TAMING IMPLICIT SYNCHRONIZATIONS IN CONCURRENT PROGRAMS

BY

WEIWEI XIONG

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Yuanyuan Zhou, Chair
Associate Professor Samuel T. King
Associate Professor Darko Marinov
Professor Geoffrey M. Voelker, University of California, San Diego
Abstract

Synchronization takes an important role in multi-threaded programs. Due to the non-deterministic nature of concurrency, it is always difficult for developers to make synchronizations correct. As a consequence, concurrent programs are vulnerable to concurrency bugs and fail during run-time.

Many synchronizations in existing multithreaded programs are implicit. They are either implemented in an ad hoc way, or customized by programmers to be application specific. These implicit synchronizations are difficult to be recognized by programmers or automatic program analysis tools. As a result, they often cause critical reliability problems to concurrent programs, and also make concurrent program analysis tools to be less effective. Unfortunately, this type of synchronizations has significantly been under-studied.

This dissertation makes three major contributions towards studying and improving implicit synchronizations in multi-threaded programs. The first contribution is a comprehensive characteristic study of ad hoc synchronizations. By studying 229 ad hoc synchronizations in 12 programs of various types (server, desktop and scientific), including Apache, MySQL, Mozilla, etc., we find several interesting and perhaps alarming characteristics: (1) Every studied application uses ad hoc synchronizations. Specifically, there are 6-83 ad hoc synchronizations in each program. (2) Ad hoc synchronizations are error-prone. Significant percentages (22-67%) of these ad hoc synchronizations introduced bugs or severe performance issues. (3) Ad hoc synchronization implementations are diverse and many of them cannot be easily recognized as synchronizations, i.e., have poor readability and maintainability.

The second contribution is a tool called SyncFinder to automatically identify and annotate ad hoc synchronizations in concurrent programs written in C/C++ to assist programmers in porting their code to better structured implementations, while also enabling other tools to recognize them as synchronizations. Our evaluation using 25 concurrent programs shows that, on average, SyncFinder can automatically identify 96% of ad hoc synchronizations with 6% false positives. The dissertation also uses two use cases to leverage SyncFinder’s auto-annotation. The first one uses annotation to detect 5 deadlocks (including 2 new ones) and 16 potential issues missed by previous analysis tools in Apache, MySQL and Mozilla. The second use case reduces Valgrind data race checker’s false positive rates by 43–86%.
The third contribution of this dissertation is to remove false positives of state-of-the-art data race detectors by detecting *address transfer*, a major cause of false positives in data race detections. Most data race detectors require the accurate knowledge of synchronizations in the programs and cannot recognize implicit synchronizations, thus suffer from false positives which limit their usability. To address this problem, some tools such as Intel® Inspector provide mechanisms for suppressing false positives and/or annotating synchronizations not automatically recognized by the tools. However, they require users’ input or even changes of the source code. This dissertation approaches this problem in a different way. The dissertation first uses a state-of-the-art commercial data race detector, namely Intel® Inspector on 17 applications of various types including 5 servers, 5 client/desktop applications, and 7 scientific ones, without utilizing any suppression or annotation mechanisms provided by the product that need users’ input. By examining a total of 1420 false data races, the dissertation identifies two major root causes including address transfer, where one thread passes memory address to another thread. It is revealed that more than 62% false data races were caused by address transfer. Based on this observation, this dissertation invents a tool called ATDetector that automatically identifies address transfer and uses the information to prune the false data races. The evaluation with 8 real-world applications shows that it can effectively prune all false data races caused by unrecognized address transfers, without eliminating any true data race that was originally reported.
To my parents, my sisters, and my dear wife
Acknowledgments

I would like to express my deepest gratefulness to my advisor, Dr. Yuanyuan (YY) Zhou, who gives tremendous help to me on research and also many other facets. Without her great visionary guidance, encouragement, and continuous support, the dissertation would not exist. During the whole PhD journey, YY advises and guides me from identifying a concrete and real world problem, to incubating a research idea and establishing a practical solution to solve the problem. YY always inspires me with her great visions on research problems and ideas. I cannot count how many times I was inspired by her thoughts of identifying impactful problem from the real world, and also insights of seeing the problems. I always learn a lot from working with YY. The first important lesson I learnt in the whole PhD study can be summarized in one word: “beyond”. Think beyond and outside the box so that one can see things in a way other than traditional angles, and also do beyond normal so that one can achieve higher. More importantly, YY makes a good example herself to teach me how to keep learning and adapt to new things quickly, and also to be patient to accumulate and gradually become solid in one particular aspect. These qualities developed in research finally also extend to my whole life to build my traits to be a better person. Lastly but not least, the most important thing YY have impacted on me is lightening the flame in my heart to have a dream, and also build me the confidence to take action to pursue it. The PhD journey is not the end, but a whole new start of another adventure in the whole life. Thank you YY for helping me to prepare myself for the next step of my life, via a such substantial and fun journey in the six years.

I would also like to express my appreciations to the other committee members – Darko Marinov, Sam King and Geoff Voelker for advice and help on my research. Thanks to Darko and Sam’s great lectures, I built my background on systems and also research and presentation skills in the early years of the PhD. And also thanks to Darko’s challenging questions, which taught me to think deeper on research directions and approaches. Geoff has always been encouraging and supporting not only on the hiking trails but also my PhD life in UCSD. Your advice guided me how to dig deeper into the problems and also be open minded to leverage others feedback.

I also feel so happy and grateful to my colleagues and also great friends in the Opera group. In research, Joe, Shan, Lin, Weihang, Soyeon, Xiao, Ding and Zuoning made good examples to
teach me how to be a good researcher. I also learnt numerous things from working and discussions with these senior students in the group, from how to critically think about a problem, presentation skills, and many many technical details. Joe leaded me into research on the first project in the group, from which I gradually built systems background and also started to love hacking software systems. I also appreciate Joe for guide me step by step on how to be a good system administrator for the group machines. It was a great experience of the whole PhD study. Thank Shan for the advices on how to do research and also think deep and tracking down a problem to the extreme. Thank Lin for the discussions on the early PhD study, and also walking me through the experiment and evaluation part of my first research paper in the PhD. I have worked with Soyeon in several projects, in which I learnt a lot about how to be responsible as a team member, and caring about details in paper writing, etc. I also thank Xiao, Ding and Zuoning for the countless discussions about research ideas, technical details and life during the whole PhD study. I have learnt so much about being professional, leadership and perseverance from them. It has also been a great time to work with Jiaqi and Soyeon on the thesis research. Thank you guys for the collaboration and great help through the thesis research. I feel grateful for your inspiring thoughts and great spirit throughout the project and paper deadlines. Thanks to Yoann for encouraging me to speak out ask stupid questions during the reading group presentations and discussions. I also thank all the other members in the group, including Weihang, Chongfeng, Tianwei, Yao, Jing, Ryan, Tianyin, Xinxin, Zhuoer, Rishan, Gen, to name a few here. Thanks for all the happy time together. All members in Opera make it like a big family and also make the PhD journey so much fun beyond research. The basketball games, tennis, badminton, and parties we had together are all good memory that last my whole life.

Thank all the other people in University of Illinois at Urbana Champaign(UIUC) and University of California, San Diego(UCSD). Specifically, thank professor Josep Torrellas, Darko Marinov, Sam King, Vikram Adve, Sanjay Kale and Marc Snir. I have built my background and research skills through the great lectures you gave, and also the qualifying exams. Thanks professor Sanjay Kale for his mentoring of my TA work. I am also thankful for Sheila Clark, Mary Beth Kelley, Elaine Wilson and Virginia McIlwain for their assitent in many administrative work. Thanks to the entire UCSD department, especially the Sysnet group, for the great syslunch talks, free pizzas, the social hour free food every week and also the great Chez Bob. Thanks to all the friends in UIUC and UCSD. I feel grateful to have met and shared with you all the happy time we had together in my life. You are the precious gifts that make my PhD life colorful.

