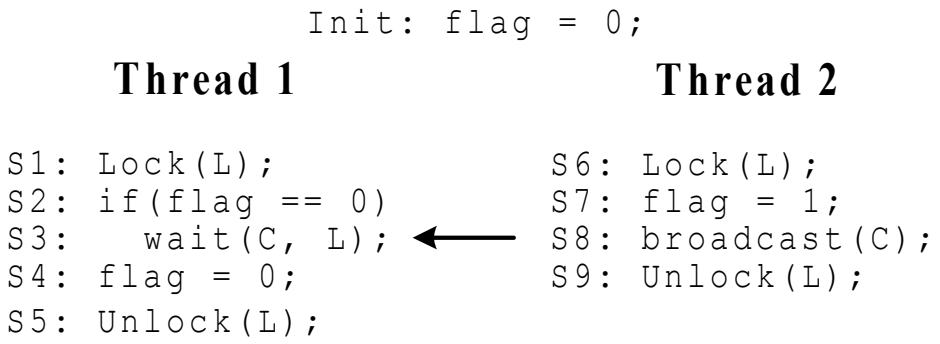


Figure 8. An example of conditional variable

In fact, the illusion is not true. The reason is that wait(C, L) is implemented as following:

Unlock (L);
Wait on C;
Lock (L);

Therefore, we just need to treat Unlock/Lock in conditional variable waiting operations as the point of critical section exiting or entering.

Critical Section Instrumentation

Instructions executed in critical sections are instrumented to redirect the accesses to shared variables. It is implemented by rewriting the operands of these instructions. Some instructions in

IA-32, like MOVS series, or CMPS series, have multiple memory operands. Thus we have to rewrite all memory operands of these instructions. The operands are converted from its original addressing mode to the base register addressing mode via Pin's scratch registers. A routine is inserted for each memory operand in one instruction to obtain the address after redirecting. Pin's scratch registers are filled up with the return value of this routine. Then the memory operands of this instruction are rewritten.

Routine Calls in Critical Sections

Routines called inside critical sections also need to be instrumented to redirect the accesses to shared variables, while there is no need to instrument routines called outside critical sections. In practice, the same routine may be called both inside and outside critical sections. If a routine is called outside critical sections at the first time, it will never be instrumented. That is because the routine used to instrument in Pin is executed only at the first time that the routine to be instrumented is executed.

To overcome this limitation, we define a rule for instrumenting routines: *once a routine has been executed in a critical section, it will always be instrumented, or it will never be instrumented*. We record a Boolean flag F_r for every routine *r*. F_r is initialized when *r* is called first time with the value if *r* is called in a critical section. If *r* is called in a critical section at the first time, its F_r is TRUE. Otherwise its F_r is FALSE.

All call instructions executed in a critical section are examined. For direct call instructions, the callee routine *r* is known at instrumenting time, and is fixed. Thus we just need to check F_r of *r*. If F_r is FALSE, the uninstrumented code cache of *r* in Pin is invalidated and the routine used to instrument in Pin is re-executed to instrument *r*. Then F_r is set to TRUE, which means *r* has been executed in some critical section. For indirect call instructions, the callee routine *r* is not fixed. Thus we insert a routine to obtain the callee routines. The inserted routine is executed every time the indirect call instruction is executed.

System Calls

System calls executed in a critical section may also access shared variables. For example,

```
Lock(L);
…
gettimeofday (&tv, NULL);
…
Unlock(L);
```

where *tv* is a shared variable defined in user space but accessed in kernel space. However, the address of *tv* should not be delivered to the kernel. That is because the page containing *tv* may have been protected to be read-only. If the address of *tv* is delivered to the kernel, when the kernel writes the system call result to *tv*, it will fail. This failure may never happen in executions without ARace. Beside system calls inside critical sections, system calls outside critical sections have the same problem.

To avoid unexpected failures of system calls, our implementation wraps system calls that access variables in user space. The real addresses delivered to the kernel are from new variables. If the system call is executed in a critical section and the original variable is shared, the new variable is allocated in the write buffer. And the system call result is written back along with other write buffer items. Otherwise, the new variable is allocated in an independent memory region that is always readable and writeable, and is written back to the original variable immediately after the execution of the system call.

Evaluation

Experimental Setup

We evaluate ARace with all 14 applications from SPLASH-2 and all 8 applications from Phoenix. For SPLASH-2 applications, we use their default inputs but increase the size to lengthen the runtime when necessary. Phoenix is a shared memory implementation of Google's MapReduce programming model for multi-core chips and shared-memory multiprocessors. The source code of Phoenix is

downloaded from the website. Every application in Phoenix has three versions: *MapReduce*, *Pthreads* and *Sequential*. We use the MapReduce version with the large dataset to evaluate ARace. Besides, we also use two real multi-threaded applications, Pbzip2 and Aget, to evaluate ARace.

To eliminate the impact of performance fluctuations due to random factors, each application from SPLASH-2 and Phoenix is tested for ten times, and the final result is the arithmetic average of these ten times.

All of our evaluations are conducted on a HP laptop computer with Intel(R) Core(TM)2 Duo CPU T7250 2.00 GHz, 2 MB L2 Cache, and 1 GB main memory. The operating system is 32 bit Fedora 14, which is a Red Hat-sponsored community project. The version of the Linux kernel is 2.6.35. The compiler is gcc with version 4.5.1. Applications from SPLASH-2 and Phoenix are complied with the default options in Makefiles. The two real applications are also complied with their default options. In addition, the performance is measured by the elapsed time via the command “time” when each application runs alone on the platform.

Result

Critical Section Characterization

TABLE 1: CRITICAL SECTION CHARACTERIZATION

Applications	#Lock Active	#Lock Total	#CS executed	#Inst per CS	#Read SV perl CS	#Write SV perl CS	#ReadWrite SV perl CS	%Inst in CS
cholesky	7	7	91	112.51	7.64	2.7	0	0.00
fft	1	1	2	553.5	9.5	1.5	0	0.00
lu-con	1	1	2	553.5	9.5	1.5	0	0.00
lu-non	1	1	2	549.5	6.5	1.5	0	0.00
radix	4	6	12	336.25	4.08	1.25	0	0.00
barnes	2049	2050	686646	265.68	15.54	15.57	0	0.33
fmm	2051	2052	330980	481.32	21.46	24.49	0.000012	0.21
ocean-con	2	6	2416	16.13	4.91	0.91	0	0.00
ocean-non	3	6	89044	15.33	4.77	0.77	0	0.00
radiosity	3914	3915	3212879	21.06	6.29	2.43	0	0.24
raytrace	5	5	196133	21.94	3.45	1.16	0	0.00
volrend	5	67	70766	25.2	4	1	0	0.02
water-nsquared	517	521	4130	277.8	58.49	8.93	0	0.06
water-spatial	70	70	2035	55.42	9.38	1.49	0	0.01
histogram	2	4	21718	61.51	8.95	2.98	0	0.02
kmeans	2	4	341715	129.68	13.17	5.21	2.427959	0.00
linear_regression	2	4	8538	60.76	8.88	2.94	0	0.00
matrix_multiply	2	4	369	83.14	7.43	3.4	0.897019	0.00
pca	2	4	7432	3349.73	192.74	475.16	19.12	0.08
reverse_index	2	4	6790	149.88	21.35	13.5	0	0.01
string_match	2	4	8537	243.12	12.76	3.91	2.91	0.00
word_count	4	7	2143	93.35	6.53	1.76	0	0.00

TABLE 1 presents the critical section characterization of applications from SPLASH-2 and Phoenix. The second and third columns are respectively the number of active lock and total lock. They represent lock variables used in critical sections, and lock variables initialized. These two columns show that there are locks initialized but not used. The fourth column shows the number of critical sections dynamically executed. Some applications, including *radiosity*, *barnes*, and *kmeans*, execute many critical sections. The fifth column is the number of dynamic instructions per critical section. The following three columns show the numbers of instructions reading, writing and readwriting shared variables per

critical section. And the last column shows the percentage of total dynamic instructions executed in critical sections.

Performance

TABLE 2: EXECUTION STATISTICS OF ARACE

Applications	#fault	#fault static	#fault dynamic	#invalidate	#page written back
cholesky	166	32	134	42	110
fft	4	2	2	13	3
lu-con	4	2	2	13	3
lu-non	2	1	1	13	3
radix	3	2	1	22	12
barnes	23470	2	23468	21	1599015
fmm	194838	7	194831	56	330038
ocean-con	18211	5	18206	13	23871
ocean-non	57898	13378	44520	13	66167
radiosity	1314821	4	1314817	29	3150186
raytrace	8599	14	8585	13	203094
volrend	45	13	32	37	70751
water-nsquared	9566	5	9561	13	4195
water-spatial	528	4	524	53	2587
histogram	9	5	4	14	43172
kmeans	504386	83560	420826	38	932763
linear_regression	10	7	3	14	16812
matrix_multiply	116	4	112	48	583
pca	26725	8	26717	42	36441
reverse_index	164047	6	164041	14	13316
string_match	8285	4	8281	41	25087
word_count	45	20	25	15	3758

In this section, we study the performance of ARace and ARace-LU on applications from SPLASH-2 and Phoenix. Figure 9 presents the performance results. All execution times are normalized to the runtime with Pin.

There are four bars for each application. The first bar is the normalized native runtime. The second bar is the base, runtime with Pin. The third and fourth bars respectively indicate the normalized runtime with ARace and ARace-LU. For applications that execute many critical sections except *radiosity*, ARace only

incurs about 4x overhead. But for *radiosity*, ARace incurs about 36x overhead, which is the worst case. On average, ARace incurs only about 1x overhead to the runs with Pin. This performance of ARace is competitive, especially for applications that require a high level of security.

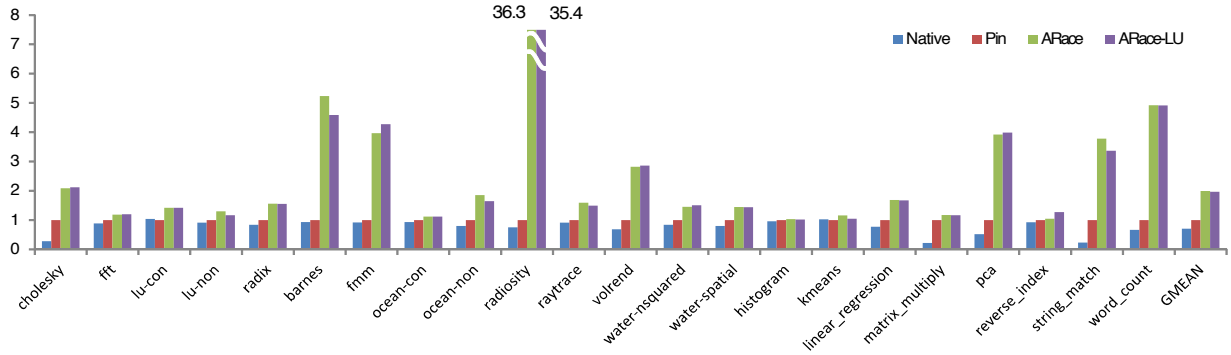


Figure 9. Normalized execution times of ARace and ARace -LU

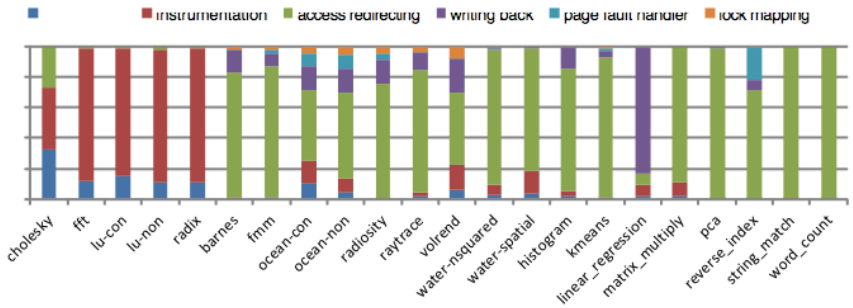


Figure 10. Overhead proportion of ARace

As expected, lazy unprotecting reduces the overhead of ARace for some applications, i.e. *barnes*, *radiosity*, *string_match*, etc. Unfortunately, it also increases the overhead for other applications, i.e. *fmm*, *reverse_index*, etc. This demonstrates that lazy unprotecting is mild for some applications but wild for some other applications.

TABLE II presents some execution statistics of ARace. The second column shows the number of page faults introduced by ARace. The third and fourth columns respectively indicate the number of page faults on static data and dynamic heap. For most applications, except *ocean-non* and *kmeans*, most of page faults happen on dynamic heap. The fifth column demonstrates that the amount of code cache invalidated by ARace is very tiny. The sixth column

presents the total number of pages that are written back at the end of critical sections. Except the first five applications, the numbers are large. The reason is that all writes to shared variables in critical sections are cached in the write buffer by ARace.

To study the overhead proportion of each component in ARace, we also gather the relative ratio of execution times of each component. Figure 10 presents the relative ratio of six components in ARace: *initialization*, *instrumentation*, *access redirecting*, *writing back*, *page fault handler* and *lock mapping*. The *initialization* work is done by Pin before the application starts. And the *page fault handler* is the handler that a thread executes when it receives a page fault exception. Except *cholesky* and *linear_regression*, the rest applications fall into two categories. In one category, the main part of the overhead is *instrumentation*, i.e. *fft*, *lu-con*, *lu-non*, and *radix*. In another category, the main part of the overhead is *access redirecting*, i.e. *barnes*, *radiosity*, *string_match*, etc. This difference results from the number of dynamically executed critical sections, which in second category is far more than that in first category. For *cholesky*, the number of executed critical sections is between the first category and the second category. Thus, the relative ratio of *instrumentation* and *access redirecting* nearly equals one. However, for *linear_regression*, the main part of the overhead is *writing back*. By studying the source code of this application, we found that it emits many shared intermediate statuses in a map callback function which is executed in a critical section. Thus ARace has to write back these statuses at the end of the critical section, which will introduce a lot of overhead. Figure 10 also shows that the proportion of the overhead introduced by *initialization*, *page fault hander* and *lock mapping* is not high.

Real Applications

Two real multi-threaded applications, Pbzip2 and Aget, are also used to evaluate ARace. Pbzip2 is a parallel implementation of the bzip2 file compressor. Aget is a multi-threaded http download accelerator [48]. To evaluate ARace, we use Pbzip2 to compress a 73MB file with tar format and download a 321MB file from a local

web server via Aget. These two applications are tested with 1, 2, 4, and 8 threads.

Effectiveness

During the evaluation of Pbzip2, ARace found a known real asymmetric race bug. The bug is illustrated in Figure 11. This bug takes place when thread 1 writes to fifo during thread 2 reading from fifo in the critical section guarded by the lock fifo->mut. ARace prevents this bug by protecting fifo to be read-only when thread 2 executes the critical section.

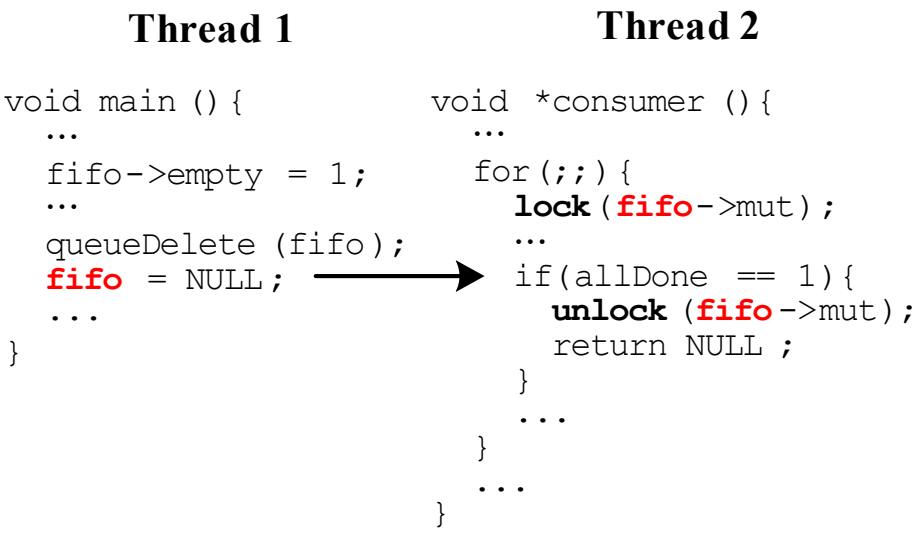


Figure 11. A real asymmetric race bug in Pbzip2

Performance

Figure 12 presents the execution time of the two applications. The results show that the overheads introduced by ARace are acceptable for real applications.

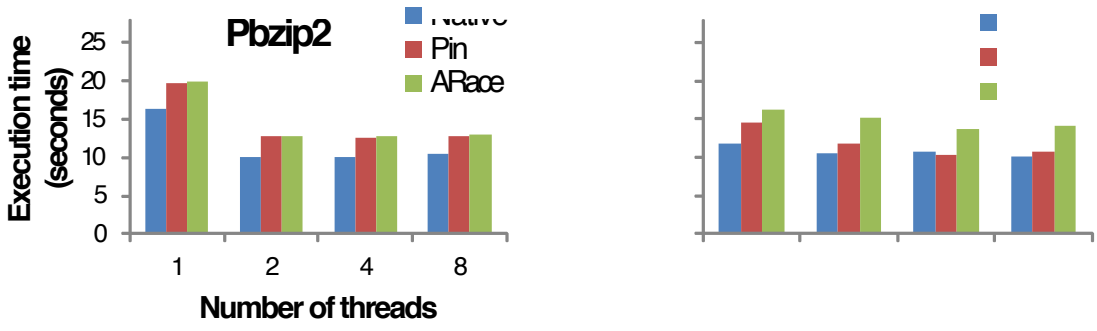


Figure 12. Performance of real applications

Conclusion

In this project, we propose a consistent execution model for critical sections in lock-based multi-threaded programs. Asymmetric races can never be triggered under this model. Based on this consistent execution model, a new software-based scheme ARace is presented to dynamically tolerate asymmetric races. Unlike previous schemes, ARace can guarantee the absence of asymmetric races. In addition, ARace can be directly applied to program binaries and requires no compiler support and application changes. We also present an implementation of ARace based on dynamic binary instrumentation. The results show that the performance of ARace is competitive.

To prohibit inconsistent statuses of shared variables, two critical sections accessing same shared variables are not allowed to be executed concurrently. Actually, ARace has no prior knowledge about the shared variable set of a critical section. One feasible solution is training ARace on-the-fly. At the first few times that a critical section is executed, ARace collects shared variables accessed in this critical section. During the training stage, critical sections are executed sequentially. After training, critical sections can be executed concurrently. We leave this work as part of future work.

Output

Wenwen Wang, Chenggang Wu, Paruj Ratanaworabhan, Xiang Yuan, Zhenjiang Wang, Jianjun Li, and Xiaobing Feng, “Dynamically Tolerating and Detecting Asymmetric Races”, *Journal of Computer Research and Development*, 51(8): 1748-1763, 2014.

Scopus

Author details

Ratanaworabhan, Paruj

Follow this Author

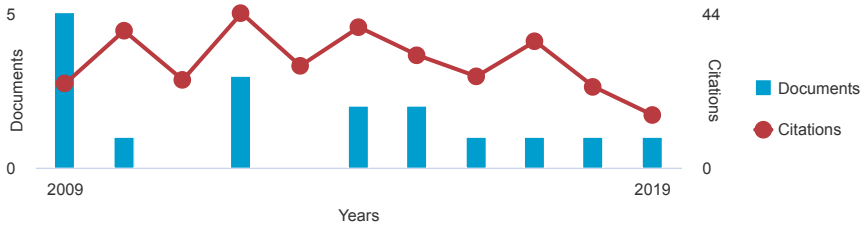
View potential author matches

Kasetsart University, Bangkok, Thailand
Author ID: 9637167700

Other name formats:

Subject area: Computer Science | Engineering | Mathematics | Medicine | Decision Sciences

Document and citation trends:



h-index: 7 View *h*-graph

Documents by author 22 Analyze author output

Total citations 361 by 292 documents

View citation overview

Get citation alerts + Add to ORCID Edit author profile

22 Documents | Cited by 292 documents | 27 co-authors | Author history | Topics

View in search results format ›

Sort on: Date (newest)

Export all Add all to list Set document alert Set document feed

Document title	Authors	Year	Source	Cited by
Exploiting Extra CPU Cores to Detect NOP Sleds Using Sandboxed Execution	Phringmongkol, N., Ratanaworabhan, P.	2019	10th International Conference on Information and Communication Technology for Embedded Systems, IC-ICTES 2019 - Proceedings 8695955	0
View abstract View at Publisher Related documents				
Using Document Classification to Improve the Performance of a Plagiarism Checker: A Case for Thai language documents	Sornsoontorn, C., Rimcharoen, S., Leelathakul, N., Kawtrakul, A., Ratanaworabhan, P.	2018	ICSEC 2017 - 21st International Computer Science and Engineering Conference 2017, Proceeding 8443906, pp. 219-223	0
View abstract View at Publisher Related documents				
A new approach to extracting sport highlight	Suksai, P., Ratanaworabhan, P.	2017	20th International Computer Science and Engineering Conference: Smart Ubiquitos Computing and Knowledge, ICSEC 2016 7859876	2
View abstract View at Publisher Related documents				
A parser generator using the grammar flow graph	Nakwijit, P., Ratanaworabhan, P.	2016	ICSEC 2015 - 19th International Computer Science and Engineering Conference: Hybrid Cloud Computing: A New Approach for Big Data Era 7401403	0
View abstract View at Publisher Related documents				

Document title	Authors	Year	Source	Cited by
A case for malware that make antivirus irrelevant	Thamsirarak, N., Seethongchuen, T., Ratanaworabhan, P.	2015	ECTI-CON 2015 - 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology 7206972	2

View abstract ⌄ View at Publisher Related documents

High-quality web-based volume rendering in real-time	Wangkaoom, K., Ratanaworabhan, P., Thongvigitmanee, S.S.	2015	ECTI-CON 2015 - 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology 7207091	3

View abstract ⌄ View at Publisher Related documents

Simple optimizations for LAMMPS	Kaewtes, V., Ratanaworabhan, P.	2014	2014 International Computer Science and Engineering Conference, ICSEC 2014 6978132, pp. 73-77	0

View abstract ⌄ View at Publisher Related documents

Dynamically tolerating and detecting asymmetric races	Wang, W., Wu, C., Ratanaworabhan, P., (...), Li, J., Feng, X.	2014	Jisuanji Yanjiu yu Fazhan/Computer Research and Development 51(8), pp. 1748-1763	1

Hide abstract ⌃ View at Publisher Related documents

Asymmetric races are a common type of data races. They are triggered when a thread accesses a shared variable in a critical section, and another thread accesses the same shared variable not in any critical section, or in a critical section guarded by a different lock. Asymmetric races in multi-threaded programs are usually harmful. To solve the problem introduced by asymmetric races, ARace is proposed. ARace utilizes shared variable protecting and write buffer to dynamically tolerate and detect asymmetric races. Shared variable protecting is used to protect shared variables that are read-only and read-before-write in critical sections, and these shared variables should not be modified out of critical sections; write buffer is used to buffer the writing operations to shared variables in critical sections. ARace can not only tolerate asymmetric races triggered by shared variable accesses in and out of critical sections, but also detect asymmetric races triggered by shared variable accesses in concurrent critical sections. ARace can be directly applied to binary code and requires neither additional compiler support nor hardware support. In addition, an implementation based on dynamic binary instrumentation is also proposed. The experimental results demonstrate that ARace guarantees the tolerance and detection of asymmetric races while incurring acceptable performance and memory overhead.

Functional cache simulator for multicore	Ratanaworabhan, P.	2012	2012 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, ECTI-CON 2012 6254278	2

View abstract ⌄ View at Publisher Related documents

Hardware support for enforcing isolation in lock-based parallel programs	Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B.	2012	Proceedings of the International Conference on Supercomputing pp. 301-310	1

View abstract ⌄ View at Publisher Related documents

Efficient Runtime Detection and Toleration of Asymmetric Races	Ratanaworabhan, P., Burtscher, M., Kirovski, D., (...), Nagpal, R., Pattabiraman, K.	2012	IEEE Transactions on Computers 61(4), pp. 548-562	4

View abstract ⌄ View at Publisher Related documents

gFPC: A self-tuning compression algorithm	Burtscher, M., Ratanaworabhan, P.	2010	Data Compression Conference Proceedings 5453485, pp. 396-405	7

View abstract ⌄ View at Publisher Related documents

Document title	Authors	Year	Source	Cited by
pFPC: A parallel compressor for floating-point data	Burtscher, M., Ratanaworabhan, P.	2009	Data Compression Conference Proceedings 4976448, pp. 43-52	8
View abstract ⌄ Related documents				
Detecting and tolerating asymmetric races	Ratanaworabhan, P., Burtscher, M., Kirovski, D., (...), Nagpal, R., Pattabiraman, K.	2009	ACM SIGPLAN Notices 44(4), pp. 173-184	5
View abstract ⌄ View at Publisher Related documents				
pFPC: A parallel compressor for floating-point data	Burtscher, M., Ratanaworabhan, P.	2009	Proceedings - 2009 Data Compression Conference, DCC 2009 4976448, pp. 43-52	2
View abstract ⌄ View at Publisher Related documents				
Detecting and tolerating asymmetric races	Ratanaworabhan, P., Burtscher, M., Kirovski, D., (...), Nagpal, R., Pattabiraman, K.	2009	Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP pp. 173-184	42
View abstract ⌄ View at Publisher Related documents				
FPC: A high-speed compressor for double-precision floating-point data	Burtscher, M., Ratanaworabhan, P.	2009	IEEE Transactions on Computers 58(1), pp. 18-31	95
View abstract ⌄ View at Publisher Related documents				
Program phase detection based on critical basic block transitions	Ratanaworabhan, P., Burtscher, M.	2008	ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software 4510734, pp. 11-21	8
View abstract ⌄ View at Publisher Related documents				
High throughput compression of double-precision floating-point data	Burtscher, M., Ratanaworabhan, P.	2007	Data Compression Conference Proceedings 4148768, pp. 293-302	52
View abstract ⌄ View at Publisher Related documents				
Load instruction characterization and acceleration of the BioPerf programs	Ratanaworabhan, P., Burtscher, M.	2006	Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC - 2006 4086135, pp. 71-79	4
View abstract ⌄ View at Publisher Related documents				

Dynamically Tolerating Asymmetric Races in Lock-Based Multi-threaded Programs

Wenwen Wang[1], Chenggang Wu[1], Paruj Ratanaworabhan[2], Jingling Xue[3], Xiang Yuan[1], Zhenjiang Wang[1], Jianjun Li[1]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]Faculty of Engineering, Kasetsart University
[3]School of Computer Science and Engineering, University of New South Wales
{wangwenwen, wucg, yuanxiang, wangzhenjiang, lijianjun}@ict.ac.cn, paruj.r@ku.ac.th, jingling@cse.unsw.edu.au

***Abstract*—Asymmetric races are data races caused when one thread accesses a shared variable guarded by a lock in a critical section and another thread accesses the same shared variable without holding the same lock. Asymmetric races are common and usually harmful. They often arise when well-tested code interacts with buggy legacy code or third-party libraries. Existing solutions for tolerating asymmetric races, whether based on software or hardware, have some limitations: they require either compiler support, or application changes, or new hardware to be added to commercial hardware platforms.**

This paper proposes a consistent execution model for critical sections in lock-based multi-threaded programs. During the consistent execution of a critical section, two conditions are satisfied: (1) shared variables read in the critical section are not written outside and (2) shared variables written in the critical section are not read and written outside. As a result, asymmetric races can never occur. Based on this consistent execution model, we present a new software-based scheme, called ARace, to dynamically ensure that all critical sections are consistently executed by exploiting write buffering and shared variable protection. ARace can be directly applied to binary code and requires neither additional compiler support nor application changes. We have implemented ARace based on dynamic binary instrumentation and evaluated it with the applications from SPLASH-2 and Phoenix. Our results show that ARace guarantees the absence of asymmetric races while incurring only about 1x overhead on average.

Keywords-asymmetric race; race tolerance; runtime supprot; dynamic instrumentation

I. INTRODUCTION

The existence of data races makes multi-threaded programs error-prone. When two threads access a shared variable without any synchronization, where one of the accesses is a write, a data race happens. Data races may cause multi-threaded programs to exhibit undesired behaviors. Some data races escaping from in-house testing may be catastrophic in the real world, as is the case for the Northeastern U.S. electricity blackout [33].

There has been a plenty of research on dealing with data races [1-4, 12, 13]. These research efforts fall into two categories: *prior detection* and *post tolerance*. The former detects and removes data races as aggressively as possible during in-house testing, while the latter tolerates data races in production runs. Despite extensive in-house testing, some data races still lurk around in released products [20]. Thus, the latter techniques are invaluable in practice.

There is one class of data races, called *asymmetric* races, which occur at the time when one thread accesses a shared variable inside a critical section protected by a lock and another thread also accesses the same shared variable due to a synchronization error (e.g., outside any critical section or inside a critical section but not protected by the same lock) [15]. Figure 1 illustrates an asymmetric race. In this example, `ptr` is a shared variable. The two reads to `ptr` in thread 1 are inside a critical section protected by a lock `L` but the write to `ptr` in thread 2 is not inside any critical section. This race may lead to inconsistent results when reading `ptr` in thread 1 during different program executions. This happens when the write to `ptr` in thread 2 takes place between the first read to `ptr` at `S2` and the second read to `ptr` at `S5` in thread 1.

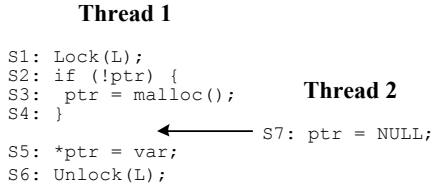


Figure 1. An asymmetric race

Asymmetric races are common in real applications and usually harmful [15]. Among the harmful data races found, about 20% are estimated to be asymmetric races [14]. The developer usually expects the accesses to shared variables to be made inside critical sections guarded by appropriate locks. Unfortunately, asymmetric races often arise when the developer's code interacts eventually with the other code from third-party libraries or legacy binaries. The latter code may be originally written only for single-threaded applications in mind. Thus, the presence of asymmetric races is often beyond the developer's control.

Although prior detection is useful for detecting and removing some asymmetric races, post tolerance can be more attractive. That is because many asymmetric races happen only when well-tested code interacts with legacy binaries or third-party libraries, whose source may be unavailable. In addition, due to their asymmetric nature, asymmetric races, which cannot be found during prior

detection, can be better prevented with post tolerance. A simple way to tolerate asymmetric races is to prevent another thread from accessing a shared variable if some thread is accessing it in a critical section, avoiding corrupting the shared variable.

There are three schemes specifically aiming to tolerate asymmetric races: software-based ToleRace [15], ISOLATOR [21], and hardware-based Pacman [14].

ToleRace makes two copies v' and v'' for each shared variable v accessed in a critical section when a thread T_1 executes the critical section. During the execution of the critical section, T_1 reads from and writes to v'. Meanwhile, another thread T_2 can read from and write to v outside the critical section. After T_1 has reached the end of the critical section, ToleRace compares the values of v and v'' and takes one of the following three actions: (1) if T_1 can be serialized before T_2, reserve the value of v; (2) if T_2 can be serialized before T_1, copy the value from v' to v; (3) if T_1 and T_2 cannot be serialized, interrupt the execution of the program. ToleRace can tolerate asymmetric races in the former two cases but is inadequate in the last case.

ISOLATOR copies each page that contains shared variables accessed in a critical section to a shadow page and protects the original page by making it inaccessible when a thread T_1 executes the critical section. Then T_1 operates only on the shadow page of each page in the critical section. If another thread T_2 tries to access an original page, then it will be instructed (via an exception) to wait for T_1. After T_1 has reached the end of the critical section, ISOLATOR copies the contents in every shadow page back to its original page and unprotects the original page by making it accessible again. ISOLATOR requires compiler support or even application changes so that pages can be shadowed appropriately. This scheme is ineffective unless the source of a program is available. Unfortunately, asymmetric races are often triggered when well-tested code interacts with legacy binaries or third-party libraries.

Pacman utilizes additional cache coherence hardware to protect variables accessed in a critical section. Compared with the two software-based schemes discussed above, this hardware-based scheme is uninstrusive and induces negligible slowdown. However, it is not yet supported by current commercial computer platforms.

To overcome the limitations inherent in the three aforementioned schemes, this paper proposes a consistent execution model for critical sections in lock-based multi-threaded programs. During the consistent execution of a critical section, two conditions are satisfied: (1) shared variables read in the critical section are not written outside and (2) shared variables written in the critical section are not read and written outside. As a result, asymmetric races can never occur. Based on this consistent execution model, we present a new software-based scheme, called **ARace**, to dynamically ensure that all critical sections are consistently executed by exploiting write buffering and shared variable protection. ARace works on-the-fly, requires neither additional compiler support nor application changes, and can be deployed even when the source code of a program is not available. We have implemented ARace based on dynamic binary instrumentation. Our results show that ARace guarantees the absence of asymmetric races with acceptable performance overhead.

There are fundamental differences between ARace and Transactional Memory (TM) [34]. Even though both use write buffers, ARace does not need to detect versions and conflicts during the execution of a critical section, because it protects the shared variables read in the critical section. When there are conflicts, TM must abort a transaction and rollback. During rollback, TM needs to handle side effect operations effectively, which is still an open problem. In contrast, there is no notion of abort-and-rollback in ARace, because its program executions are not speculative.

This paper makes following contributions:

- We propose a consistent execution model for critical sections in lock-based multi-threaded programs.
- We present a new software-based scheme ARace to dynamically tolerate asymmetric races.
- We describe an implementation of ARace and show that it is effective with small overhead.

The rest of this paper is organized as follows. Section II describes the background. Sections III and IV describe the architecture and implementation of ARace. Section V evaluates ARace. Section VI discusses related work. Finally, Section VII concludes and discusses future work.

II. BACKGROUND

In this section, we define a simple programming language and describe the consistent execution model.

A. Definitions

1) Multi-threaded Programs

A multi-threaded program is a quadruple $P = <\mathcal{A}, \Phi, \Psi, \mathcal{T}>$, where $\mathcal{A}$ is a finite set of lock variables: $\{\lambda_1, \lambda_2, \ldots\}$, Φ is a finite set of shared variables: $\{\varphi_1, \varphi_2, \ldots\}$, and Ψ is a finite set of threads: $\{T_1, T_2, \ldots\}$. Each thread T is a finite set of local variables: $\{\tau_1, \tau_2, \ldots\}$. $\mathcal{T}$ is a finite set of instructions: $\{\gamma_1, \gamma_2, \ldots\}$. The operand of an instruction can be a constant c or a variable. Each instruction in $\mathcal{T}$ belongs to one of the following operations:

- $\tau_i \leftarrow c$: writing local variable τ_i with the value of a constant c, where $\tau_i \in T_a$ and $T_a \in \Psi$;
- $\tau_i \leftarrow \varphi_j$: reading the value from shared variable φ_j to local variable τ_i, where $\varphi_j \in \Phi$, $\tau_i \in T_a$, and $T_a \in \Psi$;
- $\varphi_i \leftarrow \tau_j$: writing shared variable φ_i with the value of local variable τ_j, where $\varphi_i \in \Phi$, $\tau_j \in T_a$, and $T_a \in \Psi$;
- $\tau_i \leftarrow$ **ArithOrLogic**$(\tau_1, \tau_2, \ldots, \tau_j)$: arithmetic or logical operation on local variables $\tau_1, \tau_2, \ldots, \tau_j$ and writing the result to local variable τ_i, where $\tau_i, \tau_1, \tau_2, \ldots, \tau_j \in T_a$ and $T_a \in \Psi$;
- **BranchEQ**$(\gamma_k, \tau_i, \tau_j)$: branching next to be executed instruction to γ_k if the values of local variables τ_i and τ_j are equal, where $\gamma_k \in \mathcal{T}$, $\tau_i, \tau_j \in T_a$, and $T_a \in \Psi$;

- **Lock**(λ_i) : acquiring lock λ_i if it is not acquired by any thread in $\mathcal{Y}$, or blocking until it is released, where $\lambda_i \in \mathcal{A}$;
- **Unlock**(λ_i) : releasing lock λ_i if it is acquired by the thread that is executing this instruction, or blocking, where $\lambda_i \in \mathcal{A}$.

An execution of P is that threads in $\mathcal{Y}$ execute instructions in $\mathcal{T}$, denoted as Δ. Δ is an ordered list: $\delta_1 \prec \delta_2 \prec \ldots \prec \delta_m$, where any $\delta = T \mapsto \gamma$, $T \in \mathcal{Y}$, and $\gamma \in \mathcal{T}$. $T \mapsto \gamma$ means that thread T executes instruction γ. $\delta_i \prec \delta_j$ means that δ_i happens before δ_j. The execution space of P is the set of Δ: $\{\Delta_1, \Delta_2, \ldots\}$.

In this paper, we assume that properly synchronized multi-threaded programs adhere to the data-race-free 0 model [37]. With this model, the hardware appears sequentially consistent with respect to the programs even though it may be weakly ordered in reality.

2) Critical Sections

Now, we give the formal definition of a critical section based on above programming language. A critical section in P, where $P = <\mathcal{A}, \mathcal{\Phi}, \mathcal{Y}, \mathcal{T}>$, is a triple $\Xi = <\Lambda, \Phi, \Gamma>$, where Λ is the set of lock variables and $\Lambda \subseteq \mathcal{A}$; Φ is the set of shared variables and $\Phi \subseteq \mathcal{\Phi}$; Γ is the set of instructions and $\Gamma \subseteq \mathcal{T}$. An execution of Ξ is that some thread T in $\mathcal{Y}$ executes the instructions in Γ, denoted as Θ. Θ is also an ordered list: $\theta_1 \prec \theta_2 \prec \ldots \prec \theta_n$, where any $\theta = T \mapsto \gamma$, and $\gamma \in \Gamma$. For the convenience of description, we also say that thread T executes Θ and instruction γ is in Θ. Besides, if $\theta_1 = T \mapsto \gamma_i$ and $\theta_n = T \mapsto \gamma_j$, we can conclude that $\gamma_i = \text{Lock}(\lambda)$ and $\gamma_j = \text{Unlock}(\lambda)$, where $\lambda \in \Lambda$. In other words, any execution of a given critical section Ξ is enclosed by instructions acquiring and releasing some fixed lock λ. We say that Ξ is protected by lock λ, or each shared variable φ in Φ is protected by lock λ.