I would also like to express my thankfulness to my collaborators in Intel and Microsoft Research Redmond. Thanks Zhiqiang Ma, Bob Kuhn, Matt Frank and Paul Peterson for their supports on my thesis work. Their industry experience and practices inspired me to think deep on the real world problems. Thanks also go to my internship mentor Aman Kansal and Feng Zhao for
giving me a great experience as a summer intern in Microsoft Research at Redmond.

No words can express my deepest appreciations to my family, who always support my choices to pursue my dreams unconditionally. I want to thank my parents for their initial support to my decision to come abroad for the PhD study. There is well known Confucius quote, “While one’s parents are alive, one should not travel to distant places”, while my parents have always been encouraging during my entire PhD study. I also want to thanks my sisters who have always been there to support and encourage me for every up and down moment. I give my deepest love to my dear wife, Han, who accompanied me to US and shared every piece of bitter and joy of the journey. You have always been my best half in my life. Lastly, I want to thank my parents-in-law for their love and supports.

My PhD study is supported by grants from NSF and Intel.
Chapter 5  Background and Related Works ............................ 63
5.1  Synchronization .......................................................... 63
   5.1.1  Mutex lock ......................................................... 63
   5.1.2  Waiting and Signaling .......................................... 63
   5.1.3  Semaphore ......................................................... 64
5.2  Concurrent Program Analysis Tools ................................. 64
   5.2.1  Concurrency Bug Detection ..................................... 64
   5.2.2  Spin and Hang Detection ....................................... 65
   5.2.3  Concurrent Program Testing and Failure Diagnosis ......... 66
   5.2.4  Synchronization Annotation ..................................... 66
5.3  Program Dependency Analysis ....................................... 66
5.4  Software Bug Characteristics Studies ............................. 67
5.5  Other Related Work .................................................... 68
   5.5.1  Lock-free and Wait-free Algorithm ......................... 68
   5.5.2  Concurrent Programming Language Design ................. 68

Chapter 6  Conclusions and Future Work ............................... 69

References ................................................................. 72
List of Tables

1.1 Percentages of ad hoc synchronizations that had introduced bugs according to the bugzilla databases and changelogs of the applications. ................................. 4
2.1 Ad hoc sync loops missed by human inspections ........................................ 13
2.2 The number of ad hoc synchronizations in concurrent programs we studied ...... 13
2.3 Diverse ad hoc synchronizations in concurrent programs we studied .............. 14
2.4 Observations in programmers’ comments on ad hoc synchronization from Apache, Mozilla, and MySQL ................................................................. 19
2.5 Relationships between volatile declaration and synchronization variables. ....... 23
2.6 The number of synchronizations giving up CPU via thread yield or sleep inside loops. ................................................................. 23
3.1 Loop mechanisms used for real-world ad hoc synchronization ....................... 31
3.2 The characteristics of writes to sync variables .......................................... 34
3.3 Overall results of SyncFinder: every concurrent program uses ad hoc sync loops except LU ................................................................. 37
3.4 EDV Analysis and non-sync variable pruning .......................................... 38
3.5 False synchronization pair pruning ......................................................... 38
3.6 Deadlock and bad practice detection ...................................................... 39
3.7 False positive reduction in Valgrind ....................................................... 39
4.1 The cases reported by existing data race detectors .................................... 42
4.2 The accuracy of Inspector ................................................................. 42
4.3 The causes of Not-A-Race reports from Intel Inspector ......................... 48
4.4 Applications used in the evaluation ...................................................... 56
4.5 Effectiveness of ATDetector in pruning out Inspector’s Not-A-Race reports related to address transfer ....................................................... 57
4.6 The size of register traces per thread .................................................. 57
List of Figures

1.1 Direct spinning loop on the sync variable. ........................................ 2
1.2 Address transfer implementations that result in Not-A-Race reports .......... 3
1.3 A deadlock introduced by an ad hoc synchronization in Apache. .............. 4
1.4 A deadlock caused by a circular wait among three threads .................... 5
1.5 An ad hoc synchronization missed in our manual identification process of our characteristic study but is identified by our auto-identification tool, SyncFinder ... 7
1.6 The real-world examples of address transfer ..................................... 8

2.1 Examples of various ad hoc synchronizations .................................. 15
2.2 Direct spinning loop on the sync variable. ...................................... 15
2.3 Multiple exist conditions in an ad hoc synchronization implementation .... 16
2.4 Control dependency in ad hoc synchronization ................................... 16
2.5 Useful work inside a waiting loop ................................................ 17
2.6 An ad hoc synchronization in MySQL was revised by programmers to solve a performance problem. ......................................................... 18
2.7 A simplified example demonstrating the danger of ad hoc synchronization on modern hardware with relaxed memory consistency ................. 19
2.8 False positives in Valgrind data race detection due to ad hoc synchronizations... 20
2.9 Replacing ad hoc synchronizations with synchronization primitives using condition variables .......................................................... 22

3.1 Ad hoc synchronization abstract model .......................................... 28
3.2 SyncFinder design to automatically identify and annotate ad hoc synchronization ............... 29
3.3 Leaf-EDV identification ......................................................... 32
3.4 The non-sync variables pruned out by loop-variant based pruning .......... 33

4.1 Real-world cases reported by the Intel® Inspector .......................... 43
4.2 The real-world examples of address transfer ................................. 45
4.3 Backtracking for address transfer detection ................................... 51
4.4 Overview of address transfer detection steps .................................. 51
4.5 The process of address receiving site identification ......................... 53
4.6 Multiple threads involved in the address transfer ............................ 54
4.7 The sensitivity to the size of register trace buffer ......................... 59
4.8 ATDetector’s detailed output to aid user inspection ......................... 60
Chapter 1

Introduction

Synchronization plays an important role in concurrent programs. Recently, partially due to realization of multicore processors, much work has been conducted on synchronization in concurrent programs. For example, various hardware/software designs and implementations have been proposed for transactional memory (TM) [RH, HM93, MTC+07, ST95] as ways to replace the cumbersome “lock” operations. Similar to TM, some new language constructs [VTD06, BGK+06, FF04] such as Atomizer [FF04] have also been proposed to address the atomicity problem. On a different but related note, various tools such as AVIO [LTQZ06], CHESS [MQB+08], CTrigger [PLZ09], ConTest [BFM+05a] have been built to detect or expose atomicity violations and data races in concurrent programs. In addition to atomicity synchronization, condition variables and monitor mechanisms have also been studied and used to ensure certain execution order among multiple threads [Hoa74, How76, LR80].

So far, most of the existing work has targeted only the synchronizations implemented in a modularized way, i.e., directly calling some primitives such as “lock/unlock” and “cond_wait/-cond_signal” from standard POSIX thread libraries or using customized interfaces implemented by programmers themselves. Such synchronization methods are easy to recognize by programmers, or bug detection and performance profiling tools.

Unfortunately, besides modularized synchronizations, programmers also use their own ways to do synchronizations. It is usually hard to tell the synchronizations apart from ordinary thread-local computations, making it difficult to recognize by other programmers for maintenance, or tools for bug detection and performance profiling. We refer to such synchronization as implicit synchronizations.

This dissertation investigates how implicit synchronizations impact the reliability of concurrent programs and how to improve on them. Based on the characteristics study on the implicit synchronizations it first proposes to automatically detect and annotate implicit ad hoc synchronizations to help developers or automatic concurrent program analysis tools to be aware of them. And then the dissertation proposes a solution to circumvent implicit synchronizations to help improve data race bug detection tools.
1.1 Implicit Synchronizations

1.1.1 Ad Hoc Synchronizations

Ad hoc synchronization is often used to ensure an intended execution order of certain operations. Specifically, instead of calling “cond_wait()” and “cond_signal()” or other synchronization primitives, programmers often use *ad hoc loops* to synchronize with some shared variables, referred to as *sync variables*. According to programmers’ comments, they are implemented this way due to either flexibility or performance reasons.

```c
/* “wait for the other guy finish (not efficient, but rare)” */
while (crc_table_empty);
write_table(.., crc_table[0]);
/* MySQL */
```

Figure 1.1: Direct spinning loop on the sync variable.

Figure 1.1 shows a real world example of ad hoc synchronizations from MySQL. This example uses a simple spinning loop to do synchronization. The spinning thread is waiting for some other threads by repetitively checking on one shared variables, in this case, `crc_table_empty`. The thread stops waiting until some other thread set `crc_table_empty` to false. This synchronization is not directly using a standard synchronization primitive like pthread_mutex_lock or pthread_cond_wait to synchronize with other threads. Instead, it is ad hoc in this specific application and case. This type of ad hoc synchronizations is often implicit thus not easy for programmers to be aware of. Nor do program analysis tools recognize the semantic and ad hoc synchronizations.

Unfortunately, there have been few studies on ad hoc synchronization. It is unclear how commonly it is used, how programmers implement it, what issues are associated with it, whether it is error-prone or not.

1.1.2 False Data Race Report Caused by Customized Synchronizations

Concurrent programming is well known to be difficult and may easily introduce bugs. One notorious type of concurrency bugs is data race, which occurs when multiple threads access the same memory location without correct synchronizations, and at least one of the access is write operation.

Most existing data race detectors, including even commercial strength race detectors such as Inspector, require accurate knowledge of synchronizations in the program, and may otherwise report false races (referred to as Not-A-Race in this dissertation for clarification) or benign races.

2
Intruder example in Figure 4.1(c)). Other synchronization primitives such as transactional memory (e.g., the Apache example in Figure 1.2(a)), customized locks (e.g., the Berkley-DB example in Figure 2.2), and also lock-free data structures (e.g., the Apache example in Figure 1.2(b)), and other synchronization primitives such as transactional memory (e.g., Intruder example in Figure 4.1(c)).

However, synchronizations are in many cases implemented in customized ways in practices, including the ad hoc synchronizations as shown in Figure 2.2, 2.3, 2.4, 2.5, and also lock-free data structures (e.g., the Apache example in Figure 1.2(a)), customized locks (e.g., the Berkley-DB example in Figure 1.2(b)), and other synchronization primitives such as transactional memory (e.g., Intruder example in Figure 4.1(c)).

In general, existing data race detectors recognize only standardized synchronization primitives (e.g., pthread_mutex_lock) or simple wrappers of those primitives. Consequently, they incorrectly report Not-A-Race pairs as data races.

### 1.2 Dissertation Contributions

#### 1.2.1 Ad Hoc Synchronization Study

In the first part of our work, we conduct a "forensic investigation" of 229 ad hoc synchronizations in 12 concurrent programs of various types (server, desktop and scientific), including Apache, MySQL, Mozilla, OpenLDAP, etc. The goal of our study is to understand the characteristics and implications of ad hoc synchronization in existing concurrent programs.

Our study has revealed several interesting, alarming and quantitative characteristics as follows:

1. **Every studied concurrent program uses ad hoc synchronization.** More specifically, there are 6–83 ad hoc synchronizations implemented using ad hoc loops in each of the 12 studied programs.
2. The fact that programmers often use ad hoc synchronization is likely due to two primary reasons:
   1. Unlike typical atomicity synchronization, when coordinating execution order among threads,
<table>
<thead>
<tr>
<th>Apps.</th>
<th>#ad hoc sync</th>
<th>#buggy sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>33</td>
<td>7 (22%)</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>15</td>
<td>10 (67%)</td>
</tr>
<tr>
<td>Cherokee</td>
<td>6</td>
<td>3 (50%)</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>17</td>
<td>5 (30%)</td>
</tr>
<tr>
<td>Transmission</td>
<td>13</td>
<td>8 (62%)</td>
</tr>
</tbody>
</table>

Table 1.1: Percentages of ad hoc synchronizations that had introduced bugs according to the bugzilla databases and changelogs of the applications.

![Figure 1.3: A deadlock introduced by an ad hoc synchronization in Apache.](image)

The intended synchronization scenario may vary from one to another, making it hard to use a common interface to fit every need (more discussion follows below and in Chapter 2); (ii) Performance concerns make some of the heavy-weight synchronization primitives less applicable.

(2) Although almost all ad hoc synchronizations are implemented using loops, the implementations are diverse, making it hard to manually identify them among the thousands of computation loops. For example, Figure 2.2 directly spins on a shared variable; Figure 2.3 has multiple exit conditions; Figure 2.4 shows the exit condition indirectly depends on the sync variable and needs complicated calculation to determine whether to exit the loop; Figure 2.5 synchronizes on program states and performs useful work while checking whether the remote thread has changed the states or not. Such characteristic may partially explain why programmers use ad hoc synchronizations. More discussion and examples are in Chapter 2.

(3) Ad hoc synchronizations are error-prone. Table 1.1 shows that among the five software systems we studied, significant percentages (22-67%) of ad hoc synchronizations introduced bugs. Although some experts may expect such results, our study is among the first to provide some quantitative results to back up this observation.

Ad hoc synchronization can easily introduce deadlocks or hangs. As shown on Figure 1.3,
Figure 1.4: A deadlock caused by a circular wait among three threads. (This is a new deadlock detected by our deadlock detector leveraging SyncFinder’s auto-annotation). Thread 2 is waiting at S2.2 for the lock to be released by thread 1; thread 1 is waiting at S1.2 for thread 3 to decrease the counter at S3.3; and thread 3 is waiting at S3.2 for thread 2 to decrease another counter at S2.3.

Apache had a deadlock in one of its ad hoc synchronizations. It holds a mutex while waiting on a sync variable “queue_info→idlers”. Figure 1.4 shows another deadlock example in MySQL, which has never been reported previously. In this example Thread 2 is waiting at S2.2 for the lock to be released by thread 1; thread 1 is waiting at S1.2 for thread 3 to decrease the counter at S3.3; and thread 3 is waiting at S3.2 for thread 2 to decrease another counter at S2.3. More details and the real world examples are in Chapter 2.

Because they are different from deadlocks caused by locks or other synchronization primitives, deadlocks involving ad hoc synchronizations are very hard to detect using existing tools or model checkers [EA03a, Sunb, LELS05b]. These tools cannot recognize ad hoc synchronizations unless these synchronizations are annotated manually by programmers or automatically by our SyncFinder described in Chapter 3. For the same reason, it is also hard for concurrency testing tools such as ConTest [BFM+05a] to expose these deadlock bugs during testing.

Furthermore, ad hoc synchronizations also have problems interacting with modern hardware’s weak memory consistency model and also with some compiler optimizations, e.g. loop invariant hoisting (discussed further in Chapter 2).

By studying the comments associated with ad hoc synchronizations, we found that some pro-
grammers knew their implementations might not be safe or optimal, but they still decided to keep their ad hoc implementations.

(4) *Ad hoc synchronizations significantly impact the effectiveness and accuracy of various bug detection and performance tuning tools.* Since most bug detection tools cannot recognize ad hoc synchronizations, they can miss many bugs related to those synchronizations, as well as introduce many false positives (details and examples in Chapter 2). For the same reason, performance profiling and tuning tools may confuse ad hoc synchronizations for computation loops, thus generating inaccurate or even misleading results.