3) Asymmetric Races

Given a multi-threaded program P, where $P = <\mathcal{A}, \mathcal{\Phi}, \mathcal{Y}, \mathcal{T}>$, assume that two instructions γ_p and γ_q in $\mathcal{T}$ access the same shared variable φ in $\mathcal{\Phi}$ and at least one is writing to φ. If γ_p and γ_q satisfy any one of the following conditions, we say that they construct an asymmetric race, denoted as **AR**(γ_p, γ_q).

- γ_p is in some critical section in P, but there is no critical section in P that contains γ_q;
- γ_q is in some critical section in P, but there is no critical section in P that contains γ_p;
- γ_p and γ_q are in critical sections in P protected by different locks, but γ_p and γ_q are not in critical sections in P protected by the same lock.

All of the three requirements imply that at least one of γ_p and γ_q is in a critical section. Suppose γ_p (γ_q is the same) of AR(γ_p, γ_q) is in a critical section Ξ in P, where $\Xi = <\Lambda, \Phi, \Gamma>$ and $\gamma_p \in \Gamma$. Thus γ_q is either not in a critical section or in a critical section different with Ξ. Θ is an execution of Ξ by some thread T_a in $\mathcal{Y}$, and $\Theta = \theta_1 \prec \theta_2 \prec \ldots \prec \theta_n$. γ_p is firstly executed at θ_f in Θ: $\theta_f = T_a \mapsto \gamma_p$, where $1 < f < n$. Δ is an execution of P and $\Delta = \delta_1 \prec \delta_2 \prec \ldots \prec \delta_m$. Δ contains Θ, and $\theta_1 = \delta_i$, $\theta_n = \delta_j$, $\theta_f = \delta_k$, where $1 \le i < k < j \le m$. If there exists a δ in Δ satisfying: $\delta_k \prec \delta \prec \delta_j$, where $\delta = T_b \mapsto \gamma_q$ and $T_b \in \mathcal{Y}$, we say that AR(γ_p, γ_q) is *triggered* in Δ.

B. Consistent Execution Model

Given a critical section Ξ, where $\Xi = <\Lambda, \Phi, \Gamma>$, in a multi-threaded program P, where $P = <\mathcal{A}, \mathcal{\Phi}, \mathcal{Y}, \mathcal{T}>$, Θ is an execution of Ξ by some thread T_a in $\mathcal{Y}$, and $\Theta = \theta_1 \prec \theta_2 \prec \ldots \prec \theta_n$. Δ is an execution of P and $\Delta = \delta_1 \prec \delta_2 \prec \ldots \prec \delta_m$. Δ contains Θ, and $\theta_1 = \delta_i$, $\theta_n = \delta_j$, where $1 \le i < j \le m$. θ_f in Θ accesses shared variable φ: $\theta_f = T_a \mapsto \gamma_p$, where $1 < f < n$, $\varphi \in \Phi$, and $\gamma_p \in \Gamma$. Besides, $\theta_f = \delta_k$, where $i < k < j$. If the following two conditions are satisfied for any f and any φ, we say that Θ in Δ is *consistent* or Θ in Δ is a *consistent execution* of Ξ.

- If $\gamma_p = \tau_a \leftarrow \varphi$, where $\tau_a \in T_a$, then there is no δ in Δ satisfying following conditions: $\delta_k \prec \delta \prec \delta_j$, where $\delta = T_b \mapsto \gamma_q$, $\gamma_q = \varphi \leftarrow \tau_b$, $T_b \in \mathcal{Y}$, $\gamma_q \in \mathcal{T}$, and $\tau_b \in T_b$;
- If $\gamma_p = \varphi \leftarrow \tau_a$, where $\tau_a \in T_a$, then there is no δ in Δ satisfying following conditions: $\delta_k \prec \delta \prec \delta_j$, where $\delta = T_b \mapsto \gamma_q$, $\gamma_q = \varphi \leftarrow \tau_b$ or $\gamma_q = \tau_b \leftarrow \varphi$, $T_b \in \mathcal{Y}$, $\gamma_q \in \mathcal{T}$, and $\tau_b \in T_b$.

The first condition means that reading instructions in Θ always read consistent statuses of φ, because φ is not written by instructions not in Θ. The second condition means that any intermediate status of φ generated by writing instructions in Θ is not read and written by instructions not in Θ. If any one of the two conditions is not satisfied, we say Θ in Δ is *inconsistent* or Θ in Δ is an *inconsistent execution* of Ξ. Asymmetric races will be triggered in Δ if Θ in Δ is inconsistent. Note, Θ can be consistent in Δ_1, but inconsistent in Δ_2, where Δ_1 and Δ_2 are two different executions of P. It depends on Δ_1 and Δ_2.

From a programmer's perspective, the executions of a critical section are always consistent, because they are protected by the lock. But the existence of ill-behaved legacy code and third-party libraries destroys this consistence. They may access shared variables at any time, due to no mutually exclusive condition with the accesses in proper critical sections. These unsafe accesses will probably introduce harmful asymmetric races.

III. ARACE

A. Overview

ARace exploits two techniques to ensure that the execution Θ of a critical section Ξ is consistent, where $\Xi = <\Lambda, \Phi, \Gamma>$. The first is **Write Buffering**. The writes to any φ in Φ during Θ are redirected to the write buffer. The write buffer is written back to original shared variables when the last instruction in Θ is executed. By this way, the intermediate statuses of any φ in Φ generated by instructions in Θ are hidden, and instructions not in Θ can only see the final result of φ after Θ is finished.

Another technique utilized by ARace is **Shared Variable Protection**. Any φ in Φ read by instructions in Θ is protected to be read-only. When Θ is executed, if an

instruction not in Θ tries to modify φ after instructions in Θ have read φ, it will fail. Then it has to wait for the finish of Θ. Any protected φ is unprotected to be writeable when the last instruction in Θ is executed.

To prohibit inconsistent statuses of shared variables, ARace forbids two critical sections that access same shared variables from being executed concurrently. For two critical sections $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$ and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, if $\Phi_1 \cap \Phi_2 \neq \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 are not allowed to be executed concurrently. Otherwise, if $\Phi_1 \cap \Phi_2 = \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 can be executed concurrently. Note, if Ξ_1 and Ξ_2 are protected by the same lock, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 will not be executed concurrently even if $\Phi_1 \cap \Phi_2 = \emptyset$.

Figure 2 illustrates the main steps of ARace. The numbers in the ring manifest the happen-before order of the steps. In this example, `X` and `Y` are shared variables accessed in the critical section. `S1` indicates that this is a critical section. When `S2` is executed, the shared variable `X` is protected to be read-only firstly (1). Then `S2` can read the value of `X` (2). When `S3` is executed, a new write buffer item, `Y'`, is allocated to cache the writes to `Y` (see details in next subsection) (3). When `S4` is executed, `Y'` in the write buffer is written back to `Y` (4), and `X` is unprotected to be writeable (5).

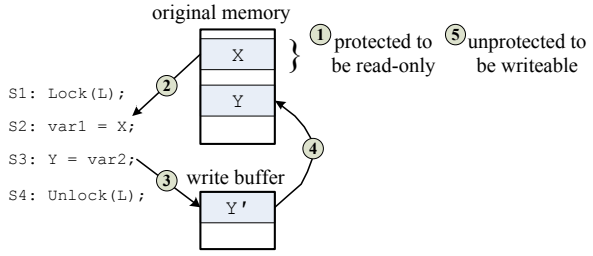


Figure 2. Main steps of ARace

B. *Write Buffer*

The write buffer is a thread private storage, allocated at thread starting and freed at thread exiting. It is constructed by write buffer items and is indexed by the memory addresses of shared variables. The size of each write buffer item is not fixed, and depends on the access size of instructions in Θ. Each φ in Φ written by instructions in Θ is mapped to a unique write buffer item. The write buffer item corresponding to φ is allocated at the first time when φ is written by some instruction γ in Θ. The size of the firstly allocated item is the same as the access size to φ in γ. In some programming languages, for example **C/C++**, it is allowed to access some bits of variables. Hence, φ probably cannot be accommodated in the firstly allocated item. To address this problem, when the access size to φ in the instruction after γ is bigger than the size of previous allocated item, ARace will allocate a new write buffer item to accommodate the bigger size and copy the contents from the old item to the new one. Then, the following accesses to φ are redirected to the new item.

Atomicity of Writing Back The write buffer item corresponding to φ is written back to φ when the last instruction in Θ is executed. If the process of writing back is not atomic, an inconsistent execution will be introduced. Figure 3 illustrates this situation. `S3` and `S4` read shared variables `X` and `Y` when `S1` and `S2` write back new values to `X` and `Y`. After this execution interleaving, `var1` and `var2` are respectively `1` and `0`, which is an inconsistent result. To guarantee the atomicity of writing back, ARace protects all corresponding shared variables to be *unreadable* and *unwriteable* at the beginning of writing back, i.e. `X` and `Y` are protected to be unreadable and unwriteable before `S1` and `S2` are executed in this example.

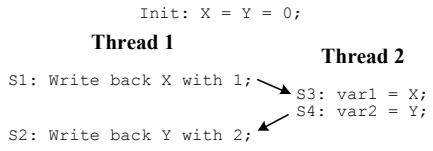


Figure 3. An example of writing back

After above protecting, ARace cannot write back write buffer items to corresponding shared variables directly. Fortunately, most modern operating systems, like UNIX, Linux, or Windows, support mapping the same physical memory at multiple virtual pages in a process's address space [26]. To write back a write buffer item to corresponding shared variable φ, ARace allocates a new virtual page, called **swap page**, to map the physical page of original virtual page that contains φ. The swap page is both readable and writeable. ARace writes back the write buffer item corresponding to φ to the swap page with the same offset of φ in original virtual page. Actually, with the help of one swap page, ARace can write back items whose corresponding shared variables lie in the same page, which is more efficient than writing back items one by one.

The protected shared variables are unprotected to be readable and writeable after the writing back process. Then instructions not in Θ will read a consistent status of shared variables. Besides, after the writing back process, the write buffer items allocated during Θ are freed for following executions of critical sections.

C. *Shared Variable Protecting*

To prevent instructions not in Θ from corrupting shared variable φ read by instructions in Θ, φ is protected to be read-only. When the last instruction in Θ is executed, φ is unprotected to be writeable. In most modern operating systems, memory is protected at a page granularity. Thus ARace has to protect the whole page that contains φ when it needs to protect φ. If φ lies in two pages, all of these two pages are protected. And the protected pages are unprotected to be writeable at the end of Θ.

False Sharing For two different critical sections $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$ and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, if $\Phi_1 \cap \Phi_2 = \emptyset$, then any Θ_1 of Ξ_1 and any Θ_2 of Ξ_2 can be executed concurrently. However, shared variable φ_1 in Φ_1 read by instructions in Θ_1 and φ_2 in Φ_2 read by instructions in Θ_2

Algorithm 1. protect_sv (*t*, *v*, *size*)

Input: thread *t*, shared variable *v* read by *t*, and the *size* of *v*
Output: none

```
1: P = pages (v, size);
2: for each p in P do
3:   Lock (globalPage.Lock);
4:   if (p is not in globalPage) then
5:     protect p to be read-only;
6:     add p to globalPage;
7:   end if
8:   add t to p.L;
9:   Unlock (globalPage.Lock);
10:  add p to t.S;
11: end for
```

Algorithm 2. unprotect_sv (*t*)

Input: thread *t* to exit a critical section
Output: none

```
1: for each p in t.S do
2:   Lock (globalPage.Lock);
3:   delete t from p.L;
4:   if p.L is empty then
5:     unprotect p to be writeable;
6:     delete p from globalPage;
7:   end if
8:   Unlock (globalPage.Lock);
9:   delete p from t.S;
10: end for
```

Algorithm 3. redirect_access (*ins*, *t*)

Input: instruction *ins* in a critical section, thread *t* executing *ins*
Output: if *ins* accesses shared variable, memory address after redirecting

```
1: type = instruction_type (ins);
2: if ((type is Read_SV) or (type is Write_SV)) then
3:   addr = shared_variable_address (ins);
4:   size = shared_variable_size (ins);
5:   if (addr is in t.writebuffer) then
6:     if (size > t.writebuffer(addr).size) then
7:       allocate a new item in t.writebuffer for (addr, size);
8:       if (type is Read_SV) then
9:         protect_sv (t, addr, size);
10:        copy the contents from addr to new item;
11:      end if
12:      copy the contents from old item to new item and free old item;
13:    end if
14:    return &t.writebuffer(addr);
15:  end if
16:  if ((type is Read_SV) and (type is not Write_SV)) then
17:    protect_sv (t, addr, size);
18:    return addr;
19:  end if
20:  if ((type is not Read_SV) and (type is Write_SV)) then
21:    allocate a new item in t.writebuffer for (addr, size);
22:    return &t.writebuffer(addr);
23:  end if
24:  if ((type is Read_SV) and (type is Write_SV)) then
25:    protect_sv (t, addr, size);
26:    allocate a new item in t.writebuffer for (addr, size);
27:    copy the contents from addr to new item;
28:    return &t.writebuffer(addr);
29:  end if
30: end if
```

may be allocated in the same page, called *p*. If Θ_1 and Θ_2 are executed by two different threads T_1 and T_2 concurrently, *p* will be protected repeatedly. More to the point, assuming T_1 finishes Θ_1 before T_2 finishes Θ_2, if T_1 unprotects *p* to be writeable at the end of Θ_1, Θ_2 will be at the risk of being inconsistent.

To solve above false sharing problem, ARace uses a global shared structure, called *globalPage*, to record which pages have been protected to be read-only so far. Each protected page has a thread list *L* to record which threads have read shared variables in this page in critical sections. In addition, each thread in ARace has a local storage *S* to record the pages containing shared variables that it has read in critical sections. Algorithm 1 and Algorithm 2 respectively illustrate the processes of shared variable protecting and share variable unprotecting.

When the page that contains φ is protected, instructions not in Θ can only read the contents in this page. If there is an instruction not in Θ that tries to modify any content in this page, it will receive a page fault exception. Then ARace suspends the thread that executes this instruction in page fault handler. The suspended thread will resume its execution when the page is writeable.

Lazy Unprotecting If Θ is executed frequently, the page *p* that contains φ is also protected and unprotected frequently. Actually, except instructions in Θ, if there is no instruction modifying any content in *p*, it does not need to unprotect *p* at the end of Θ. To utilize this feature, **ARace-LU** is proposed. ARace-LU is ARace with Lazy Unprotecting (LU). LU puts off unprotecting *p* until there is an instruction not in Θ that modifies the contents in *p*. During this process, although Θ is executed multiple times, *p* is protected and unprotected only once.

Although LU can decrease the number of unnecessary protecting and unprotecting of *p*, it may also introduce additional page fault exceptions on *p*. For example, there are instructions not in Θ modifying the contents in *p* after every Θ. The performance of ARace and ARace-LU are evaluated in Section V.

D. Access Redirecting

ARace examines each instruction γ in Θ to check whether it accesses some shared variable φ in Φ. If so, ARace will redirect the access. Algorithm 3 illustrates the process of access redirecting.

For most **RISC** architectures, like MIPS or Alpha, instructions have only two memory access types: reading and writing. But for **CISC** architectures, it is different. For example, instructions in IA-32 have three memory access types: reading, writing, and readwriting. The last access type means one instruction can read and then write the memory. The redirecting algorithm in ARace supports all access types in these architectures.

E. Lock Variable Mapping

Lock variables, like λ in Λ, are used to implement lock synchronizations. In most current popular programming languages, including **C**/**C++**, **Java**, and **C#**, programmers can define lock variables like normal variables. From the view of the compiler, lock variables have no difference with normal variables. Therefore, lock variable λ in Λ may be allocated in the same page with shared variable φ in Φ. If instructions in Θ read φ, ARace needs to protect the page that contains φ to be read-only. Thus, λ is also protected to be read-only. Figure 4 illustrates this case.

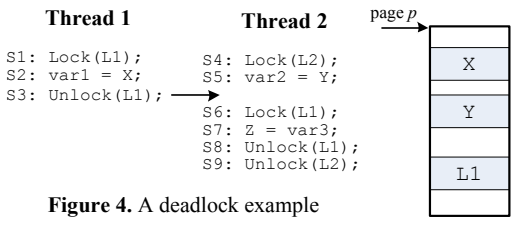


Figure 4. A deadlock example

In this example, $\Xi_1 = <\Lambda_1, \Phi_1, \Gamma_1>$, where $\Lambda_1 = \{L1\}$, $\Phi_1 = \{X\}$, $\Gamma_1 = \{S1, S2, S3\}$, and $\Xi_2 = <\Lambda_2, \Phi_2, \Gamma_2>$, where $\Lambda_2 = \{L1, L2\}$, $\Phi_2 = \{Y, Z\}$, $\Gamma_2 = \{S4, S5, S6, S7,$

S8, S9}. Because there is no branch type instruction in Γ_1 and Γ_2, Ξ_1 and Ξ_2 both have only sequential executions. Suppose they are respectively Θ_1 executed by T_1 and Θ_2 executed by T_2. Θ_1 and Θ_2 can be executed concurrently because $\Phi_1 \cap \Phi_2 = \emptyset$.

Assume that X, Y, and L1 are allocated in the same page *p* as illustrated in Figure 4. Consider the following execution interleaving between Θ_1 and Θ_2: S1 is executed between S4 and S6. Then S6 has to wait for S3 to acquire lock L1. Due to the end of Θ_1, before S3 is executed, T_1 tries to unprotect *p* to be writeable. However, because of T_2, the thread list *L* of *p* is not empty after erasing T_1. Thus *p* is still read-only when S3 is executed. L1 will not be released successfully until *p* is writeable, which means T_2 has finished Θ_2. However, if L1 cannot be acquired at S6, T_2 will not finish Θ_2. Therefore, a deadlock status happens.

To avoid this unintended deadlock status, ARace exploits a **Lock Variable Mapping Table** (**LVMT**) to map each lock variable λ in Λ to a new lock variable λ', where $\lambda' \notin \Lambda$. λ' has the same memory size with λ, and is in an independent memory region, which is always readable and writeable. LVMT is a one-to-one mapping table illustrated in Figure 5. Each term of LVMT has the information for mapping: memory addresses of λ and λ'. When Lock/Unlock instruction in Θ accesses λ, the memory address of λ is used to search LVMT to find λ'. Then λ is replaced by λ', and the probability of deadlock status is eliminated.

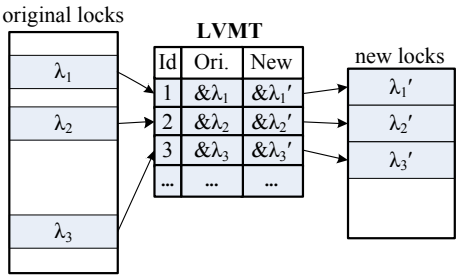


Figure 5. Lock Variable Mapping Table

F. Ad Hoc Synchronizations

In many multi-threaded programs, ad hoc synchronizations are widely used by developers [22]. If one of the synchronization pairs is in a critical section, the ad hoc synchronization itself constructs an asymmetric race. Figure 6 is an example of this case. In this example, S3 and S6 construct an asymmetric race: AR(S3, S6).