### 1.2.2 Identifying Ad Hoc Synchronizations

Our characteristic study on ad hoc synchronization reveals that ad hoc synchronization is often harmful with respect to software correctness and performance. The first step to address the issues raised by ad hoc synchronization is to identify and annotate them, similar to the way that type annotation helps Deputy [CHA+07] and SafeDrive [ZCA+06] to identify memory issues in Linux. Specifically, if ad hoc synchronizations are annotated in concurrent programs, (1) static or dynamic concurrency bug (e.g. data race and deadlock) detectors can leverage such annotations to detect more bugs and prune more false positives caused by ad hoc synchronizations; (2) performance tools can be extended to capture bottlenecks related to these synchronizations; (3) new programming language/model designers can study ad hoc synchronizations to design or revise language constructs; (4) programmers can port such ad hoc synchronizations to more structured implementations.

Unfortunately, ad hoc synchronizations are very hard and time-consuming to recognize and annotate manually. Partly because of this, although some annotation languages for synchronizations like Sun Microsystems’ LockLint [loc] have been available for several years, they are rarely used, even in Sun’s own code [PTZ09]. Furthermore, manual examination is also error-prone. Figure 1.5 shows a MySQL ad hoc synchronization example that we missed during the manual identification we conducted for our characteristic study. Fortunately, our automatic identification tool SyncFinder found it. We overlooked this example because of the complicated nested “goto” loops.

Motivated by the above reasons, the second part of our work involved building a tool called **SyncFinder** to automatically identify and annotate ad hoc synchronizations in concurrent programs. SyncFinder statically analyzes source code using inter-procedural, control and data flow analysis, and leverages several of our observations and insights gained from our study to distinguish ad hoc synchronizations apart from thousands of computation loops.

We evaluate SyncFinder with 25 concurrent programs including the 12 used in our charac-
Figure 1.5: An ad hoc synchronization missed in our manual identification process of our characteristic study but is identified by our auto-identification tool, SyncFinder. The interlocked “goto” loops can easily be missed by manual identification (Figure 2.5 shows more detailed code).

SyncFinder automatically identifies and annotates 96% of ad hoc synchronization loops with 6% false positives on average.

To demonstrate the benefits of auto-annotation of ad hoc synchronizations by SyncFinder, we design and evaluate two use cases. In the first use case, we build a simple wait-inside-critical-section detector, which can identify deadlock and bad programming practices involving ad hoc synchronizations. In our evaluation, our tool detects five deadlocks that are missed by previous deadlock detection tools in Apache, MySQL and Mozilla, and, moreover, *two of the five are new bugs and have never been reported before*. In addition, even though some(16) of the detected issues are not deadlocks, they are still bad practices and may introduce some performance issues or future deadlocks. The synchronization waiting loop inside a critical section protected by locks can potentially cause cascading wait effects among threads.

As the second use case, we extend the Valgrind [NS07] data race checker to leverage the ad hoc synchronization information annotated by SyncFinder. As a result, Valgrind’s false positive rates for data races decrease by 43–86%. This indicates that even though SyncFinder is not a bug detector itself, it can help concurrency bug detectors improve their accuracy by providing ad hoc synchronization information.

### 1.2.3 Improving Data Race Detector by Detecting Address Transfer

Implicit synchronizations (e.g., ad hoc synchronizations or other customized synchronizations) impact the accuracy and effectiveness of data race detectors. To address the issue, this dissertation first studies why an existing data race detector generates Not-A-Race reports. Our study makes a unique observation of **address transfer**, which contributes to a majority of Not-A-Race reports. Based on this observation, the second part of our study builds a tool called **ATDetector** to effectively identify address transfer related to data race candidates and eliminate them from the data race reports. As a result, we improve the accuracy of Inspector by eliminating *all* Not-A-Race
Consequently, they incorrectly report the synchronization primitives (e.g., pthread example in Figure 1.2(c)). In general, existing data race detectors recognize only standardized synchronization primitives recently developed such as transactional memory (e.g., Intruder, Inspector, may not be aware of the happens-before relations introduced by address transfer. It is

Figure 1.6: The real-world examples of address transfer. The addresses transferred are highlighted. In addition to these two, another use case of address transfer is task dispatching, which is shown in Figure 4.1(a). Due to the happens-before relation introduced by address transfer, S1 and S2 never race each other.

reports caused by unrecognized address transfer.

**Unique Observation of Address Transfer:** In multi-threaded applications, there are two common ways for multiple threads to access shared data. The memory address may be globally known to all threads from the beginning, and thus they can immediately access the data with its own knowledge of the location. Alternatively, the location to access may be made known to one thread by another at a certain execution point. In this case, one thread explicitly passes a memory address to another thread to give the location information. We refer to this as address transfer. We further identified three use cases of address transfer: (i)Task dispatching, (ii)Memory recycling, and (iii)Producer-consumer communication for data sharing. Their details are in Section 4.2.2.

Address transfer implicitly introduces a happens-before relation between the sender’s and the receiver’s memory accesses to the transferred address. For instance, in all the three examples in Figure 4.1(a) and Figure 4.2, the memory access at S1 by thread 1 always happens before the same memory access at S2 by thread 2. Since S2 uses the received address either directly or indirectly, it causally depends on the address transfer process. Again, the address transfer can happen only after S1.

Importantly, however, existing data race detectors, including commercial strength tools such as Inspector, may not be aware of the happens-before relations introduced by address transfer. It is because address transfer is often implemented with application-specific methods, such as lock-free data structure (e.g., the Apache example in Figure 1.2(a)), customized locks (e.g., the Berkley-DB example in Figure 1.2(b)), system calls (e.g., TransmissionBT example in Figure 4.1(a)), and other synchronization primitives recently developed such as transactional memory (e.g., Intruder example in Figure 1.2(c)). In general, existing data race detectors recognize only standardized synchronization primitives (e.g., pthread_mutex_lock). Consequently, they incorrectly report the pairs of S1 and S2 in all the examples as data races, while all of them are Not-A-Race.

Specifically, according to our investigation of the results from Inspector with 17 diverse open-source applications, the majority of Not-A-Race reports (62% of 1420) are caused by the unrecog-
nized address transfer (the details are shown in Table 4.3).

**Address Transfer Detection (ATDetector):** Based on the observation of address transfer, we propose a method called ATDetector that automatically identifies address transfer and uses the information to prune Not-A-Race reports. ATDetector focuses on identifying only those address transfers that are related to data race candidates collected by data race detectors.

Given a data race candidate and its memory address, starting from the later race instruction candidate, ATDetector backtracks all the relevant address propagation paths within a thread and also across multiple threads. Meanwhile, it collects all potential address receiving and sending sites, which can be shared memory read/write or certain system calls. In this way, we can recognize address transfer regardless of the customized way of implementation. Finally, if one of the address transfer candidates indeed breaks the potential race condition by enforcing happens-before relation, the data race candidate is pruned out.

To implement this idea, ATDetector conducts binary instrumentation with Pin [CRRea05] to trace memory and register accesses, and performs online dependency analysis with the traces when it encounters potential race candidates.

**Improving the accuracy of Inspector, a state-of-the-art commercial data race detector:** We used ATDetector to improve the accuracy of Inspector by eliminating Not-A-Race reports caused by unrecognized address transfer without any users’ input. Since the source code of Inspector is not available, instead of integrating ATDetector to the tool, we run it by feeding Inspector’s data race reports to ATDetector as inputs.

We evaluated ATDetector with 8 diverse open-source server, desktop, and scientific programs, including Berkeley-DB, Apache, TransmissionBT, etc. As a result, ATDetector can help eliminate all of the Not-A-Race reports caused by unrecognized address transfer, which are 62% of the total Not-A-Race reports from Inspector. More importantly, ATDetector does not eliminate any true data races that are originally reported. It has a modest memory overhead ranging from 2KB to 2MB per thread for most of the evaluated applications.