Under ARace, AR(S3, S6) will not be triggered. But, thread 1 will never exit the loop if it executes S3 before thread 2 executes S6. That is because syncFlag belongs to the shared variable set of the critical section, and if thread 1 reads different values from syncFlag, the execution of the critical section will be inconsistent. Actually, shared variables like syncFlag are only used for ad hoc synchronizations [22]. Thus there is no need to guarantee the consistent statuses of these variables in critical sections. ARace utilizes techniques proposed in [22, 31, 32] to detect shared variables like syncFlag accessed in a critical section, and deletes them from the shared variable set of the critical section.

```
Init: syncFlag = TRUE;

Thread 1                    Thread 2

S1: Lock(L);                S6: syncFlag = FALSE;
S2: ...
S3: while(syncFlag){};
S4: ...
S5: Unlock(L);
```

Figure 6. An asymmetric race with ad hoc synchronization

IV. IMPLEMENTATION

We choose **Pin** [27] to implement ARace. Pin is a dynamic binary instrumentation framework from Intel. The targets of Pin are the IA-32 and x86-64 instruction set architectures. It is extensively used in research work for dynamic program analysis. Pin instruments programs at run time. Thus it needs no recompiling of programs.

ARace is implemented as a Pintool, including two main components: instrumentation engine and analysis engine. The instrumentation engine is used to instrument instructions and routines. The analysis engine contains access redirecting, write buffer, shared variable protecting, and lock variable mapping. Figure 7 illustrates the framework of the implementation.

The target multi-threaded programs are compiled on IA-32 architecture with pthreads library, which is a widely used multi-threaded library. Although the platform and multi-threaded library are somewhat specific in our implementation, we believe that ARace scheme is general enough for other platforms and multi-threaded libraries.

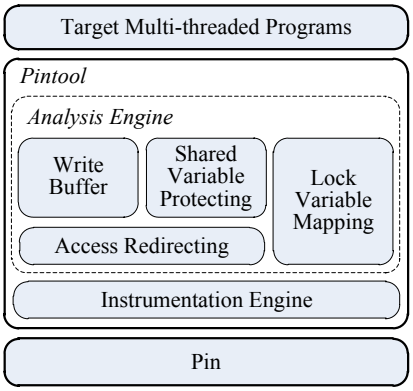


Figure 7. Implementation framework

A. Shared Variables

Because we have no any prior knowledge about that which variable is a shared variable, a conservative policy is adopted: regarding all non-stack variables as shared variables. Although this policy may introduce some false positives, it does not affect the accuracy. In addition, this policy is more efficient than determining if a variable is a shared variable at run time.

B. Critical Sections & Lock Variables

In pthreads library, the points of entering and exiting a critical section are indicated by calling *pthread_mutex_lock* and *pthread_mutex_unlock* routines. For *pthread_mutex_trylock* routine, if the calling thread acquires the lock successfully, we also consider the following instructions are executed in a critical section.

Lock variables are those arguments passed to above routines with *pthread_mutex_t* structure in pthreads. The original lock variables passed to above routines are replaced by the new lock variables via LVMT. So it is not the original lock variables but the new lock variables are really accessed in these routines. In our implementation, above three routines are all instrumented.

Moreover, current implementation of ARace utilizes techniques proposed in [31, 32] to identify critical sections enclosed by user-defined Lock/Unlock calls.

C. Conditional Variables

Besides lock variables, conditional variables are another important class of synchronizations. Conditional variables are generally accessed in critical sections. Figure 8 is a typical example using conditional variable from application *radix* in SPLASH-2 [28]. In this example, the accesses to conditional variable `C` are protected by a lock `L`. This creates an illusion that critical sections protected by the same lock can be executed concurrently.

```
                    Init: flag = 0;
        Thread 1                          Thread 2

S1: Lock(L);                      S6: Lock(L);
S2: if(flag == 0)                 S7: flag = 1;
S3:   cond_wait(C, L); ←          S8: cond_broadcast(C);
S4: flag = 0;                     S9: Unlock(L);
S5: Unlock(L);
```

Figure 8. An example of conditional variable

In fact, the illusion is not true. The reason is that `cond_wait(C, L)` is implemented as following:

Unlock (L);
Wait on C;
Lock (L);

Therefore, we just need to treat Unlock/Lock in conditional variable waiting operations as the point of critical section exiting or entering.

D. Critical Section Instrumentation

Instructions executed in critical sections are instrumented to redirect the accesses to shared variables. It is implemented by rewriting the memory operands of these instructions. Some instructions in IA-32, like `MOVS` series, or `CMPS` series, have multiple memory operands. Thus we have to rewrite all memory operands of these instructions. The memory operands are converted from their original addressing modes to the base register addressing mode via Pin's scratch registers. A routine is inserted for each memory operand in one instruction to obtain the address after redirecting. One of Pin's scratch registers is filled up with the return value of this routine. Then the memory operand of this instruction is rewritten.

E. Routine Calls in Critical Sections

Routines called inside critical sections also need to be instrumented to redirect the accesses to shared variables, while there is no need to instrument routines called outside critical sections. In practice, the same routine may be called both inside and outside critical sections. If a routine is called outside critical sections at the first time, it will never be instrumented. That is because the routine for instrumenting in Pin is executed only at the first time when the routine to be instrumented is executed.

To overcome this limitation, we define a rule for instrumenting routines: *once a routine has been executed in a critical section, it will always be instrumented, or it will never be instrumented*. We record a Boolean flag F_r for every routine r. F_r is initialized when r is called at the first time with the value if r is called in a critical section. If r is called in a critical section at the first time, its F_r is `TRUE`. Otherwise its F_r is `FALSE`.

All `call` instructions executed in a critical section are examined. For direct `call` instructions, the callee routine r is known at the instrumenting time, and is fixed. Thus we just need to check F_r of r. If F_r is `FALSE`, the uninstrumented code cache of r in Pin is invalidated and the routine for instrumenting in Pin is re-executed to instrument r. Then F_r is set to `TRUE`, which means r has been executed in some critical section. For indirect `call` instructions, the callee routine r is not fixed. Thus we insert a routine to obtain the callee routines. The inserted routine is executed every time the indirect `call` instruction is executed.

F. System Calls

System calls executed in a critical section may also access shared variables. For example,

Lock (L);
…
gettimeofday (&tv, NULL);
…
Unlock (L);

where *tv* is a shared variable defined in user space but accessed in kernel space. However, the address of *tv* should not be delivered to the kernel. That is because the page that contains *tv* may have been protected to be read-only. If the address of *tv* is delivered to the kernel, when the kernel writes the system call result to *tv*, it will fail. This failure may never happen in executions without ARace. Beside system calls inside critical sections, system calls outside critical sections also have the same problem.

To avoid these unexpected failures of system calls, our implementation wraps system calls that access variables in user space. The real addresses delivered to the kernel are from the new variables. If the system call is executed in a critical section and the original variable is shared, the new variable is allocated in the write buffer. And the system call result is written back along with other write buffer items. Otherwise, the new variable is allocated in an independent memory region that is always readable and

writeable, and is written back to the original variable immediately after the execution of the system call.

V. EVALUATION

A. Experimental Setup

We evaluate ARace with all 14 applications from SPLASH-2 [28] and all 8 applications from Phoenix [29]. For SPLASH-2 applications, we use their default inputs but increase the size to lengthen the runtime when necessary. Phoenix is a shared memory implementation of Google's MapReduce programming model for multi-core chips and shared-memory multiprocessors. The source code of Phoenix is downloaded from the website [30]. Each application in Phoenix has three versions: *MapReduce*, *Pthreads*, and *Sequential*. We use the MapReduce version with the large dataset to evaluate ARace. Besides, we also use two real multi-threaded applications, Pbzip2 [39] and Aget [48], to evaluate ARace.

To eliminate the impact of performance fluctuations due to random factors, each application from SPLASH-2 and Phoenix is tested for ten times, and the final result is the arithmetic average of these ten times.

All of our evaluations are conducted on a HP laptop computer with Intel(R) Core(TM)2 Duo CPU T7250 2.00 GHz, 2 MB L2 Cache, and 1 GB main memory. The operating system is 32 bit Fedora 14, which is a Red Hat-sponsored community project. The version of the Linux kernel is 2.6.35. The compiler is gcc with version 4.5.1. Applications from SPLASH-2 and Phoenix are complied with the default options in `Makefiles`. The two real applications are also complied with their default options. In addition, the performance is measured by the elapsed time via the command "`time -p`" when each application runs alone on the platform.

B. Critical Section Characterization

TABLE I. CRITICAL SECTION CHARACTERIZATION

Applications	#Lock active	#Lock total	#CS executed	#Inst per CS	#Read SV perl CS	#Write SV perl CS	#ReadWrite SV perl CS	%Inst in CS
cholesky	7	7	91	112.51	7.64	2.7	0	0.00
fft	1	1	2	553.5	9.5	1.5	0	0.00
lu-con	1	1	2	553.5	9.5	1.5	0	0.00
lu-non	1	1	2	549.5	6.5	1.5	0	0.00
radix	4	6	12	336.25	4.08	1.25	0	0.00
barnes	2049	2050	686646	265.68	15.54	15.57	0	0.33
fmm	2051	2052	330980	481.32	21.46	24.49	0.000012	0.21
ocean-con	2	6	2416	16.13	4.91	0.91	0	0.00
ocean-non	3	6	89044	15.33	4.77	0.77	0	0.00
radiosity	3914	3915	3212879	21.06	6.29	2.43	0	0.24
raytrace	5	5	196133	21.94	3.45	1.16	0	0.00
volrend	5	67	70766	25.2	4	1	0	0.02
water-nsquared	517	521	4130	277.8	58.49	8.93	0	0.06
water-spatial	70	70	2035	55.42	9.38	1.49	0	0.01
histogram	2	4	21718	61.51	8.95	2.98	0	0.02
kmeans	2	4	341715	129.68	13.17	5.21	2.427959	0.00
linear_regression	2	4	8538	60.76	8.88	2.94	0	0.00
matrix_multiply	2	4	369	83.14	7.43	3.4	0.897019	0.00
pca	2	4	7432	3349.7	192.74	475.16	19.12	0.08
reverse_index	2	4	6790	149.88	21.35	13.5	0	0.01
string_match	2	4	8537	243.12	12.76	3.91	2.91	0.00
word_count	4	7	2143	93.35	6.53	1.76	0	0.00

TABLE I presents the critical section characterization of applications from SPLASH-2 and Phoenix. The second and third columns are respectively the number of active locks and total locks. They represent lock variables used in critical sections, and lock variables only initialized. These two columns demonstrate that there are locks initialized but not used. The fourth column shows the total number of critical sections dynamically executed. Some applications, including *radiosity*, *barnes*, *kmeans*, and *fmm*, execute a plenty of critical sections. The fifth column is the average number of dynamic instructions per critical section. The following three columns present the average numbers of instructions reading, writing, and readwriting shared variables per critical section. And the last column shows the total percentage of dynamic instructions executed in critical sections.

C. Performance

TABLE II. EXECUTION STATISTICS OF ARACE

Applications	#fault	#fault static	#fault dynamic	#invalidate	#page written back
cholesky	166	32	134	42	110
fft	4	2	2	13	3
lu-con	4	2	2	13	3
lu-non	2	1	1	13	3
radix	3	2	1	22	12
barnes	23470	2	23468	21	1599015
fmm	194838	7	194831	56	330038
ocean-con	18211	5	18206	13	23871
ocean-non	57898	13378	44520	13	66167
radiosity	1314821	4	1314817	29	3150186
raytrace	8599	14	8585	13	203094
volrend	45	13	32	37	70751
water-nsquared	9566	5	9561	13	4195
water-spatial	528	4	524	53	2587
histogram	9	5	4	14	43172
kmeans	504386	83560	420826	38	932763
linear_regression	10	7	3	14	16812
matrix_multiply	116	4	112	48	583
pca	26725	8	26717	42	36441
reverse_index	164047	6	164041	14	13316
string_match	8285	4	8281	41	25087
word_count	45	20	25	15	3758

In this section, we study the performance of ARace and ARace-LU on applications from SPLASH-2 and Phoenix. Figure 9 presents the performance results. All execution times are normalized to the runtimes with Pin.

There are four bars for each application. The first bar is the normalized native runtime. The second bar is the base, runtime with Pin. The third and fourth bars respectively indicate the normalized runtime with ARace and ARace-LU. For applications that execute many critical sections except *radiosity*, ARace only incurs about 4x overhead. But for *radiosity*, ARace incurs about 35x overhead, which is the worst case. On average, ARace incurs only about 1x overhead to the run with Pin. This performance of ARace is competitive, especially for applications that require a high level of security.

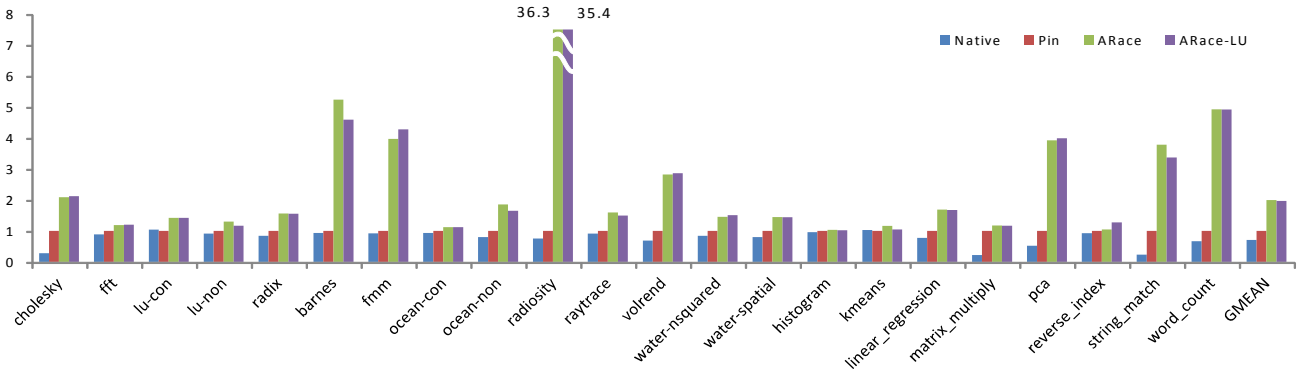


Figure 9. Normalized execution times of ARace and ARace-LU

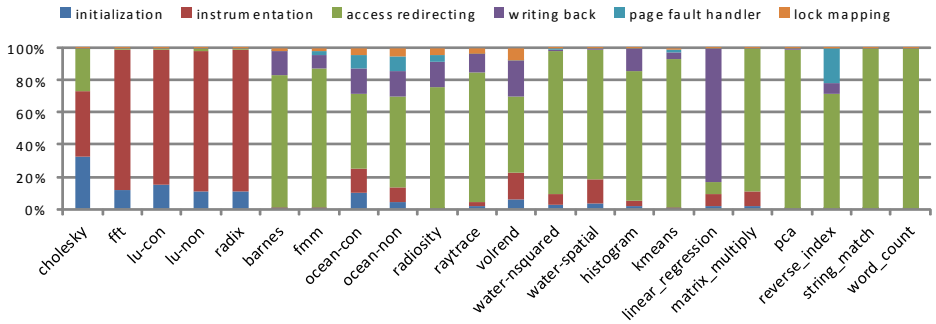


Figure 10. Overhead proportion of ARace

As expected, lazy unprotecting reduces the overhead of ARace for some applications, i.e. *barnes*, *radiosity*, *string_match*, etc. Unfortunately, it also increases the overhead for other applications, i.e. *fmm*, *reverse_index*, etc. This demonstrates that lazy unprotecting is mild for some applications but wild for some other applications.

TABLE II presents some execution statistics of ARace. The second column shows the total number of page faults introduced by ARace. The third and fourth columns respectively indicate the number of page faults on static data and dynamic heap. For most applications, except *ocean-non* and *kmeans*, most of page faults happen on dynamic heap. The fifth column demonstrates that the amount of code cache invalidated by ARace is very tiny. The sixth column presents the total number of pages that are written back at the end of critical sections. Except the first five applications, the numbers are large. The reason is that all writes to shared variables in critical sections are cached in the write buffer by ARace.

To study the overhead proportion of each component in ARace, we also gather the relative ratio of execution times of each component. Figure 10 presents the relative ratio of six components in ARace: *initialization*, *instrumentation*, *access redirecting*, *writing back*, *page fault handler*, and *lock mapping*. The *initialization* work is done by Pin before the application starts. And the *page fault handler* is the handler that a thread executes when it receives a page fault exception. Except *cholesky* and *linear_regression*, the rest applications fall into two categories. In one category, the main part of the overhead is *instrumentation*, i.e. *fft*, *lu-con*, *lu-non*, and *radix*. In another category, the main part of the overhead is *access redirecting*, i.e. *barnes*, *radiosity*, *string_match*, etc. This difference results from the number of dynamically executed critical sections, which in second category is far more than that in first category. For *cholesky*, the number of executed critical sections is between the first category and the second category. Thus, the relative ratio of *instrumentation* and *access redirecting* nearly equals one. However, for *linear_regression*, the main part of the overhead is *writing back*. By studying the source code of this application, we found that it emits many shared intermediate statuses in a callback function which is executed in a critical section. Thus ARace has to write back these statuses at the end of the critical section, which will introduce a lot of overhead. Figure 10 also shows that the proportion of the overheads introduced by *initialization*, *page fault hander*, and *lock mapping* is not high.

D. Real Applications

Two real multi-threaded applications, Pbzip2 and Aget, are also used to evaluate ARace. Pbzip2 is a parallel implementation of the bzip2 file compressor [39]. Aget is a multi-threaded http download accelerator [48]. To evaluate ARace, we use Pbzip2 to compress a 73MB file with `tar` format and download a 321MB file from a local web server via Aget. These two applications are tested with 1, 2, 4, and 8 threads.

1) Effectiveness

During the evaluation of Pbzip2, ARace found a known real asymmetric race bug. The bug is illustrated in Figure 11. This bug takes place when thread 1 writes to `fifo` during thread 2 reading from `fifo` in the critical section protected by the lock `fifo->mut`. ARace prevents this bug by protecting `fifo` to be read-only when thread 2 executes the critical section.

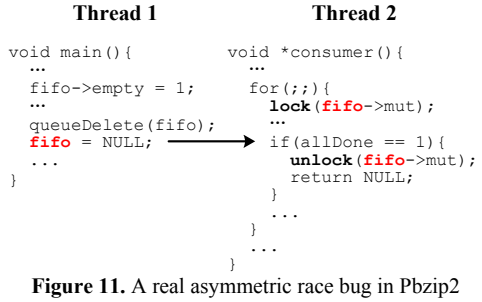


Figure 11. A real asymmetric race bug in Pbzip2

2) Performance

Figure 12 presents the execution times of the two applications. The results show that the overheads introduced by ARace are acceptable for real applications.

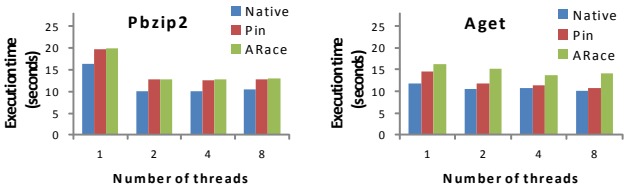


Figure 12. Performance of real applications

VI. Related Work

A. Asymmetric Races

ToleRace [15-17, 19] is the first proposed software scheme for detecting and tolerating asymmetric races. ToleRace copies two shadows, v' and v'', for each shared variable v accessed in a critical section when a thread T_1 executes the critical section. Then T_1 accesses v' in the critical section. At the same time, another thread T_2 can access v outside the critical section. After T_1 has reached the end of the critical section, ToleRace compares the values of v and v''. Then ToleRace decides which value of v and v' should be reserved as the new value of v: (1) if T_1 can be serialized before T_2, the value of v is reserved; (2) if T_2 can be serialized before T_1, the value of v' is reserved; (3) if T_1 and T_2 cannot be serialized, ToleRace has to interrupt the execution of the program. ToleRace can tolerate asymmetric races in the former two cases but is inadequate in the last case ([21] illustrates one such example). Compared with ToleRace, although there is performance penalty, ARace can tolerate these asymmetric races correctly. The reason is that T_2 is not allowed to access v when T_1 executes the critical section.