### 1.3 Outline

The remainder of this dissertation is organized as follows: Chapter 2 presents the characteristics study result of ad hoc synchronizations and discusses the findings, followed by Chapter 3, the descriptions of SyncFinder, the ad hoc synchronization detection and annotation tool. Then in Chapter 4 the dissertation discusses address transfer, a happens before relation to improve data race detector accuracy and effectiveness. Chapter 5 describes related work on synchronizations and concurrent program analysis, followed by the conclusions and future work.
The materials in some chapters have been published as conference papers. Materials in Chapter 2 and 3 have appeared in the Symposium on Operating Systems Design and Implementation (OSDI’12) [XPZ+10a]. The materials in Chapter 4 were published in the IEEE/ACM International Symposium on Microarchitecture 2011 (MICRO’11) [ZXP+11].

The results of all the works presented in the dissertation are the results of a group effort. Please refer to the conference publications for the list of all collaborators who contributed to the dissertation [XPZ+10a, ZXP+11].
Chapter 2

Characteristics Study of Ad Hoc Synchronizations

2.1 Overview

This Chapter describes a quantitative characteristics study of ad hoc synchronizations. In order to understand ad hoc synchronization characteristics, we have manually studied 12 representative applications of three types (server, desktop and scientific/graphic), as shown on Table 2.2. Two inspectors separately investigated almost every line of source code and compared the results with each other. As shown on Table 2.1, in our initial study, we missed a few ad hoc synchronizations, most of which are those implemented using interlocked or nested goto loops (e.g., the example in Figure 1.5). Fortunately, our automatic identification tool, SyncFinder, discovers them, and we were able to extend our manual examination to include such complicated types.

2.1.1 Motivation

Synchronization plays an important role in concurrent programs. In recent years, multi-core processors become more and more popular. In order to take advantage of multi-core hardware, more and more applications are becoming multi-threaded. However, it is always difficult to do correct synchronizations, thus concurrent programs are prone to bugs such as data races and deadlocks.

Modularized synchronization primitives, such as Standard POSIX thread library functions, are well recognized and commonly used in concurrent programs. So far, most of existing work has also targeted only these modularized synchronizations. Such synchronization methods are easy to recognize, by programmers, bug detection or performance profiling tools.

Unfortunately, not all synchronizations are easy to recognize, but rather are implicit and often hard to be recognized by programmers or program analysis tools, because they are often mixed together with normal computations. We refer to such synchronization as implicit synchronizations.

One type of synchronization, which we refer to as ad hoc synchronizations here, is commonly used in concurrent programs. In order to understand more about this type of synchronizations, in this Chapter we convey a quantitative study to understand how ad hoc synchronizations impact the reliability of concurrent programs and how to improve on them. This study also servers as the first
step towards improving the reliability of concurrent bugs related to implicit synchronizations.

2.1.2 Summary of Contributions

We examined 229 ad hoc synchronizations in 12 concurrent programs of various types including server applications, desktop applications and scientific programs. After this study, we have made two major contributions in this part: (1) Interesting, and maybe also alarming findings regarding to ad hoc synchronizations. (2) Insights of ad hoc synchronizations to be used for further research on identifying and improve them.

The major findings from the study include (1) Every studied application uses ad hoc synchronizations. Specifically, there are 6-83 ad hoc synchronizations in each program. (2) Ad hoc synchronizations are error-prone. Significant percentages (22-67%) of these ad hoc synchronizations introduced bugs or severe performance issues. (3) Ad hoc synchronization implementations are diverse and many of them cannot be easily recognized as synchronizations, i.e., have poor readability and maintainability. Through these findings, we send a signal to programmers that ad hoc synchronizations are often harmful in several respects. Therefore it is desirable that programmers use synchronization primitives such as cond_wait, rather than ad hoc synchronizations in concurrent programs.

2.2 Threats to Validity

Similar to previous work, characteristic studies are all subject to the validity problem. Potential threats to the validity of our characteristic study are the representativeness of applications and our examination methodology. To address the former, we chose a variety of concurrent programs, including four servers, three client/desktop concurrent applications as well as five scientific applications from SPLASH-2, all written in C/C++, one of the popular languages for concurrent programs. These applications are well representative of server, client/desktop-based and scientific applications, three large classes of concurrent programs.

In terms of our examination methodology, we have examined almost every line of code including programmers’ comments. This was an immensely time consuming effort that took three months of our time. To ensure correctness, the process was repeated twice, each time by a different author. Furthermore, we were also quite familiar with the examined applications, since we have modified and used them in many of our previously published studies.

Overall, while we cannot draw any general conclusions that can be applied to all concurrent programs, we believe that our study does capture the characteristics of synchronizations in three large important classes of concurrent applications written in C/C++. 
Table 2.1: Ad hoc sync loops missed by human inspections. Two inspectors, $I_a$ and $I_b$, investigate the same source code separately. Most of the sync loops missed by both inspectors (i.e., those in Apache and MySQL) are interlocked or nested goto loops. Others (in OpenLDAP) are for-loops doing complicated useful work and checking synchronization condition in it, like one in Figure 2.5.

<table>
<thead>
<tr>
<th>Apps</th>
<th>#sync loops</th>
<th>$I_a$</th>
<th>$I_b$</th>
<th>both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>33</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>83</td>
<td>12</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PBZip2</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: The number of ad hoc synchronizations in concurrent programs we studied. Ad hoc sync is implemented with an ad hoc loop using shared variables (i.e., sync variables) in it.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Desc.</th>
<th>LOC.</th>
<th>Total loops</th>
<th>Ad hoc loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.2.14</td>
<td>Web server</td>
<td>228K</td>
<td>1462</td>
<td>33</td>
</tr>
<tr>
<td>MySQL 5.0.86</td>
<td>Database server</td>
<td>1.0M</td>
<td>4265</td>
<td>83</td>
</tr>
<tr>
<td>OpenLDAP 2.4.21</td>
<td>LDAP server</td>
<td>272K</td>
<td>2044</td>
<td>15</td>
</tr>
<tr>
<td>Cherokee 0.99.44</td>
<td>Web server</td>
<td>60K</td>
<td>748</td>
<td>6</td>
</tr>
<tr>
<td>Mozilla-js 0.9.1</td>
<td>JS engine</td>
<td>214K</td>
<td>848</td>
<td>17</td>
</tr>
<tr>
<td>PBZip2 2-1.1.1</td>
<td>Parallel bzip2</td>
<td>3.6K</td>
<td>45</td>
<td>7</td>
</tr>
<tr>
<td>Transmission 1.83</td>
<td>BitTorrent client</td>
<td>96K</td>
<td>1114</td>
<td>13</td>
</tr>
<tr>
<td>Radiosity</td>
<td>SPLASH-2</td>
<td>14K</td>
<td>80</td>
<td>12</td>
</tr>
<tr>
<td>Barnes</td>
<td>SPLASH-2</td>
<td>2.3K</td>
<td>88</td>
<td>7</td>
</tr>
<tr>
<td>Water</td>
<td>SPLASH-2</td>
<td>1.5K</td>
<td>84</td>
<td>9</td>
</tr>
<tr>
<td>Ocean</td>
<td>SPLASH-2</td>
<td>4.0K</td>
<td>339</td>
<td>20</td>
</tr>
<tr>
<td>FFT</td>
<td>SPLASH-2</td>
<td>1.0K</td>
<td>57</td>
<td>7</td>
</tr>
</tbody>
</table>

2.3 Common Practice of Ad Hoc Synchronizations

The first major finding from studying ad hoc synchronizations is that every studied application uses ad hoc synchronizations. More specifically, there are 6–83 ad hoc synchronizations in each of the 12 studied programs. As shown in Table 2.2, ad hoc synchronizations are used in all of our evaluated programs, and some programs (e.g. MySQL) even use as many as 83 ad hoc synchronizations. This indicates that, in the real world, it is not rare for programmers to use ad hoc synchronizations in their concurrent programs.

While we are not 100% sure why programmers use ad hoc synchronizations, after studying the code and comments, we speculate there are two primary reasons. The first is because there are diverse synchronization needs to ensure execution order among threads. Unlike atomicity synchronization that shares a common goal, the exact synchronization scenario for order assurance
Table 2.3: Diverse ad hoc synchronizations in concurrent programs we studied. (i) The number of exit conditions in synchronization loops are various (sc vs. mc); (ii) There can be multiple, different types of dependency relations between sync variables and loop exit conditions (-dir, -df, -cf, -func); (iii) Some synchronization loops do useful work with asynchronous condition checking (async). In the table, sc denotes “single condition”; mc denotes “multiple conditions”; -dir denotes “directly depends on a sync variable”; df denotes “data dependency”; cf denotes “control dependency”; -all denotes “all exit conditions depend on sync variables”; -Nall denotes “not all, but at least one depends on sync variables”; func denotes “inter-procedural dependency”; and async denotes “useful work while waiting”.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Total loops</th>
<th>Total Ad hoc</th>
<th>Single exit condition</th>
<th>Multiple exit cond.</th>
<th>Total async</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>sc-dir</td>
<td>sc-df</td>
<td>sc-cf</td>
</tr>
<tr>
<td>Apache</td>
<td>1462</td>
<td>33</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>4265</td>
<td>83</td>
<td>23</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>2044</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cherokee</td>
<td>748</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>848</td>
<td>17</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>PBZip2</td>
<td>45</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transmission</td>
<td>1114</td>
<td>13</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Radiosity</td>
<td>80</td>
<td>12</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Barnes</td>
<td>88</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Water</td>
<td>84</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ocean</td>
<td>339</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT</td>
<td>57</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

may vary from one to another, making it hard to design a common interface to fit every need (more discussion in Finding 2).

The second reason is due to performance concerns on synchronization primitives, especially those heavyweight ones implemented as system calls. If the synchronization condition can be satisfied quickly, there is no need to pay the high overhead of context switches and system calls. Such performance justifications are frequently mentioned in programmers’ comments associated with ad hoc synchronization implementations.

While ad hoc synchronizations are seemingly justified, are they really worthwhile? What are their impact on program correctness and interaction with other tools? Can they be expressed using some common, easy-to-recognize synchronization primitives? We will dive into these questions in our finding 3 and 4, trying to shed some lights into the tradeoffs.
while(1) {
    int oldcount = (global->barrier).count;
    ... 
    If(updatedcount == oldcount) break;
} /* SPLASH2 */

int finished = 0;
for(i = 0; i < 1000 && !finished; i++) {
    if(global->pbar_count >= n_proc)
        finished = 1;
} /* Radiosity*/

(a) sc-df (data dependency)
(b) mc-Nall (some are local exit conditions)

/* wait for the next block from a producer queue */
safe_mutex_lock(fifo->mut);
for(;;) {
    if(!queue->empty && queue->getData(Data))
        break;
}

/* PBZip2 */
Bool queue::getData(ElemPtr &fileData) {
    ElemPtr &headElem = qData[head];
    ...
    /* search qData to find the requested block. If finds out, return true;
     * otherwise, return false */
}

(c) Function call

/* MySQL */

while (crc_table_empty);
write_table(.., crc_table[0]);

2.4 Diversity of Ad Hoc Synchronizations

By checking the detail implementation of each ad hoc synchronization, we found they are quite diverse. Table 2.3 further categorizes ad hoc synchronizations from several perspectives. Some real world examples for each category can be found in Figure 2.1, 2.2, 2.3, 2.4 and 2.5.

Some ad hoc synchronization loops have only one exit condition \(^1\). We call such sync loops \(sc\) loops. Unfortunately, many others (up to 86% of ad hoc synchronizations in a program) have more than one exit condition. We refer to them as \(mc\) loops. In some of them (referred to as \(mc\_all\)), all exit conditions are satisfied by remote threads. In the other loops (referred to as \(mc\_Nall\)), there are also some local exit conditions such as time-outs, etc., that are independent of remote threads.

\(^1\)A condition that can break the execution out of a loop.
/* wait until some waiting threads enter */
while(group->waiter->count == 0) {
    ... 
    /* abort if the group is not running */
    if(group->state != _prmw_running) {
        PR_SetError(..);
        goto aborted;
    }
} /* Mozilla */

Figure 2.3: Multiple exist conditions in an ad hoc synchronization implementation

for (deleted=0; ;) {
    THREAD_LOCK(., dbmp->mutex);
    /* wait for other threads to release their
     * reference to dbmfp */
    if(dbmfp->ref == 1) {
        if (F_ISSET(dbmfp, MP_OPEN_CALLED))
            TAILQ_REMOVE(&dbmp->dbmfq, ..);
        deleted = 1;
    }
    THREAD_UNLOCK(., dbmp->mutex);
    if (deleted)
        break;
    __os_sleep(dbenv, 1, 0);
} /* OpenLDAP */

Figure 2.4: Control dependency in ad hoc synchronization
and can be satisfied locally.

(ii) Dependency on sync variables: The simplest ad hoc synchronization is just directly spinning on a sync variable as shown on Figure 2.2. In many other cases (50-100% of ad hoc synchronizations in a program), exit conditions indirectly depend on sync variables via data dependencies (referred to as \(df\), Figure 2.1(a)), control dependencies (referred to as \(cf\), (Figure 2.4), even interprocedural dependencies (referred to as \(func\), Figure 2.1(c)).

(iii) Asynchronous synchronizations (referred to as async): In some cases (77% of ad hoc synchronizations in server/desktop applications we studied), a thread does not just wait in synchronization. Instead, it also performs some useful computations while repetitively checking sync variables at every iteration. For example, in Figure 2.5, a MySQL master thread does background tasks like log flushing until a new SQL query arrives (by checking \(new\_activity\_counter\)).

2.5 Ad Hoc Synchronizations are Error-prone

Ad hoc synchronizations can easily introduce bugs or performance issues. After studying the 5 applications listed in Table 1.1, we found that 22–67% of synchronization loops previously introduced bugs or performance issues. These high issue rates are alarming, and, as a whole, may be a strong sign that programmers should stay away from ad hoc synchronizations.

For each ad hoc synchronization loop, we use its corresponding file and function names to
/* get tuple id of a table */
do {
    ret = m_skip_auto_increment ?
        readAutoIncrementValue(...) :
        getAutoIncrementValue(...);
} while (ret == -1 && --retries && ..);

for (;;) {
    if (m_skip_auto_increment
        && readAutoIncrementValue(...) || getAutoIncrementValue(...) {
        if (--retries && ..) {
            my_sleep(retry_sleep);
            continue;  /* 30 ms sleep for transaction */
        }
    } break;
}

Figure 2.6: An ad hoc synchronization in MySQL was revised by programmers to solve a performance problem.

find out in the source code repository if there was any patch associated with it. If there is, we manually check if the patch involves the ad hoc sync loop. We then use this patch’s information to search the bugzilla databases and commit logs to find all relevant information. By examining such information as well as the patch code, we identify whether the patch is a feature addition, a bug not related to synchronization, or a bug caused exactly by the ad hoc sync loop. We only count the last case.

Besides deadlocks (as demonstrated in Figure 1.3 and 1.4), ad hoc synchronization can also introduce other types of concurrency bugs. In some cases, an ad hoc synchronization fails to guarantee an expected order and lead to a crash because the exit condition can be satisfied by a third thread unexpectedly. Due to space limitations, we do not show those examples here.

In addition to bugs, ad hoc synchronizations can also introduce performance issues. Figure 2.6 shows such an example. In this case, the busy wait can waste CPU cycles and decrease throughput. Therefore, programmers revised the synchronization by adding a sleep inside the loop.