ISOLATOR [21] is another software scheme. At the beginning of a critical section, any page p that contains shared variables accessed in the critical section is copied to a shadow page p'. Then ISOLATOR protects p by making it inaccessible. The accesses to p in the critical section are redirected to p'. The accesses to p not in the critical section will cause page fault exceptions. At the end of the critical section, ISOLATOR copies the contents from p' to p, and unprotects p to be accessible. ISOLATOR needs compiler support or even application changes so that pages can be shadowed appropriately. In contrast, ARace has no such restriction, because it is directly applied to program binaries. Besides, for every shadow page, ISOLATOR uses a temporary page to copy it back. However, if there are multiple shadow pages, the atomicity of copying them back is not guaranteed in ISOLATOR. In ARace, we use another way to write back write buffer items, which can guarantee the atomicity of writing back.

Pacman [14] also aims to asymmetric races. The main difference between Pacman and above two schemes is that Pacman is based on hardware. Pacman exploits cache coherence hardware to protect cache lines that contain variables accessed in a critical section. If instructions not in the critical section try to access these cache lines, they will fail and have to wait. Pacman needs additional hardware support to exploit cache coherence. Besides, Pacman has no knowledge about critical sections. That is because critical sections have no difference with normal code from the perspective of hardware. Compared with software-based schemes, Pacman is uninstrusive and has negligible execution overhead. Nevertheless, it is not yet supported by current computer platforms.

B. Transactional Memory

Transactional Memory (TM) is another way to provide atomicity for lock-free data structures [34]. In TM, an atomic region is considered as a transaction and the transaction is executed speculatively. At the end of the transaction, TM checks whether there are conflicts. If so, TM aborts the transaction and rolls back to re-execute the transaction. Otherwise, the transaction is committed. TM needs to handle side effect operations effectively during rollback, which is still an open problem. TM can be implemented based on hardware [23, 26, 40, 42], software [50, 44, 45], or hybrid [63, 57, 52, 54]. The difference between ARace and TM is that ARace does not need speculative execution, rollback, version management, and timestamp support.

C. Data Race Detection

There is a large body of research focusing on data race detection, including static and dynamic. Static detections use program analysis techniques, like type-based checking [59, 55, 56, 58], static flow analysis [1, 2], or lockset analysis [35, 60]. One inherent drawback of static detections is that a lot of false positives are reported. Dynamic detections are mainly based on the lock-set algorithm [3, 51, 18, 20, 41], happens-before analysis [43, 36, 47], or hybrid of the two [24, 53, 49, 4]. Although dynamic detections have fewer false positives than static

detections, they have the challenge of coverage. Different with the prior detection, ARace is a post tolerance scheme.

D. Other Related Work

There are also some other related research work to facilitate debugging and diagnosing of multi-threaded programs, including studying concurrency bugs [6], classifying benign and harmful data races [10, 61], avoiding atomicity violations [7-9, 25, 38, 46], avoiding deadlock [62], and surviving or bypassing software failures [5, 11-13, 64]. Different with these techniques, ARace specially aims to dynamically tolerate asymmetric races in lock-based multi-threaded programs.

VII. Conclusions and Future Work

In this paper, we propose a consistent execution model for critical sections in lock-based multi-threaded programs. Asymmetric races can never be triggered under this model. Based on this consistent execution model, a new software-based scheme ARace is presented to dynamically tolerate asymmetric races. Unlike previous schemes, ARace can guarantee the absence of asymmetric races. In addition, ARace can be directly applied to program binaries and requires neither additional support from the compiler nor application changes. We also present an implementation of ARace based on dynamic binary instrumentation. The results show that the performance of ARace is competitive.

As described in section III, to prohibit inconsistent statuses of shared variables, two critical sections that access same shared variables are not allowed to be executed concurrently. Actually, ARace has no prior knowledge about the shared variable set of a critical section. One feasible solution is training ARace on-the-fly. At the first few times when a critical section is executed, ARace collects shared variables accessed in this critical section. During the training stage, critical sections are executed sequentially. After training, critical sections can be executed concurrently. In addition, with the help of prior knowledge about the share variable set of a critical section, ARace can protect shared variables read in the critical section at the point of entering the critical section, which will prevent a few potential races. We leave this work as part of future work.

Another future direction is to reduce the overhead introduced by ARace, for example, via dynamic program analysis or with the aid of data race detections.

References

[1] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.

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

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

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

[5] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *SOSP*, 2005.

[6] S. Lu, S. Park, E. Seo and Y. Zhou. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, 2008.

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

[8] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI : Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *SOSP*, 2007.

[9] B. Lucia, J. Devietti, K. Stauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.

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

[11] J. Yu and S. Narayanasamy. Tolerating Concurrency Bugs Using Transactions as Lifeguards. In *MICRO*, 2010.

[12] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*, 2011.

[13] J. Wu, H. Cui, and J. Yang. Bypassing Races in Live Applications with Execution Filters. In *OSDI*, 2010.

[14] S. Qi, N. Otsuki, L. O. Nogueira, A. Muzahid, and J. Torrellas. Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware. In *HPCA*, 2012.

[15] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and Tolerating Asymmetric Races. In *PPoPP*, 2009.

[16] P. Ratanaworabhan, D. Kirovski, and R. Nagpal. Efficient Runtime Detection and Toleration of Asymmetric Races. In *IEEE Trans. on Comput.*, Vol. 61, No. 4, 2012.

[17] P. Ratanaworabhan, M. Burtscher, D. Kirovshi, and B. Zorn. Hardware Supprot for Enforcing Isolation in Lock-Based Parallel Programs. In *ICS*, 2012.

[18] J. Erichson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, *http://usenix.org/event/osdi10/tech/slides/erickson.pdf*, 2010.

[19] D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. An Oracle for Tolerating and Detecting Asymmetric Races. Microsoft Research Technical Report MSR-TR-2007-122, 2007.

[20] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *ICSE*, 2011.

[21] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dyamically Ensuring Isolation in Concurrent Programs. In *ASPLOS*, 2009.

[22] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[23] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, 2008.

[24] A. Muzahid, D. S. Gracia, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

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

[26] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity using off-the-shelf Memory Protection Hardware. In *PPoPP*, 2009.

[27] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic instrumentation. In *PLDI*, 2005.

[28] S. Woo, M. Ohara, E. Torrie, J. Singh and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[29] C. Ranger, R. Raghuraman, A. Penmestsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, 2007.

[30] The Phoenix System for MapReduce Programming. *http://mapreduce.stanford.edu/*

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

[32] A. Jannesari and W. F. Tichy. Identifying Ad-hoc Synchronization for Enhanced Race Detection. In *IPDPS*, 2010.

[33] Software Bug Contributed to Blackout. SecurityFocus. *http://www.securityfocus.com/news/8032*

[34] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*, 1993.

[35] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.

[36] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, 1991.

[37] S. V. Adve and M. D. Hill. Weak Ordering – A New Definition. In *ISCA*, 1990.

[38] R. Agarwal, A. Sasturkar, L. Wang, and S. Stoller. Optimized Run-Time Race Detection and Atomicity Checking Using Partial Discovered Types. In *ASE*, 2005.

[39] Parallel BZIP2. *http://compression.ca/pbzip2*

[40] M. Lupon, G. Magklis, A. Gonzalez. A Dynamically Adaptable Hardware Transactional Memory. In *MICRO*, 2010.

[41] X. Xie and J. Xue. AccuLock: Accurate and Efficient Detection of Data Races. In *CGO*, 2011.

[42] B. Khan, M. Horsnell, M. Lujan, and I. Watson. Scalable Object-Aware Hardware Transactional Memory. In *Euro-Par*, 2010.

[43] E. Schonberg. On-the-fly Detection of Access Anomalies. In *PLDI*, 1989.

[44] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM^{2}C: A Software Transactional Memory for Many-Cores. In *EuroSys*, 2012.

[45] B. Saha, A. Adi-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *MICRO*, 2006.

[46] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Using Data Structure Knowledge for Efficient Lock Generation and Strong Atomicity. In *PPoPP*, 2010.

[47] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. Helgrind+: An Efficient Dynamic Race Detector. In *IPDPS*, 2009.

[48] Aget: Multithreaded HTTP Download Accelerator. *http://www.enderunix.org/aget*

[49] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*, 1991.

[50] Z. He, X. Yu, and B. Hong. Profiling-based Adaptive Contention Management for Software Transactional Memory. In *IPDPS*, 2012.

[51] C. von Praun and T. Gross. Object Race Detection. In *OOPSLA*, 2001.

[52] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *PPoPP*, 2006.

[53] R. O'Callahan and J. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, 2003.

[54] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *ASPLOS*, 2006.

[55] C. Flanagan and S. N. Freund. Type-based Race Detection for Java. In *PLDI*, 2000.

[56] H. A. Andrade and B. Sanders. An Approach to Compositional Model Checking. In *IPDPS*, 2002.

[57] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *ISCA*, 2007.

[58] C. Boyapati and M. C. Rinarad. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[59] D. Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, 2003.

[60] N. Sterling. Warlock: A Static Data Race Analysis Tool. In *USENIX Winter Technical Conference*, 1993.

[61] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[62] K. Agrawal, J. Buhler, P. Li, and R. Chamberlain. Efficient Deadlock Avoidance for Streaming Computations with Filtering. In *PPoPP*, 2012.

[63] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *SPAA*, 2011.

[64] J. Oh, C. J. Hughes, G. Venkataramani, M. Prvulovic. LIME: A Framework for Debugging Load Imbalance in Multi-threaded Execution. In *ICSE*, 2011.

计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2014.20130123
51(8):1748-1763,2014

动态容忍和检测非对称数据竞争

王文文[1,2] 武成岗[1] Paruj Ratanaworabhan[3] 远 翔[1,2] 王振江[1] 李建军[1] 冯晓兵[1]
[1](计算机系统结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)
[2](中国科学院大学 北京 100049)
[3](农业大学工程学院 泰国曼谷 10900)
(wangwenwen@ict.ac.cn)

Dynamically Tolerating and Detecting Asymmetric Races

Wang Wenwen[1,2], Wu Chenggang[1], Paruj Ratanaworabhan[3], Yuan Xiang[1,2], Wang Zhenjiang[1], Li Jianjun[1], and Feng Xiaobing[1]
[1](*Key Laboratory of Computer System and Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*Faculty of Engineering*, *Kasetsart University*, *Bangkok*, *Tailand* 10900)

Abstract Asymmetric races are a common type of data races. They are triggered when a thread accesses a shared variable in a critical section, and another thread accesses the same shared variable not in any critical section, or in a critical section guarded by a different lock. Asymmetric races in multi-threaded programs are usually harmful. To solve the problem introduced by asymmetric races, ARace is proposed. ARace utilizes shared variable protecting and write buffer to dynamically tolerate and detect asymmetric races. Shared variable protecting is used to protect shared variables that are read-only and read-before-write in critical sections, and these shared variables should not be modified out of critical sections; write buffer is used to buffer the writing operations to shared variables in critical sections. ARace can not only tolerate asymmetric races triggered by shared variable accesses in and out of critical sections, but also detect asymmetric races triggered by shared variable accesses in concurrent critical sections. ARace can be directly applied to binary code and requires neither additional compiler support nor hardware support. In addition, an implementation based on dynamic binary instrumentation is also proposed. The experimental results demonstrate that ARace guarantees the tolerance and detection of asymmetric races while incurring acceptable performance and memory overhead.

Key words asymmetric race; tolerating and detecting; write buffer; page protecting; dynamic binary instrumentation

摘 要 非对称数据竞争是数据竞争中一种常见的类型.当一个线程在临界区内访问某个共享变量,另外一个线程在临界区外或不同的临界区内同时也访问这个共享变量时,就触发了非对称数据竞争.多线程程序中的非对称数据竞争往往是有害的.为了解决非对称数据竞争引入的问题,提出了 ARace.它使用共享变量保护和写缓冲区来动态容忍和检测非对称数据竞争.其中,共享变量保护用于保护临界区内

收稿日期:2013-02-01;**修回日期**:2013-06-26
基金项目:国家"八六三"高技术研究发展计划基金项目(2012AA010901);国家自然科学基金青年科学基金项目(61100011);国家自然科学基金杰出青年科学基金项目(60925009)

只读和先读后写的共享变量,防止这些变量在临界区外被修改;写缓冲区用于缓存临界区内对共享变量的写操作.ARace 不仅可以容忍临界区内和临界区外之间的非对称数据竞争,还可以对并发临界区之间的非对称数据竞争进行检测.ARace 既不依赖程序源代码和编译器的支持,也不依赖额外硬件的支持.此外,还提出了一种通过动态二进制插桩技术实现 ARace 的方法.实验结果表明,ARace 在保证容忍和检测非对称数据竞争的同时,并未引入很大的性能开销和内存开销.

关键词 非对称数据竞争;容忍和检测;写缓冲区;页保护;动态二进制插桩

中图法分类号 TP311

近年来,随着多核处理器的应用,在桌面电脑和移动终端上,多线程程序的应用越来越普及.虽然多线程程序可以充分利用多核处理器的资源,发挥多核处理器的优势,但是编写和调试这类程序却非常困难.这是因为:1)在编写和调试传统串行程序过程中存在的困难,在多线程程序中依然存在;2)多线程程序还会引入一些新的导致程序出错的并发错误(concurrency bug),如:数据竞争、死锁、原子性冲突以及顺序冲突等.这使得多线程程序的开发和调试相对于传统串行程序更加困难.

数据竞争的存在使多线程程序很容易出错.所谓数据竞争,是指两个线程在没有任何同步的情况下,同时访问相同的共享变量,并且其中至少有一个是写操作.数据竞争可能会使多线程程序表现出非程序员预期的行为.有些隐藏在已发布产品中的数据竞争还会引发实际生活中的灾难性事故,例如 20 世纪 80 年代放射医疗设备 Therac-25 导致的医疗事故[1],2003 年北美地区的大规模停电事件[2]以及 2012 年纳斯达克的 Facebook 股票交易问题[3]等,导致它们发生的故障根源均是隐藏在软件产品中的数据竞争.

在数据竞争中,有一类是非对称数据竞争[4].所谓非对称数据竞争,就是存在数据竞争的两个访问,其中一个在临界区内,而另一个不在临界区内,或者它们在不同锁保护的临界区内.图 1 给出了一个非对称数据竞争的实例,其中 *ptr* 是共享变量,线程 T_1 对 *ptr* 的访问在锁 L 保护的临界区内,线程 T_2 对 *ptr* 的赋值不在任何临界区内.当 T_1 在临界区内访问 *ptr* 时,T_2 仍可以对它进行修改.如果 T_2 对 *ptr* 的修改在 T_1 对 *ptr* 的两次访问之间(图 1 中箭头所示),那么 T_1 将会读到错误的 *ptr*,导致程序错误.

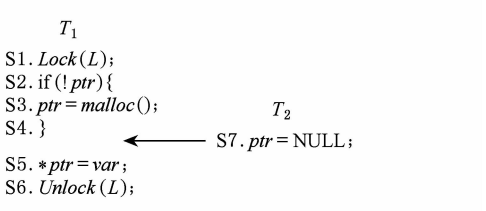


Fig. 1 Asymmetric race example.

图 1 非对称数据竞争实例

非对称数据竞争是很常见的数据竞争类型,并且一般会导致程序出现错误[4].文献[5]经过调查发现,在常见的有危害性的数据竞争中,20%左右均是非对称数据竞争.一般来说,程序员在开发多线程程序时,会尽可能使用合适的锁对共享变量的访问进行保护.但是,当程序代码量非常大时,程序员往往会出现遗漏或误加锁的情况.例如文献[6]对非死锁的并发错误进行统计发现,27%左右的错误均是由于程序员遗漏或误加锁导致的.另外,即使某一应用程序已经经过了严格的测试,当它与遗产代码和第三方库代码交互时,也会由于遗产代码和第三方库代码的不安全性,触发非对称数据竞争.这些不安全的代码在初始设计时可能仅仅针对传统的串行程序.此外,现代大型软件开发过程中的模块化设计技术和软件复用技术,也使得非对称数据竞争不可避免.

存在非对称数据竞争的多线程程序在执行过程中,并不一定会触发非对称数据竞争.这是因为,在多线程程序中,不同线程之间的执行交叠存在不确定性.图 2 给出了图 1 中的非对称数据竞争在 3 种不同的线程执行交叠下的触发情况.

在图 2(a)中,当线程 T_1 在临界区内对 *ptr* 进行访问时,线程 T_2 对 *ptr* 的修改触发了非对称数据竞争;而在图 2(b)(c)中,线程 T_2 对 *ptr* 的修改分别发生在线程 T_1 执行临界区之前和之后,因此未触发非对称数据竞争.

所谓动态容忍非对称数据竞争,就是在多线程程序执行过程中,通过限制不同线程之间的执行交叠,消除非对称数据竞争触发的可能性.如在图 2 中,限制线程的执行交叠为图 2(b)或图 2(c),就可以实现对这个非对称数据竞争的动态容忍.

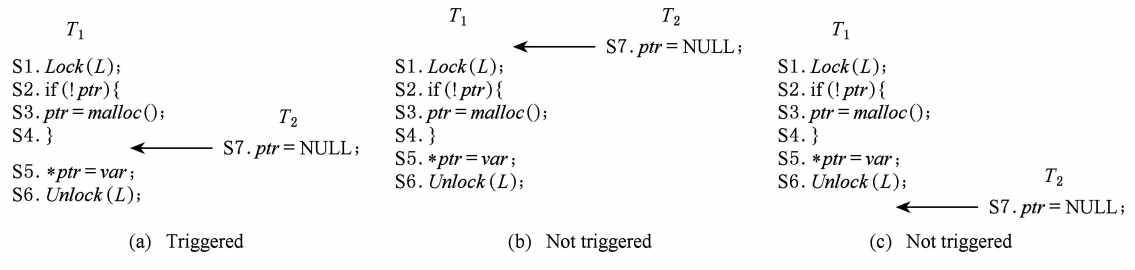


Fig. 2 Different interleavings of asymmetric race.

图 2 不同执行交叠导致的非对称数据竞争触发情况

为了解决基于锁的多线程程序中非对称数据竞争引入的问题,本文提出了一种动态容忍和检测非对称数据竞争的方法:ARace.它使用共享变量保护和写缓冲区来实现对非对称数据竞争的动态容忍和检测.共享变量保护用于保护临界区内只读和先读后写的共享变量,防止这些变量在临界区外被修改;写缓冲区用于缓存临界区内对共享变量的写操作.ARace 不仅可以容忍临界区内和临界区外之间的非对称数据竞争,还可以对并发临界区之间的非对称数据竞争进行检测.ARace 既不依赖程序源代码和编译器的支持,也不依赖额外硬件的支持.此外,本文还提出了一种通过动态二进制插桩技术实现 ARace 的方法.实验结果表明,ARace 在保证容忍和检测非对称数据竞争的同时,并未引入很大的性能开销和内存开销.