Ad hoc synchronizations also have problematic interactions with modern hardware’s relaxed consistency models [BA08, MPV05, VG96]. These modern microprocessors can reorder two writes to different locations, making ad hoc synchronizations such as the one in Figure ??(a) fail to
guarantee the intended order in some cases. As such, experts recommended programmers to stay away from such ad hoc synchronization implementations, or at least implement synchronizations using atomic instructions instead of just simple reads or writes [BA08, MPV05, VG96].

Figure 2.7 borrows an example from S.Adve’s presentation in PODC, SPAA keynote speech [Adv], showing the danger of some ad hoc synchronization interacting with modern hardware that supports relaxed memory consistency models.

```
Initially A = Flag = 0

thread 1            thread 2
S1: A = 90;          S3: while(Flag!=1) {}  
...                   S4: r1 = A;   // 90? No, still 0!
S2: Flag = 1;        ...
```

Figure 2.7: A simplified example demonstrating the danger of ad hoc synchronization on modern hardware with relaxed memory consistency. The expected order, S1 → S4 may not be guaranteed by the possible reorder between data access(S1) and ad hoc synchronization write(S2).

Table 2.4: Observations in programmers’ comments on ad hoc synchronization from Apache, Mozilla, and MySQL. We study 63 comments associated with ad hoc synchronizations.

<table>
<thead>
<tr>
<th>Comment examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmers are aware of better design but still use ad hoc implementation (8%)</td>
</tr>
<tr>
<td>/* This can be built in smarter way, like pthread_con, but we do it since the</td>
</tr>
<tr>
<td>status can come from.. */</td>
</tr>
<tr>
<td>/* By doing.. applications will get better performance and avoid the problem</td>
</tr>
<tr>
<td>entirely. Regardless, we do this... because we’d rather write error message in</td>
</tr>
<tr>
<td>this routine, ..*/</td>
</tr>
<tr>
<td>Programmers try to prevent bugs at the first place (22%)</td>
</tr>
<tr>
<td>/* We could end up spinning indefinitely with a situation where.. The ‘i++’ stops</td>
</tr>
<tr>
<td>the infinite loop */ We can safely wait here in the case.. without fear of dead-</td>
</tr>
<tr>
<td>lock because we made.. */ */ This spinning actually isn’t necessary except when</td>
</tr>
<tr>
<td>the compiler does corrupt 64bit arithmetic.. */</td>
</tr>
<tr>
<td>Programmers explicitly state their sync assumptions (75%)</td>
</tr>
<tr>
<td>/* GC doesn’t set the flag until it has waited for all active requests to end</td>
</tr>
<tr>
<td><em>/ /</em> We must break the wait if one of the following occurs: i).. ii).. iii)..</td>
</tr>
<tr>
<td>iv).. v).. */</td>
</tr>
</tbody>
</table>

To make things even worse, ad hoc synchronizations also have problematic interactions with compiler optimizations such as loop invariant hoisting. Programmers should avoid such optimizations on sync variables, and ensure that waiting loops always read the up-to-date values instead
of the cached values from registers. As a workaround, programmers may need to use wrapping variable accesses with function calls [cer]. All of these just complicate programming as well as software testing and debugging.

Interestingly, some programmers are aware of the above ad hoc synchronization problems but still use them. We study the 63 comments associated with ad hoc synchronizations in MySQL, Apache, and Mozilla. As illustrated in Table 2.4, programmers sometimes mentioned better alternatives, but they still chose to use their ad hoc implementations for flexibility. In some cases, they explicitly indicated their preference for the lightness and simplicity of ad hoc spinning loops, especially when the synchronizations were expected to rarely occur or rarely need to wait long. Also, programmers often explicitly stated their assumptions/expectation in comments about what remote threads should do correspondingly, since ad hoc synchronizations are complex and hard to understand.

### 2.6 Impacts on Concurrency Program Analysis

Ad hoc synchronizations can significantly impact the effectiveness and accuracy of concurrency bug detection and performance profiling tools. As mentioned earlier, since existing concurrency bug (deadlock, data race) detection tools cannot recognize ad hoc synchronizations, they will fail to detect bugs that involve such synchronizations (e.g. deadlock examples shown on Figure 1.3
and 1.4).

In addition, they can also introduce many false positives. It has been well known that most data race detectors incur high false positives due to ad hoc synchronizations. Such false positives come from two sources: (1) Benign data races on sync variables: typically an ad hoc synchronization is implemented via an intended data race on sync variables. Figure 2.8(a) shows such a benign data race reported by Valgrind [NS07] in MySQL. (2) False data races that would never execute in parallel due to the execution order guaranteed by ad hoc synchronizations: For example, in Figure 2.8(b), the two threads are synchronized at S2 and S3, which guarantees the correct order between S1 and S4’s accesses to $q\text{--info--pools}$. S1 and S4 would never race with each other. However, most data race checkers cannot recognize this ad hoc synchronization and, as a result, incorrectly report S1 and S4 as a data race.

Synchronization is also a big performance and scalability concern because time waiting at synchronization is wasted. Unfortunately, existing work in synchronization cost analysis [LLS06, NSKI05] and performance profiling [MCC+95] cannot recognize ad hoc synchronizations, and therefore the synchronizations can easily be mistaken as computation. As a result, the final performance profiling results may cause programmers to make less optimal or even incorrect decisions while performance tuning.

2.7 Replacing with Synchronization Primitives

Our findings above reveal that ad hoc synchronization is often harmful in several respects. Therefore, it is desirable that programmers use synchronization primitives such as cond_wait, rather than ad hoc synchronization. Figure 2.9 shows how ad hoc synchronization can be replaced with a well-known synchronization primitive, POSIX pthread_cond_wait(). Note that it may not always be straightforward to use existing synchronization primitives to replace all ad hoc synchronizations, because existing synchronization primitives may not be sufficient to meet the diverse synchronization needs as well as the performance requirements, as discussed in Finding 1.

2.8 Discussions

Besides above major findings, we also observed several other characteristics that may worth to be aware of. Some of them lead to interesting discussions as presented in this section.

2.8.1 Volatile Variables

All volatile variables are sync variables, but not all sync variables are declared volatile. As shown
/* “wait for the other guy to finish (not efficient, but rare)” */
while (crc_table_empty);
write_table(out, crc_table[0]);

pthread_mutex_lock(&mutex);
while (crc_table_empty) {
    pthread_cond_wait(&cond_var, &mutex);
}
pthread_mutex_unlock(&mutex);
write_table(out, crc_table[0]);

while(1) {
    int oldcount = (global->barrier).count;
    ...
    if(updatedcount == oldcount) break;
}

pthread_mutex_lock(&mutex);
while(1) {
    int oldcount = (global->barrier).count;
    ...
    if(updatecount == oldcount) break;
    pthread_cond_wait(&cond_var, &mutex);
}
pthread_mutex_unlock(&mutex);

(a) MySQL

(b) SPLASH2

Figure 2.9: Replacing ad hoc synchronizations with synchronization primitives using condition variables. (a) shows the re-implementation of ad hoc synchronization in Figure 2.2; (b) is for Figure 2.1(a).
**Table 2.5:** Relationships between volatile declaration and synchronization variables.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Sync var.</th>
<th>Non-sync var.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td>Apache</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>MySQL</td>
<td>24</td>
<td>37</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>0</td>
<td>33</td>
</tr>
</tbody>
</table>

on Table 2.5, not all sync variables (and none in Mozilla-js) are declared volatile. However, all variables declared as volatile have been used as sync variables.

### 2.8.2 Server v.s. Desktop Applications

Server applications have more spin-blocks while desktop applications just spin at order synchronizations. As shown on Table 2.6, throughput-conscious server applications like Apache and MySQL implement 27-30% of ad hoc synchronizations use spin-blocks instead of just busy waiting. In contrast, responsiveness-centric desktop applications like Mozilla-js only has one spin-block and the rest just spins on sync variables without giving up CPU via thread yield or sleep.