1 ARace

为了动态容忍和检测非对称数据竞争,ARace 使用两项关键技术:共享变量保护和写缓冲区.

共享变量保护用于保护那些在临界区内只读和先读后写的共享变量.线程在临界区内读这些共享变量之前,ARace 对它们进行保护,以禁止临界区外的线程修改它们.在线程退出临界区时,ARace 再对它们进行解保护,之后临界区外的线程才可以对它们进行修改.

线程在临界区内对共享变量的写被缓存在写缓冲区中.当线程退出临界区时,ARace 会将写缓冲区中的内容写回到原始的共享变量中.通过这种方式,ARace 就可以隐藏在临界区内产生的共享变量中间状态,从而使临界区外的线程仅能看到这些共享变量的最终状态.

图 3 描述了 ARace 的主要工作步骤,圆圈中的数字表示各步骤发生的先后顺序.在这个例子中,X 和 Y 是在临界区内被访问的共享变量.S1 表明这是一个临界区;当 S2 被执行时,①ARace 对 X 进行保护,②X 可以被读取;③当 S3 被执行时,ARace 在写缓冲区中分配一个新的写缓冲区项 Y'(见 2.2 节),用于缓存对共享变量 Y 的写;④当 S4 被执行时,ARace 将写缓冲区项 Y' 写回到原始变量 Y 中,⑤解保护 X.

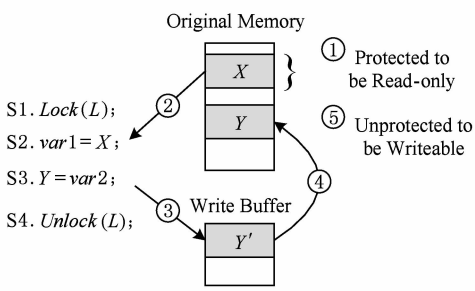


Fig. 3 System overview of ARace.

图 3 ARace 的主要工作步骤

1.1 共享变量保护

为了阻止临界区外的线程修改临界区内只读和先读后写的共享变量,ARace 将这些共享变量保护为只读状态.当线程退出临界区时,这些被保护的共享变量会被解保护为可写状态.对于嵌套临界区,ARace 在线程退出最外层临界区时才进行共享变量的解保护.在大部分现代操作系统中,内存的保护是以内存页为粒度进行的.因此,当 ARace 需要保护某个共享变量 v 时,就需要保护 v 所在的整个内存页.如果 v 占据多个内存页,例如 v 在两个内存页的边界处,ARace 就需要把所有这些内存页都保护起来.

假设两个线程 T_1 和 T_2 并发执行不同锁保护的临界区,并且它们要读的共享变量 v_1 和 v_2 在同一个页 p 中.如果 T_1 和 T_2 分别对 p 进行页保护,那么 p 将会被重复保护.更重要的是,如果 T_1 在 T_2 之前退出临界区,那么在 T_1 退出临界区并对 p 进行解保护之后,T_2 将面临触发非对称数据竞争的风险.

为了解决上述问题,ARace 将那些已经被保护为只读状态的页面,记录在一个全局共享的结构 *globalPage* 中,其中每个页面均有一个线程链表 L,记录那些在临界区内读过这个页面中共享变量的线程.此外,ARace 中的每个线程,均会将在临界区内读过的,并且含有共享变量的页面,记录在私有存储 S 中.算法 1 和算法 2 分别描述了共享变量保护和解保护的基本过程.

算法 1. 共享变量保护 *protect_sv*(t,v,*size*).

输入:thread t,shared variable v read by t,and the size of v.

① P=*pages*(v,*size*);
② for each p in P do
③ *Lock*(*globalPage*.*Lock*);
④ if(p is not in *globalPage*) then
⑤ protect p;
⑥ add p to *globalPage*;
⑦ end if
⑧ add t to $p.L$;
⑨ *Unlock*(*globalPage*.*Lock*);
⑩ add p to $t.S$;
⑪ end for

算法 2. 解保护 *unprotect_sv*(t).

输入:thread t to exit acritical section.

① for each p in $t.S$ do
② *Lock*(*globalPage*.*Lock*);
③ delete t from $p.L$;
④ if $p.L$ is empty then
⑤ unprotect p;
⑥ delete p from *globalPage*;
⑦ end if
⑧ *Unlock*(*globalPage*.*Lock*);
⑨ delete p from $t.S$;
⑩ end for

如果某个临界区被频繁地执行,那么对应的需要保护的内存页 p 就会被频繁地保护和解保护.实际上,如果临界区外的线程没有修改 p 中的内容,那么就没必要在临界区结束时解保护 p,这样就可以减少在进入和退出临界区时,对 p 进行保护和解保护的次数.这种解保护内存页的机制称为延迟解保护(lazy unprotecting, LU).ARace-LU 推迟 p 的解保护直到有临界区外的线程对 p 中的内容进行修改.虽然 ARace-LU 能够减少保护和解保护的次数,但是它也可能会引入额外的保护异常.第 3 节实验数据会对 ARace 和 ARace-LU 进行比较.

1.2 写缓冲区

写缓冲区是一块可读可写的存储区,在 ARace 启动时分配,退出时释放,其结构如图 4 所示:

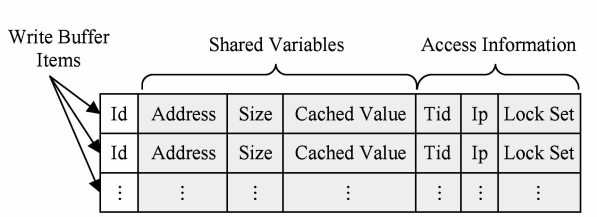


Fig. 4 Structure of write buffer.

图 4 写缓冲区的结构

写缓冲区由写缓冲区项组成,每个写缓冲区项对应一个共享变量,并且使用共享变量的地址进行索引访问.每个写缓冲区项的大小是不固定的,这取决于存放的共享变量的大小.当临界区内的指令第一次写共享变量 v 时,ARace 就在写缓冲区中为其分配一个写缓冲区项 v',并且 v'的大小和指令对 v 的访问大小是一致的.

为了检测并发临界区之间的非对称数据竞争,ARace 在每个写缓冲区项上记录线程对该写缓冲区项最近一次访问的信息(access information):线程号、指令地址和线程持有的锁集合.当线程访问某个写缓冲区项时,ARace 首先比较该线程的线程号和写缓冲区项中记录的线程号是否相同.如果相同,线程直接对写缓冲区项进行访问,并更新写缓冲区项中的访问信息;如果不同,ARace 就进一步比较该线程当前持有的锁集合和写缓冲区项中记录的锁集合是否有交集.如果没有交集,ARace 就认为这两次访问之间存在非对称数据竞争,并将这两次访问的详细信息作为检测结果进行报告.这里 ARace 并没有检查共享变量的访问类型,虽然这可能会引入一些误报,但是文献[5]经过调查发现,这类并发临界区之间的非对称数据竞争在实际程序中非常少见,因此程序员可以根据 ARace 的报告很容易过滤这些误报.

当线程退出临界区时,ARace 将写缓冲区中该线程分配的写缓冲区项写回到对应的原始共享变量中.对于嵌套临界区的情况,ARace 只在线程退出最外层临界区时才进行写回操作.这是因为如果在线程退出内层临界区时就进行写回,可能会漏掉对一些非对称数据竞争的检测.图 5 给出了一个这样的实例,其中线程 T_1 在 S3 和 S5 处访问共享变量 *ptr*,线程 T_2 在 S8 处访问 *ptr*.如果 ARace 在 T_1 退出内层

临界区时就写回 ptr 对应的写缓冲区项,那么就无法检测会导致程序出错的非对称数据竞争(图 5 中箭头所示).

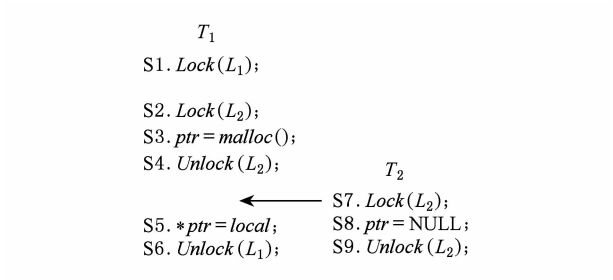


Fig. 5 Asymmetric race in nested critical section.

图 5 嵌套临界区引入的非对称数据竞争

ARace 在进行写回操作时,需要保证写回过程的原子性.这是因为不原子的写回会导致执行结果的不一致性.图 6 给出了不原子写回的实例.线程 T_1 在退出临界区时,将写缓冲区中共享变量 X 和 Y 的新值分别写回到 X 和 Y 中,同时线程 T_2 分别读 X 和 Y 的值到局部变量 $local_1$ 和 $local_2$.按照图 6 中箭头所示的执行交叠,$local_1$ 和 $local_2$ 的最终结果将分别是 1 和 0,这和程序员期望的顺序一致性是冲突的.为了保证写回过程的原子性,ARace 在进行写回之前,将要写回的共享变量保护为不可读不可写.在这个例子中,就是在写回之前将 X 和 Y 保护为不可读不可写.

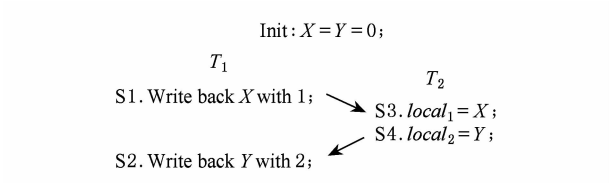


Fig. 6 An example of writing back.

图 6 不原子写回实例

经过上述保护之后,ARace 也无法直接进行写回操作.但是,大部分现代操作系统,例如 UNIX, Linux 以及 Windows 等,均支持将单个物理页映射到进程地址空间中的多个虚拟页[7].为了写回共享变量 v 对应的写缓冲区项 v',ARace 首先分配一个新的虚拟页,然后将这个新的虚拟页映射到 v 所在的原始虚拟页对应的物理页.这样,ARace 就可以根据 v 在原始虚拟页中的偏移,将 v' 写回到这个新的虚拟页.实际上,ARace 可以以页为单位进行写回操作,这比以写缓冲区项(或共享变量)为单位进行写回更加高效.在写回过程结束后,ARace 释放分配的虚拟页,并解保护那些被保护为不可读不可写的共享变量.

1.3 共享变量访问重定向

ARace 对临界区内的每条指令进行检查,判断其是否访问共享变量.如果该指令访问共享变量,ARace 就对其进行访问重定向;否则不进行任何操作.访问重定向如算法 3 所示.

算法 3. 访问重定向 $redirect\_access(ins,t)$.

输入: instruction ins in a critical section, thread t executing ins;

输出: if ins accesses shared variable, memory address after redirecting.

① $type=instruction\_type(ins)$;
② if(($type$ is Read_SV) or ($type$ is Write_SV)) then
③ $addr=shared\_variable\_address(ins)$;
④ $size=shared\_variable\_size(ins)$;
⑤ if ($addr$ is in write buffer) then
⑥ if ($size>writebuffer(addr).size$) then
⑦ allocate a new item in write buffer for ($addr,size$);
⑧ if ($type$ is Read_SV) then
⑨ $protect\_sv(t,addr,size)$;
⑩ copy the contents from $addr$ to new item;
⑪ end if
⑫ copy the contents from old item to new item and free old item;
⑬ end if
⑭ detect asymmetric race and update access information;
⑮ return $\&writebuffer(addr)$;
⑯ end if
⑰ if (($type$ is Read_SV) and ($type$ is not Write_SV)) then
⑱ $protect\_sv(t,addr,size)$;
⑲ return $addr$;
⑳ end if
㉑ if (($type$ is not Read_SV) and ($type$ is Write_SV)) then
㉒ allocate a new item in write buffer for ($addr,size$);
㉓ record access information;
㉔ return $\&writebuffer(addr)$;
㉕ end if
㉖ if (($type$ is Read_SV) and ($type$ is Write_SV)) then

㉗ *protect_sv*(*t*,*addr*,*size*);
㉘ allocate a new item in write buffer for (*addr*,*size*);
㉙ copy the contents from *addr* to new item;
㉚ record access information;
㉛ return &*writebuffer*(*addr*);
㉜ end if
㉝ end if

算法 3 根据指令对共享变量访问的不同类型,进行不同的重定向. 对于大部分精简指令集计算机(reduced instruction set computer, RISC)体系结构,如 *MIPS*,*ALPHA* 和 *PowerPC* 等,指令仅有两种访存方式,即读内存和写内存. 但是对于复杂指令集计算机(complex instruction set computer, CISC)体系结构,例如 *IA*-32,指令可以有 3 种访存方式:读内存、写内存和读写内存,其中第 3 种访存方式允许指令先读内存然后再写内存. *ARace* 中的访问重定向算法根据指令对共享变量的访问类型分别进行处理,因此 *ARace* 支持上述各种访存类型.

1.4 锁变量映射

多线程之间的锁同步是通过锁变量来实现的. 在当前流行的大部分编程语言中,包括 C/C++, Java 和 C# 等,程序员可以像定义普通变量那样定义锁变量. 从编译器的角度来看,锁变量和普通变量并没有区别. 因此,锁变量和共享变量可以被分配在同一个内存页中. 当临界区内的线程因保护共享变量而保护某个内存页时,也会把该内存页中的锁变量同时保护为只读状态. 图 7 描述了一个这样的实例.

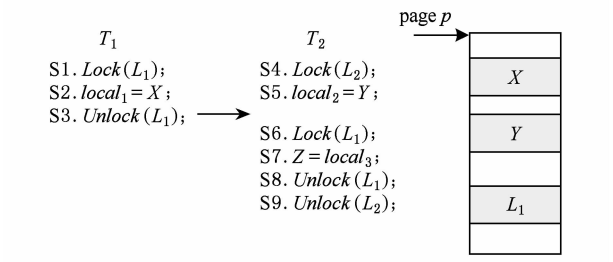


Fig. 7 A deadlock example.

图 7 锁变量和共享变量在同一个页中导致的死锁

在此例中,共享变量 X,Y 和锁变量 L_1 被分配在同一个内存页 p 中. 考虑线程 T_1 和线程 T_2 的如下执行交叠:T_1 和 T_2 并发执行临界区,T_2 在 S6 处等待 T_1 释放锁 L_1. T_1 在退出临界区时(执行 S3 之前)尝试解保护 p 为可写. 然而,由于 T_2 仍然在临界区内执行,在删除 T_1 后,p 的线程链表 L 仍然非空,因此 p 并没有被真正地解保护为可写状态. 这导致 T_1 无法成功释放 L_1,因为释放锁需要对锁变量进行写操作. T_1 和 T_2 相互等待,程序进入死锁状态.

为了避免上述死锁状态,ARace 使用一个锁变量映射表(lock variable mapping table, LVMT)将锁变量 L 映射到锁变量 L',其中 L' 和 L 有相同的大小,并且 L' 在一块独立的可读可写的内存区域中. 如图 8 所示,LVMT 是个一一映射的映射表,它的每一项记录着 L 及其对应的 L' 的内存地址. 当线程通过 *Lock*/*Unlock* 对 L 进行访问时,ARace 就用 L 的内存地址查找 LVMT 以获得 L' 的地址,然后将 L 替换为 L'. 这样就消除了上述死锁的可能性.

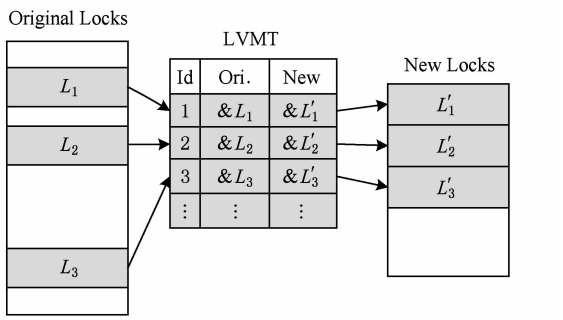


Fig. 8 Lock variable mapping table.

图 8 锁变量映射表

1.5 自定义同步

在一些多线程程序中,程序员为了满足性能的需求,可能会使用自定义同步[8]. 如果自定义同步中的一端在临界区内,那么自定义同步本身就构成了非对称数据竞争. 如图 9 所示,自定义同步 S2 和 S4 就构成了非对称数据竞争.

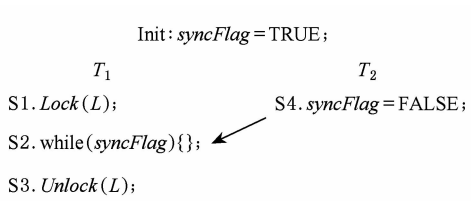


Fig. 9 An asymmetric race with ad hoc synchronization.

图 9 一个含有非对称数据竞争的自定义同步

虽然在 ARace 下执行,不会触发这个非对称数据竞争,但是,如果线程 T_1 执行 S2 在线程 T_2 执行 S4 之前,那么 T_1 将无法退出 S2 处的 while 循环,这是因为 *syncFlag* 被保护成只读状态,T_2 无法对它进行修改. 实际上,像 *syncFlag* 这样的同步变量仅仅是用来实现多线程之间同步的[8],因此没必要在临界区内保护这些共享变量. ARace 使用文献

[8-10]提出的方法来识别 *syncFlag* 这样的同步变量,并且不对它们进行保护.这样临界区内的指令就可以读到同步变量的最新值,从而不会进入无限等待状态.

2 实 现

本节介绍 ARace 的一种实现方式:基于动态二进制插桩技术.这里选择使用 Pin[11]来实现 ARace. Pin 是 Intel 开发的动态二进制插桩框架,并被广泛地应用于动态程序分析的研究中. Pin 通过对程序的二进制代码进行动态即时插桩,来实现插桩工具需求的功能,因此不需要程序的源代码和重新编译.

ARace 的实现是一个 Pintool,它主要包含两个部分:插桩引擎和分析引擎.插桩引擎用来对指令和函数进行插桩,分析引擎包含写缓冲区、共享变量保护、访问重定向以及锁变量映射等.图 10 描述了该实现的基本框架,其中目标多线程程序在 IA-32 平台上使用 pthreads 库进行编译,pthreads 库是一个被广泛使用的多线程库.

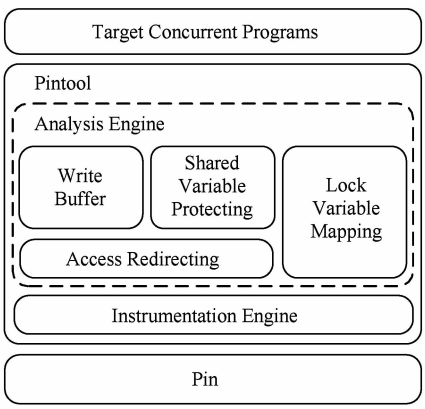


Fig. 10 Implementation framework of ARace.

图 10 ARace 实现的基本框架

2.1 共享变量

从二进制代码上很难判断一个变量是否是共享变量,因此在实现过程中,采取了如下策略:临界区内对全局数据区和动态堆数据区中变量访问的指令,均被认为是访问共享变量.虽然该策略可能会引入一些误判,但它并不会影响 ARace 的正确性.

2.2 临界区和锁变量

在 pthreads 库中,程序分别通过调用库函数 *pthread_mutex_lock* 和 *pthread_mutex_unlock* 进入和退出临界区.对于库函数 *pthread_mutex_trylock*,如果调用线程成功获取相应的锁,那么调用线程也被认为是进入临界区执行.

锁变量就是那些传递给上述库函数、并且具有 *pthread_mutex_t* 结构的参数.在调用上述库函数时,原始的锁变量被新的锁变量替换,这些新的锁变量通过使用原始锁变量的地址查找 LVMT 获得.因此在上述库函数中真正访问的是新的锁变量而不是原始的锁变量.

此外,ARace 的实现还可以使用文献[9-10]提出的方法来识别由程序员自定义的获取锁和释放锁的函数,进而识别自定义的临界区和锁变量.

2.3 条件变量

除了锁变量之外,条件变量是另外一类多线程之间常见的同步方式.一般来说,程序往往在临界区内对条件变量进行访问.图 11 给出了 SPLASH-2[12]的 radix 程序中使用条件变量进行同步的实例.在这个例子中,对条件变量 C 的访问被锁 L 保护.这样就产生了一种误解,即相同锁保护的临界区可以被不同线程并发执行.

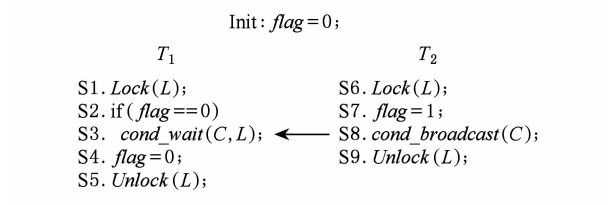


Fig. 11 An example of conditional variable.

图 11 使用条件变量进行同步

实际上,这种误解是错误的,因为条件变量等待操作 *cond_wait*(C,L)的内部实现是:

Unlock(L);

Wait on C;

Lock(L);

所以,只需将条件变量等待操作中的 *Unlock*|*Lock* 操作当作临界区的退出和进入标志即可.在 *pthreads* 库中,它们分别对应库函数 *__pthread_mutex_unlock_usercnt*|*__pthread_mutex_cond_lock*.

2.4 临界区插桩

在临界区内对共享变量访问的指令,通过动态插桩重写它们的内存操作数,以实现对共享变量访问的重定向. IA-32 平台中的一些指令,如 MOVS 系列指令和 CMPS 系列指令等,可以有多个内存操作数.对于这些指令,所有的内存操作数均需要被重写.所谓内存操作数的重写,就是将该内存操作数原始的寻址模式,修改为使用 Pin 中临时寄存器进行寄存器寻址的模式.在指令每次执行时,每个内存操

作数均被插入一个函数来获取其重定向后的地址,并将该地址作为返回值填入 Pin 中的临时寄存器,这样就可以实现对指令内存操作数的重写.

2.5 临界区内的函数调用

在临界区内被调用的函数也需要进行插桩,以实现对共享变量访问的重定向.但是,在临界区外被调用的函数就没有必要进行插桩.实际上,同一个函数可能既在临界区内被调用,又在临界区外被调用.如果一个函数第 1 次被调用是在临界区外,即使之后可能会在临界区内被调用,那么它也不会被插桩.这是因为 Pin 中用于插桩的函数仅在被插桩函数第一次被调用时才执行.

为了克服上述插桩的局限性,在实现的过程中,需要为函数插桩定义一个规则:一旦函数在临界区内被调用,它就会一直被插桩,即使之后它可能会在临界区外被调用;否则它永远不会被插桩.在对共享变量访问进行重定向之前,需要判断当前线程是否是在临界区内执行,如果不是,就可以直接返回,不进行重定向;如果是,就继续进行重定向.这样对于那些先在临界区内被调用的函数,虽然它已经被插桩,但是之后在临界区外被调用时,实际上并不会进行共享变量访问的重定向.

每个函数 r 均记录一个布尔值 Fr.在 r 第 1 次被调用时,根据 r 是否是在临界区内被调用对 Fr 进行初始化.如果 r 第 1 次被调用是在临界区内,Fr 就被初始化为 TRUE,否则,Fr 就被初始化为 FALSE.在临界区内执行的每条 call 指令均需要被检查.对于直接 call 指令,其目标函数 r 是固定的,并且在插桩时是可知的.因此只需检查 r 的 Fr 值.如果 Fr 是 FALSE,那么首先需要失效(invalidate)掉 Code Cache 中与 r 对应的未插桩的代码,然后重新执行 Pin 中用于插桩的函数,以实现对 r 的插桩.最后再将 Fr 设置为 TRUE,这表示 r 已经在临界区内被调用.对于间接 call 指令,目标函数 r 是不固定的,因此需要在该指令每次执行时均插入一个函数来获取 r 及其对应的 Fr 值,并按照上述过程对 Fr 的值进行检查.

2.6 系统调用

在临界区内执行的系统调用也有可能访问共享变量,例如:

```
Lock(L);
⋮
gettimeofday(&tv,NULL);
⋮
Unlock(L);
```

其中,tv 是在用户空间定义并在内核空间被修改的共享变量.然而,ARace 不应该将 tv 的地址直接传递给内核.这是因为 tv 所在的内存页有可能已经被 ARace 保护为只读状态.如果直接将 tv 的地址传递给内核,那么在内核将系统调用的结果写到 tv 时就会发生错误.这种错误在程序直接运行时一般是不会发生的.除了在临界区内执行的系统调用外,在临界区外执行的系统调用也有同样的问题.

为了避免这种不期望的系统调用错误,在实现 ARace 时,需要对那些访问用户空间定义的共享变量的系统调用进行包装.系统调用包装就是分配一个新变量,并将该新变量的地址作为系统调用的参数传递给内核.如果系统调用是在临界区内执行并且该变量是共享变量,那么就在写缓冲区中分配新变量,并在线程退出临界区时写回到原始变量;否则可以直接使用一个临时变量作为新变量,并在系统调用执行完后立即将该新变量写回到原始变量.

3 实验数据

3.1 实验环境

表 1 给出了具体的实验环境,包括实验平台以及测试程序.实验平台拥有 Intel® Core™ 2 Duo T7250 2.00 GHz 的处理器,2 MB 的二级缓存和 1 GB 的主存;操作系统是 32 位 Fedora 14,内核版本是 2.6.35;编译器是 GCC-4.5.1.

Table 1 Experimental Environment

表 1 实验环境

Platform and Benchmarks	Arguments
Processor	Intel® Core™ 2 Duo T7250 2.00 GHz
L2 Cache/MB	2
Memory/GB	1
OS	32 b Fedora 14, Linux-2.6.35
Compiler	GCC-4.5.1
SPLASH-2	default input, just incease input size when necessary
Phoenix	MapReduce version, large input size
Real Apps	pbzip2, compress a 73 MB tar file
	aget, download a 321 MB file from a local http server
	pfscan, find a string in a directory containg 227 MB files

测试程序包括 SPLASH-2[12] 和 Phoenix[13] 中

的程序,以及实际的多线程应用程序. Phoenix 中的每个程序有 MapReduce, Pthreads 和 Sequential 3 个版本,其中 MapReduce 和 Pthreads 版本均使用 pthreads 库进行编译,这里选择 MapReduce 版本进行测试. 实际的多线程应用程序包括: pbzip2[14], aget[15] 和 pfscan[16]. pbzip2 是文件压缩器 bzip2 的并行实现;aget 是一个多线程的 http 下载器;pfscan 是并行的文件扫描器,可以实现文件的 find, grep 等功能.

测试程序的性能通过使用命令"time-p"获得. 为了消除系统中一些随机因素带来的性能干扰,测试程序均运行多遍,最终性能取多遍的平均值.

3.2 临界区特性

表 2 给出了测试程序的动态临界区特性. 其中第 3 列和第 4 列分别是活动的锁变量和总共的锁变量数目,它们分别表示程序使用的锁变量和申请的锁变量,这两列的数据表明程序中存在申请但未使用的锁变量;第 5 列是动态执行的临界区的数目,可以看出,一些测试程序执行了大量的临界区代码,例如 radiosity 和 barnes;第 6 列是平均在每个临界区内执行的指令数;后面 3 列分别是平均在每个临界区内执行的读、写以及读写共享变量的指令数;最后一列给出了在临界区内执行的指令数占总体指令数的百分比.

Table 2 Critical Section Characterization

表 2 临界区动态特性

Suites	Benchmarks	# Lock Active	# Lock Total	# CS Executed	# Inst per CS	# Read SV per CS	# Write SV per CS	# ReadWrite SV per CS	Inst in CS/%
SPLASH-2	cholesky	7	7	91	112.51	7.64	2.7	0	0.00
	fft	1	1	2	553.5	9.5	1.5	0	0.00
	lu-con	1	1	2	553.5	9.5	1.5	0	0.00
	lu-non	1	1	2	549.5	6.5	1.5	0	0.00
	radix	4	6	13	303.77	58.69	2.54	1.54	0.00
	barnes	2 049	2 050	686 646	265.68	15.54	15.57	0	0.33
	fmm	2 051	2 052	330 980	481.32	21.46	24.49	0.000 012	0.21
	ocean-con	2	6	2 416	16.13	4.91	0.91	0	0.00
	ocean-non	3	6	89 044	15.33	4.77	0.77	0	0.00
	radiosity	3 914	3 915	3 212 879	21.06	6.29	2.43	0	0.24
	raytrace	5	5	196 133	21.94	3.45	1.16	0	0.00
	volrend	5	67	70 766	25.2	4	1	0	0.02
	water-nsquared	517	521	4 130	277.8	58.49	8.93	0	0.06
	water-spatial	70	70	2 035	55.42	9.38	1.49	0	0.01
Phoenix	histogram	2	4	21 718	61.51	8.95	2.98	0	0.02
	kmeans	2	4	341 715	129.68	13.17	5.21	2.43	0.00
	linear_regression	2	4	8 538	60.76	8.88	2.94	0	0.00
	matrix_multiply	2	4	369	83.14	7.43	3.4	0.90	0.00
	pca	2	4	7 432	3 349.73	192.74	475.16	19.12	0.08
	reverse_index	2	4	6 790	149.88	21.35	13.5	0	0.01
	string_match	2	4	8 537	243.12	12.76	3.91	2.91	0.00
	word_count	4	7	2 143	93.35	6.53	1.76	0	0.00
Real Apps	pbzip2	3	3	680	132.22	16.07	3.33	1.26	0.00
	aget	1	1	82 947	7.01	2.00	1.00	0.000 024	0.59
	pfscan	4	4	14 977	21.71	5.70	1.67	1.00	0.00

3.3 性能开销

图 12 给出了测试程序归一化的执行时间. 由于 ARace 的实现并不依赖于 Pin(可以使用别的方法或工具),因此这里以 Pin 的执行时间为基础,对各

运行时间进行归一化.每个测试程序共有4根柱子,其中第1根是程序原始的运行时间,第2根是Pin的运行时间,第3根和第4根分别是ARace和ARace-LU的运行时间.从图12中可以看出,除了radiosity,ARace引入的性能开销均在3倍以内,而对于radiosity,ARace引入的性能开销最大,在17.2倍左右.平均来看,相对于Pin,ARace引入的性能开销仅有49%左右.这说明ARace在动态容忍和检测非对称数据竞争的同时,引入的性能开销并不大.

图13给出了与ARace相比,ARace-LU减少的进入和退出临界区时保护与解保护内存页面的次数,以及增加的页面异常次数.从图13中可以看出,虽然ARace-LU减少了进入和退出临界区时,对页面进行保护与解保护的次数,但同时也增加了页面异常的次数.这和前面2.1节对ARace-LU行为的分析结果是相吻合的.

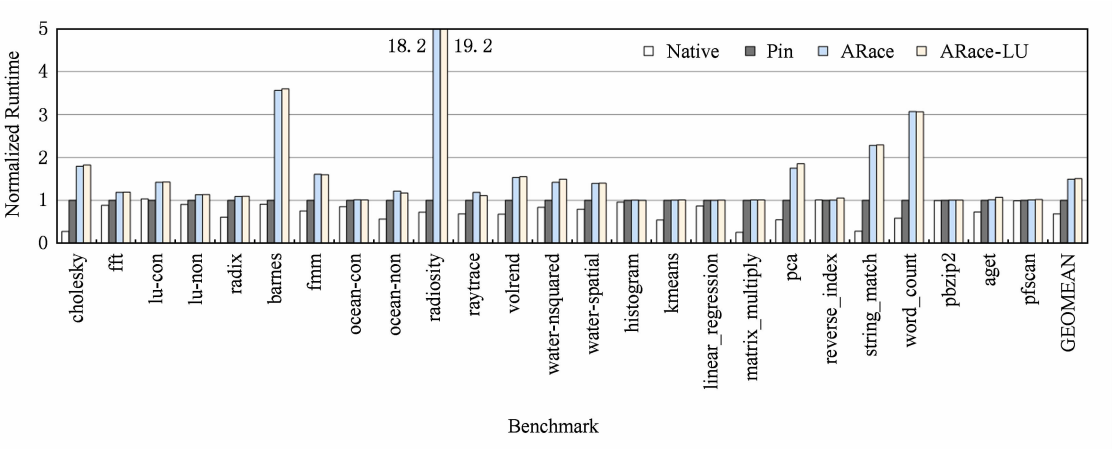


Fig. 12 Normalized run time.

图12 归一化的运行时间

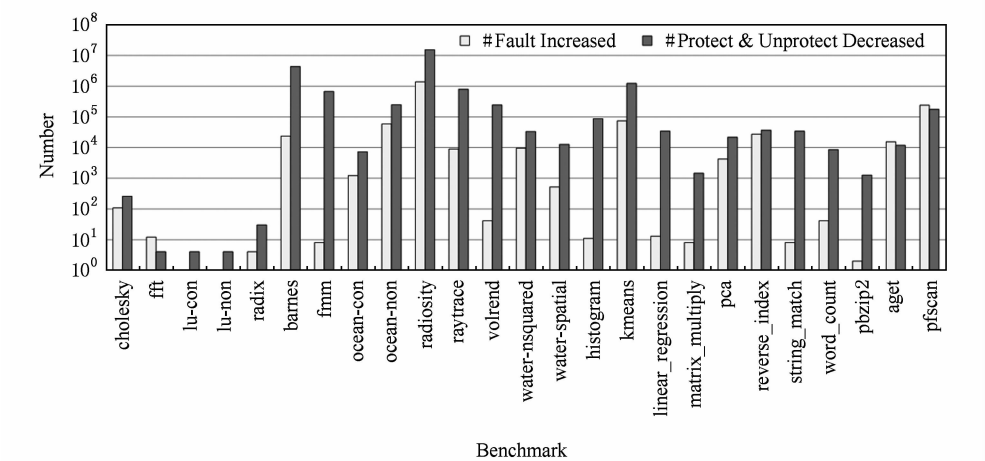


Fig. 13 Execution characterization of ARace-LU.

图13 ARace-LU运行特点

表3给出了ARace的一些运行时特性.第3列显示了ARace引入的总的页面异常次数,可以看出,对于大部分程序,ARace引入的异常次数并不多;第4列和第5列分别给出了在静态数据和动态堆上的页面异常次数;第6列为ARace在执行过程中,失效Code Cache中未插桩函数的次数;第7列给出了ARace进行页面保护和解保护的次数;最后一列给出了ARace在临界区结束时写回的页面总数.对大部分程序来说,ARace需要写回的页面都比较多,这是因为所有在临界区内对共享变量的写都被缓存在写缓冲区中.

为了进一步分析ARace各部分引入的性能开销,图14还收集了ARace各部分运行时间的相对比例.其中,初始化部分(initialization)是在执行测试程序前完成的,用于对Pin进行初始化;页面异常处理部分(page fault handler)是在线程访问被保护页

Table 3 Execution Statistics of ARace

表 3 ARace 的运行时特性

Suites	Benchmarks	# Fault	# Fault Static	# Fault Dynamic	# Invalid	# Protect & Unprotect	# Page Written Back
SPLASH-2	cholesky	1	0	1	44	442	36
	fft	1	0	1	14	8	3
	lu-con	2	0	2	14	8	3
	lu-non	1	1	0	14	8	3
	radix	0	0	0	24	40	15
	barnes	2	1	1	22	4 376 954	1 956 805
	fmm	0	0	0	62	661 096	10
	ocean-con	0	0	0	14	9 664	1 745
	ocean-non	0	0	0	14	356 176	64 119
	radiosity	103 639	0	103 639	32	17 740 892	4 464 888
	raytrace	11	1	10	14	802 450	203 010
	volrend	9	1	8	42	240 564	70 751
	water-nsquared	1	1	0	14	56 470	4 195
	water-spatial	2	1	1	60	14 156	3 060
Phoenix	histogram	1	0	1	15	86 872	43 172
	kmeans	22 225	20 943	1 282	42	1 366 860	539 550
	linear_regression	0	0	0	15	34 152	16 812
	matrix_multiply	0	0	0	50	1 476	474
	pca	111	0	111	46	29 728	14 336
	reverse_index	33 413	0	33 413	15	105 904	13 316
	string_match	0	0	0	43	34 148	16 810
	word_count	6	5	1	16	8 572	3 758
Real Apps	pbzip2	28	28	0	66	1 266	4
	aget	37 428	37 428	0	14	82 078	41 309
	pfscan	83 461	64 511	18 950	24	363 788	164 845

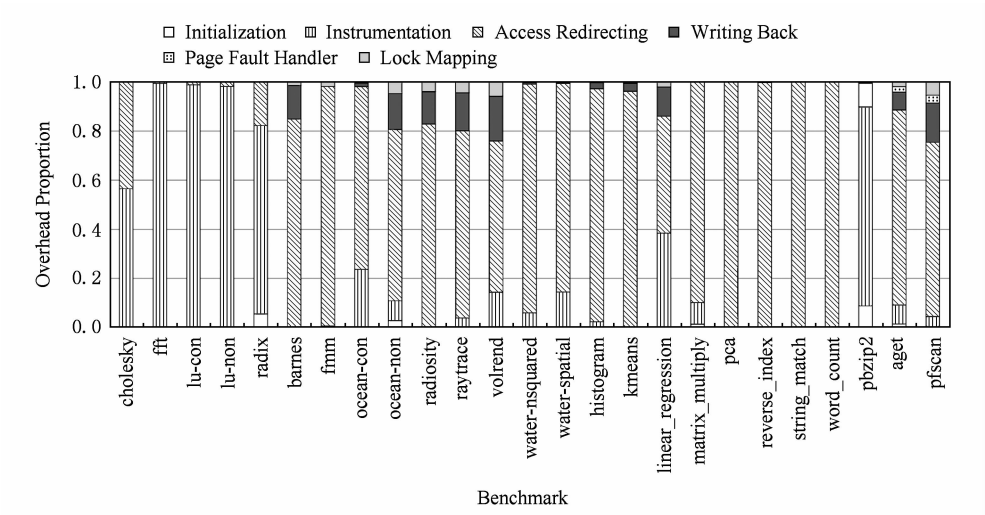


Fig. 14 Overhead proportion of ARace.

图 14 ARace 各部分性能开销的相对比例

并发生页面异常时执行的.

从图 14 中可以看出,除了 cholesky 和 linear_regression,剩余的程序基本可以分为两大类.在第 1 类中,性能开销的主要部分是插桩(instrumentation),例如 fft,lu-con,lu-non,radix 和 pbzip2 等;在另外一类中,性能开销的主要部分是访问重定向(access redirecting),例如 barnes,radiosity 和 string_match 等.这种区别的主要原因在于,第 2 类程序在临界区内动态执行的指令数占总指令数的比例远远大于第 1 类程序.对于 cholesky 和 linear_regression,它们的比例在第 1 类和第 2 类之间,因此插桩和访问重定向的时间比例比较接近.从图 14 中还可以看出,对于一些程序,写回(writing back)也是性能开销的重要部分,例如 radiosity,volrend 和 pfscan 等,这是因为临界区内有大量对共享变量的写操作.此外,从图 14 中也可以看出初始化、页面异常处理和锁变量映射(lock mapping)引入的性能开销并不大.

3.4 内存开销

表 4 给出了 ARace 在运行过程中的内存占用量,其中第 3 列为写缓冲区占用的内存;第 4 列是 globalPage 结构占用的内存;第 5 列为线程私有存储 S 占用的内存;第 6 列为锁变量映射占用的内存;最后一列为总共占用的内存,为前面 4 列之和.从表 4 中可以看出,ARace 占用的内存一般都很小,其中最大的 barnes 也只有 169 KB.

图 15 给出了 ARace 的内存开销占程序原始所需内存总量的百分比.从图 15 中可以看出,ARace 引入的内存开销平均在 0.1%以下.

Table 4 Memory Overhead of ARace

表 4 ARace 的内存占用量 B

Suites	Benchmarks	Write Buffer	Page Info	Thread Page Info	Lock	Total
SPLASH-2	cholesky	528	112	56	168	864
	fft	88	16	16	24	144
	lu-con	88	16	16	24	144
	lu-non	88	16	16	24	144
	radix	88	16	16	96	216
	barnes	123 080	448	136	49 176	172 840
	fmm	176	48	24	49 224	49 472
	ocean-con	88	16	16	48	168
	ocean-non	88	16	16	72	192
	radiosity	22 836	256	104	93 936	117 132
	raytrace	208	144	64	120	536
	volrend	792	32	16	120	960
	water-nsquared	18 576	240	72	12 408	31 296
	water-spatial	144	96	32	1 680	1 952
Phoenix	histogram	176	16	16	48	256
	kmeans	220	16	16	48	300
	linear_regression	176	16	16	48	256
	matrix_multiply	132	16	16	48	212
	pca	176	16	16	48	256
	reverse_index	176	112	112	48	448
	string_match	132	16	16	48	212
	word_count	132	16	16	96	260
Real Apps	pbzip2	44	16	8	72	140
	aget	88	8	8	24	128
	pfscan	176	16	16	96	304

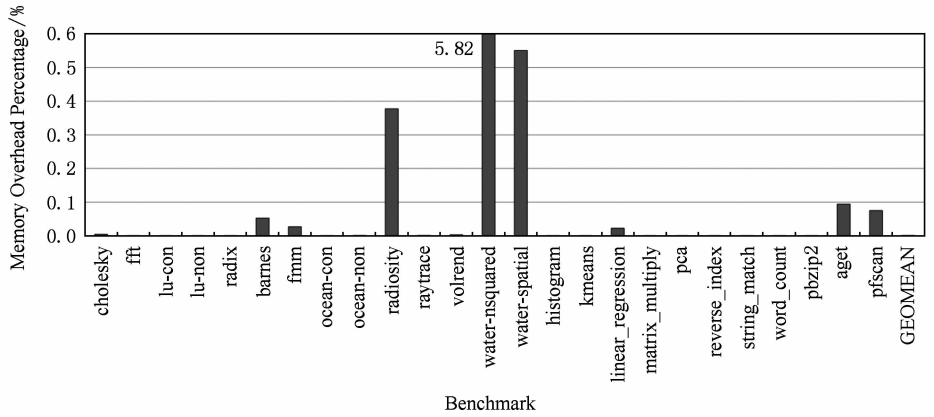


Fig. 15 Memory overhead of ARace.

图 15 ARace 的内存开销

3.5 可扩展性

图 16 给出了 ARace 在测试程序不同线程数的条件下,性能开销和内存开销的变化情况. 由于 aget 程序要求线程数不超过 10,因此这里无法给出线程数 16 和 32 的数据. 对于 aget,随着线程数的增加,ARace 引入的内存开销逐渐下降,这是因为,被测试程序本身所使用的内存总量,随着线程数增加而增加的幅度超过了 ARace.

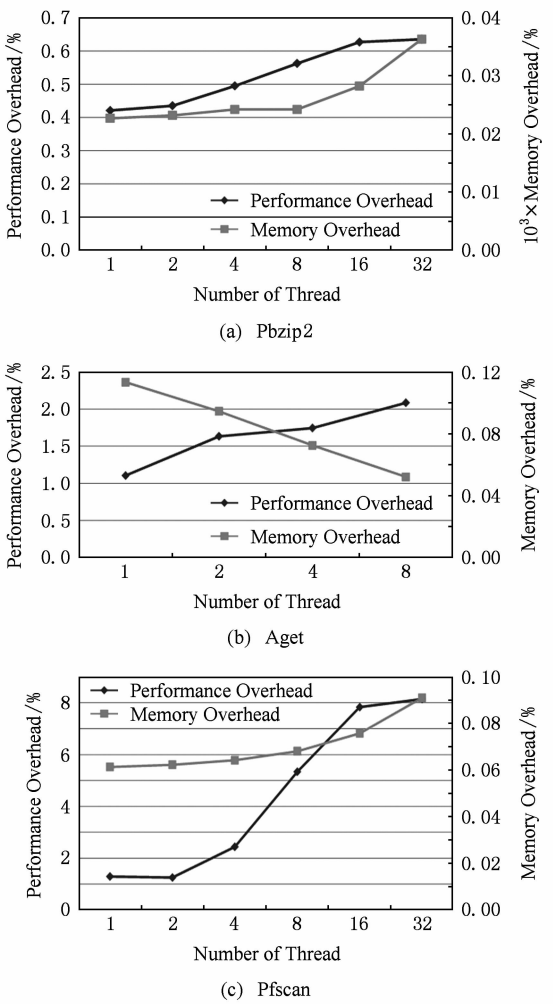


Fig. 16 Scalability of ARace.

图 16 ARace 的可扩展性

由图 16 可以发现,随着线程数的增加,ARace 引入的性能开销和内存开销的增加幅度均不大. 这表明 ARace 具有较好的可扩展性.

4 相关工作

4.1 非对称数据竞争

目前国际上对非对称数据竞争的研究主要是从动态容忍方面入手. 根据这些方法实现机制的不同,大体可以将它们分为两类:基于软件的方法和基于硬件的方法. 其中,基于软件的方法中比较有代表性的是 ToleRace[4,17-18] 和 ISOLATOR[19];基于硬件的方法比较有代表性的是 Pacman[5]. 下面分别详细介绍.

当线程 T_1 进入临界区时,ToleRace 为每个在该临界区内访问的共享变量 v 创建两个备份 v' 和 v'',并复制 v 的值到 v' 和 v'',使得 $v=v'=v''$. T_1 在临界区内对 v 的访问被修改成对 v' 的访问. 与此同时,在临界区外执行的线程 T_2 仍然可以正常访问共享变量 v. 当 T_1 退出临界区时,ToleRace 比较 v 和 v'' 的值. 如果 v 和 v'' 相等,ToleRace 认为没有触发非对称数据竞争;否则 ToleRace 需要根据发生的非对称数据竞争的类型来决定共享变量 v 的最终取值:1)如果 T_1 的执行可以串行化到 T_2 之前,那么就保留 v 的值;2)如果 T_2 的执行可以串行化到 T_1 之前,那么就复制 v' 的值到 v;3)如果 T_1 的执行和

T_2 的执行无法串行化,那么 ToleRace 就终止程序的执行,因为它无法容忍这类非对称数据竞争.实际上,第 3 类非对称数据竞争在实际程序中是很常见的[5],这是因为程序在临界区内对共享变量进行访问时,往往会先读该共享变量,然后再写该共享变量.此外,如果临界区内存在对多个共享变量的访问,那么分析非对称数据竞争的类型会非常复杂,这也大大限制了 ToleRace 的容忍能力.

ISOLATOR 使用影子页来容忍非对称数据竞争.当线程 T_1 进入临界区时,ISOLATOR 复制每个在临界区内访问的共享页到其对应的影子页,并将原始的共享页设置成不可写状态.T_1 在临界区内对共享页的访问被修改为对相应的影子页的访问.与此同时,如果在临界区外执行的线程 T_2 尝试修改相同的共享页,那么 T_2 将会被阻塞直到 T_1 退出临界区.当 T_1 退出临界区时,ISOLATOR 将所有影子页的内容写回到共享页,然后把共享页设置为可写状态,这样被阻塞的 T_2 就可以对共享页进行修改.由于需要对共享页进行重新布局,因此 ISOLATOR 需要对程序的源代码进行重新编译,对于某些程序甚至需要手工修改程序的源代码.这大大限制了 ISOLATOR 的使用范围,这是因为存在非对称数据竞争的程序的源代码往往是无法获得的,例如遗产代码和第三方库代码等.当临界区访问多个共享页时,ISOLATOR 无法保证写回多个影子页到共享页的原子性,这样就会导致不一致的执行结果.此外,当两个并发临界区同时访问相同的共享变量时,ISOLATOR 还存在死锁的可能性.

和上述两个基于软件的方案不同,Pacman 是基于硬件的动态容忍非对称数据竞争的方案.Pacman 通过在处理器 cache 中添加额外的硬件来监控多核处理器之间的 cache 一致性通信,从而控制处理器对共享变量的访问,进而实现动态容忍非对称数据竞争.从硬件角度看,在临界区内执行的指令和在临界区外执行的指令没有区别.因此,Pacman 需要对多线程库(如 pthreads 库等)进行修改,以标示进入或退出临界区的函数调用.虽然 Pacman 通过添加额外的硬件降低了动态容忍非对称数据竞争的性能开销,但是由于目前的商业处理器上没有这类硬件支持,所以 Pacman 仍然无法直接应用在现有的处理器上.因此,开发软件的方法来解决非对称数据竞争引入的问题就很有必要.

4.2 事务内存

事务内存(transactional memory, TM)是另一类与 ARace 相关的研究工作[20].在 TM 中,原子区域被看作是一个事务,并且事务可以投机地执行.在事务结束时,TM 检查事务在执行过程中,是否存在对共享变量的冲突访问.如果存在,TM 将丢弃(abort)该事务的执行并回滚(roll back)重新执行该事务;否则,TM 就提交该事务的执行.事务内存按照其实现方式可以分为基于硬件[21-22]、基于软件[23]、以及基于软硬件混合[24-25].

虽然 TM 也有和 ARace 类似的写缓冲区,但是 TM 主要是提供一种机制,用以保证无锁数据结构的原子性,而 ARace 则是要解决多线程程序中非对称数据竞争的容忍和检测问题.

ARace 的实现机制和 TM 有本质区别.这是因为 ARace 没有投机执行临界区,不需要对临界区进行丢弃和回滚;而 TM 在回滚的过程中需要有效地处理那些有副作用(side effect)的操作,目前这仍然是个公认的难题[4].此外,ARace 使用 LVMT 解决程序执行过程中潜在的死锁问题(见 2.4 节),这也是 TM 中没有遇到的问题.

4.3 数据竞争检测

传统的数据竞争检测就是通过程序分析等方法实现对数据竞争的检测.从使用方法的不同上,数据竞争检测基本可以分为静态检测和动态检测两类.静态检测主要使用程序分析技术,例如基于类型的检查[26]、静态流分析[27]、以及锁集合分析[28]等.由于分析方法的不精确,静态检测一个固有的缺陷就是存在大量的误报(false positive).动态检测主要基于锁集合算法[29]、happens-before 分析[30]、以及二者的混合[31].虽然和静态检测相比,动态检测的误报更少,但是动态检测面临着覆盖率问题.与传统的数据竞争检测相比,ARace 针对的主要是非对称数据竞争,并对这类数据竞争进行动态容忍和检测.

5 结 论

本文提出了一种在基于锁的多线程程序中动态容忍和检测非对称数据竞争的方法 ARace.它通过共享变量保护和写缓冲区两项技术,实现对非对称数据竞争的容忍和检测.与已有的方法相比,ARace 不仅可以容忍临界区内和临界区外之间的非对称数据竞争,还可以对并发临界区之间的非对称数据

竞争进行检测.此外,ARace 不依赖任何程序源码,编译器和硬件的支持.本文还给出了 ARace 的一种基于动态二进制插桩的实现方式.实验数据表明,ARace 在容忍和检测非对称数据竞争的同时,并未引入很大的性能开销和内存开销.

参 考 文 献

[1] Leveson N G, Turner C S. An investigation of the Therac-25 accidents [J]. Computer, 1993, 26(7): 18-41

[2] Security Focus. Software bug contributed to blackout [EB/OL]. [2004-02-12]. http://www.securityfocus.com/news/8032

[3] PC World. Nasdaq's facebook glitch came from race conditions [EB/OL]. [2012-09-21]. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html

[4] Ratanaworabhan P, Burtscher M, Kirovski D, et al. Detecting and tolerating asymmetric races [C] //Proc of PPoPP'09. New York: ACM, 2009: 173-184

[5] Qi Shanxiang, Otsuki N, Nogueira L O, et al. Pacman: Tolerating asymmetric data races with unintrusive hardware [C] //Proc of HPCA'12. Piscataway, NJ: IEEE, 2012: 1-12

[6] Lu S, Park S, Seo E, et al. Learning from mistakes—A comprehensive study on real world concurrency bug characteristics [C] //Proc of ASPLOS'08. New York: ACM, 2008: 329-339

[7] Abadi M, Harris T, Mehrara M. Transactional memory with strong atomicity using off-the-shelf memory protection hardware [C] //Proc of PPoPP'09. New York: ACM, 2009: 185-195

[8] Xiong W, Park S, Zhang J, et al. Ad hoc synchronization considered harmful [C] //Proc of OSDI'10. Berkeley: USENIX, 2010: 1-14

[9] Tian C, Nagarajan V, Gupta R, et al. Dynamic recognition of synchronization operations for improved data race detection [C] //Proc of ISSTA'08. New York: ACM, 2008: 143-153

[10] Jannesari A, Tichy W F. Identifying ad-hoc synchronization for enhanced race detection [C] //Proc of IPDPS'10. Piscataway, NJ: IEEE, 2010: 1-10

[11] Luk C, Cohn R, Muth R, et al. Pin: Building customized program analysis tools with dynamic instrumentation [C] //Proc of PLDI'05. New York: ACM, 2005: 190-200

[12] Woo S, Ohara M, Torrie E, et al. The SPLASH-2 programs: Characterization and methodological considerations [C] //Proc of ISCA'95. New York: ACM, 1995: 24-36

[13] Ranger C, Raghuraman R, Penmestsa A, et al. Evaluating MapReduce for multi-core and multiprocessor systems [C] //Proc of HPCA'07. Piscataway, NJ: IEEE, 2007: 13-24

[14] Pbzip2. Parallel bzip2 [EB/OL]. [2012-09-20]. http://compression.ca/pbzip2

[15] Aget. Multithreaded HTTP download accelerator [EB/OL]. [2013-01-20]. http://www.enderunix.org/aget

[16] Pfscan. Parallel file scanner [EB/OL]. [2013-01-25]. http://ostatic.com/pfscan

[17] Ratanaworabhan P, Kirovski D, Nagpal R, et al. Efficient runtime detection and toleration of asymmetric races [J]. IEEE Trans on Computer, 2012, 61(4): 548-562

[18] Ratanaworabhan P, Burscher M, Kirovshi D, et al. Hardware support for enforcing isolation in lock-based parallel programs [C] //Proc of ICS'12. New York: ACM, 2012: 301-310

[19] Rajamani S, Ramalingam G, Ranganath V P, et al. ISOLATOR: Dynamic ensuring isolation in concurrent programs [C] //Proc of ASPLOS'09. New York: ACM, 2009: 181-192

[20] Herlihy M, Moss J. Transactional memory: Architectural support for lock-free data structures [C] //Proc of ISCA'93. New York: ACM, 1993: 289-300

[21] Lupon M, Magklis G, Gonzalez A. A dynamically adaptable hardware transactional memory [C] //Proc of MICRO'10. Piscataway, NJ: IEEE, 2010: 27-38

[22] Khan B, Horsnell M, Lujan M, et al. Scalable object-aware hardware transactional memory [C] //Proc of Euro-Par'10. Berlin: Springer, 2010: 268-279

[23] Saha B, Adi-Tabatabai A, Jacobson Q. Architectural support for software transactional memory [C] //Proc of MICRO'06. Piscataway, NJ: IEEE, 2006: 185-196

[24] Kumar S, Chu M, Hughes C J, et al. Hybrid transactional memory [C] //Proc of PPoPP'06. New York: ACM, 2006: 209-220

[25] Damron P, Fedorova A, Lev Y, et al. Hybrid transactional memory [C] //Proc of ASPLOS'06. New York: ACM, 2006: 336-346

[26] Flanagan C, Freund S N. Type-based race detection for Java [C] //Proc of PLDI'00. New York: ACM, 2000: 219-232

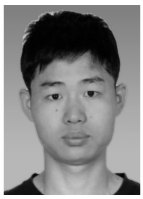
[27] Huo Wei, Yu Hongtao, Feng Xiaobing, et al. Static race detection of interrupt-driven programs [J]. Journal of Computer Research and Development, 2011, 48(12): 2290-2299 (in Chinese)
(霍玮, 于洪涛, 冯晓兵, 等. 静态检测中断驱动程序的数据竞争[J]. 计算机研究与发展, 2011, 48(12): 2290-2299)

[28] Pratikakis P, Foster J S, Hicks M. LOCKSMITH: Context-sensitive correlation analysis for race detection [C] //Proc of PLDI'06. New York: ACM, 2006: 320-331

[29] Savage S, Burrows M, Nelson G, et al. Eraser: A dynamic data race detector for multithreaded programs [J]. ACM Trans on Computer Systems, 1997, 15(4): 391-411

[30] Adve S V, Hill M D, Miller B P, et al. Detecting data races on weak memory systems [C] //Proc of ISCA'91. New York: ACM, 1991: 234-243

[31] O'Callahan R, Choi J. Hybrid dynamic data race detection [C] //Proc of PPoPP'03. New York: ACM, 2003: 167-178



Wang Wenwen, born in 1986. PhD candidate. His main research interests include multicore programming, dynamic optimization, and binary translation.

Wu Chenggang, born in 1969. PhD and associate professor. Senior member of China Computer Federation. His main research interests include dynamic optimization and binary translation (wucg@ict.ac.cn).

Paruj Ratanaworabhan, born in 1970. PhD and lecturer at the Department of Computer Engineering, Kasetsart University in Thailand. His current research focus is on system security and multicore programming (paruj.r@ku.ac.th).

Yuan Xiang, born in 1984. PhD candidate. His main research interests include dynamic optimization and binary translation (yuanxiang@ict.ac.cn).

Wang Zhenjiang, born in 1983. PhD and assistant professor. His main research interests include dynamic optimization and binary translation (wangzhenjiang@ict.ac.cn).

Li Jianjun, born in 1984. PhD. His main research interests include program analysis and dynamic optimization (lijianjun@ict.ac.cn).

Feng Xiaobing, born in 1969. Professor. Member of China Computer Federation. His main research interests include compiler techniques and programming environment (fxb@ict.ac.cn).