### 2.8.3 Why Developers Use Ad Hoc Synchronizations

As shown in Table 2.4, we could divide developers’ comments into three categories.

First, with regards to performance, they put reasons why they use ad hoc way instead of primitives, or why ad hoc way is still fine for their purpose.

Second, they are concerned about potential dangers around ad hoc synchronizations such as not-happened-yet but possible deadlocks due to their ad hoc order synchronization, and prevent that in ad hoc way. Sometimes, they are aware of benign data races due to their ad hoc order synchronization and reason about why it is not a problem in their design.

Third, developers often want to describe semantic information related to their order synchronization, such as what event a synchronization loop is waiting for.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Ad hoc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>Apache</td>
<td>33</td>
</tr>
<tr>
<td>MySQL</td>
<td>83</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>15</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 2.6: The number of synchronizations giving up CPU via thread yield or sleep inside loops.
Those comments imply that even though developers choose to use ad hoc order synchronization, they pay attention to potential issues that it may introduce. It also indicates that order synchronization condition is often complex in a real world applications, and therefore it would be good if new programming paradigm will come to help developers relieve from those concerns.

We further reason about this important issue with all the real world order synchronization we manually collected. As a result, we made the following observation.

**Performance concern.** Even though it is feasible to use existing primitives, programmers often have performance concern when using primitives, since many existing blocking-wait primitives are using heavyweight system calls, signal, and wake up mechanism.

Especially, if developers know that in many cases a synchronization condition will shortly be satisfied, they do not want to have context switch and system call overhead with primitives, but take their own ad hoc way. However, please note that even a simple ad hoc order synchronization like `sc_dir` can raise many problematic issues that we mentioned in Chapter 1.

**The lack of flexibility to meet developers’ functional demand** Interestingly, not all of ad hoc order synchronizations can be effectively substituted with existing order synchronization primitives (e.g., `pthread_cond_wait`, `signal`), while still preserving the same functionality.

It is mainly because almost all existing primitives are blocking wait (w/ or w/o timeout) and do not allow much flexibility for developers to achieve intended purpose: for example, (i) it is difficult to deal with multiple synchronization conditions, since until one condition is satisfied (or until timeout), any other conditions cannot continuously checked; (ii) blocking wait may not be good design choice if a thread needs to do computation and make an application progress while waiting for order synchronization; (iii) conventional order primitives usually require developers to explicitly make a pair among a synchronization read and writes. However, in real world programs, it is not always necessary (e.g., Figure 2.5), but it is enough for a waiting thread to be synchronized with any peers.

### 2.9 Summary

This Chapter describes a quantitative characteristics study of ad hoc synchronizations in concurrent programs. By examining 229 ad hoc synchronization loops from 12 concurrent programs, we found several interesting and alarming characteristics. Among them, the most important results include: all concurrent programs have used ad hoc synchronizations and their implementations are very diverse and hard to recognize manually. Moreover, a large percentage (22-67%) of ad hoc loops in these applications have introduced bugs or performance issues. They also greatly impact the accuracy and effectiveness of bug detection and performance profiling tools.
In the next Chapter, I describe SyncFinder, a tool that automatically detects and annotates ad hoc synchronizations in concurrent programs, based on the insights from the study in this Chapter.
Chapter 3

Ad Hoc Synchronizations Detection and Annotation

3.1 Overview

3.1.1 Motivation

As ad hoc synchronizations have raised many challenges and issues related to correctness and performance, it would be useful to identify and annotate them. Manually doing this is tedious and error-prone since they are diverse and hard to tell apart from computation. Therefore, in this Chapter we build a tool called SyncFinder to automatically identify and annotate them in the source code of concurrent programs. The annotation can be leveraged in several ways as discussed in Chapter 1.

3.1.2 Related Works

Spin and Hang Detection

There are some previous works in detecting simple spinning-based synchronizations [NSKI05, LLS06, JT10a]. For example, [LLS06] proposed some new hardware buffers to detect spinning loops on-the-fly. [JT10a] also provides similar capability but does it in software. Both can detect only simple spinning loops, i.e. those sync loops with only one single exit condition and also directly depend on sync variables (referred to as “sc-dir” in Table 2.3 in Chapter 2). As shown in Table 2.3 such simple spinning loops account for less than 16% of ad hoc sync loops on average in server/desktop applications we studied.

Besides, both of them are dynamic approaches and thereby suffer from the coverage limitation of all dynamic approaches (discussed in Chapter 3). In contrast, SyncFinder uses a static approach and can detect various types of ad hoc synchronizations. Additionally, we also conduct an ad hoc synchronization characteristic study.
Synchronization Annotation

Many annotation languages [sal, loc, jav, Ste93] have been proposed for synchronizations in concurrent programs. Unfortunately, annotation is not frequently used by programmers since it is tedious. SyncFinder is complementary to these work by providing automatic annotation for ad hoc synchronizations.

Program Dependency Analysis

Several works exist for program dependency analysis such as dynamic dependency profiling, memory tainting, and backtracking. Dynamic dependency profiling [XR04] builds the program’s dependency graph by analyzing the execution traces, and is widely used to automatically parallelize sequential code. SD\(^3\) [MHC10] accelerates this process and reduces the memory overhead by parallelizing the dependency profiling steps and compressing the traces with stride patterns. Memory tainting marks some initial memory locations, and tracks the data flow by tainting the memory that depends on the tainted set [JD05, AH09, AMC\(^+\)06]. It is often used to detect security attacks. LIFT [FCZea06] reduces the overhead by eliminating unnecessary dynamic information flow tracking aggressively, and running instrumented code as short as possible by efficiently switching between the target program and the instrumented code. Backtracking techniques are often used in intrusion analysis [SP03, ADKD11], and are aimed to detect the sequence of events that lead to certain consequences. When intrusion is detected, it traces back the system logs and tries to locate the origin of the intrusion.

Dynamic dependency analysis is not suitable for ATDetector because ATDetector does not need the information of whole program dependency. Furthermore, dependency analysis in parallel programs is even more complicated [RJRS08]. ATDetector cannot utilize memory tainting because it does not know the initial tainting locations to begin with. Compared with backtracking technique, ATDetector needs to trace back the program memory and register trace, instead of the system logs used for the intrusion detection purpose.

3.1.3 Overall Architecture

There are two possible approaches to achieve the above goal. One is dynamic and is done by analyzing run-time traces. The other approach is static, involving the analysis of source code. Even though the dynamic approach has more accurate information than the static method, it can incur large (up to 30X [LTQZ06]) run-time overhead to collect memory access traces. In addition, the number of ad hoc synchronizations that can be identified using this method would largely depend on the code coverage of test cases. Also some ad hoc synchronization loops may terminate
Figure 3.1: Ad hoc synchronization abstract model. The loop exit condition (i.e., sync condition) either directly or indirectly depends on a sync variable.

after only one iteration, making it hard to identify them as ad hoc synchronization loops [JT10a]. Due to these reasons, we choose the static method, i.e., analyzing source code.

The biggest challenge to automatically identify ad hoc synchronizations is how to separate them from computation loops. The diversity of ad hoc synchronizations makes it especially hard. To address the above challenge, we have to identify the common elements among various ad hoc synchronization implementations.

**Commonality among ad hoc synchronizations:** Interestingly, ad hoc synchronizations are all implemented using loops, referred to as *sync loops* (Figure 3.1). While a sync loop can have many exit conditions, at least one of them is the exit condition to be satisfied when an expected synchronization event happens. We refer to such exit conditions as *sync conditions*. The sync condition directly or indirectly depends on a certain shared variable (referred to as *a sync variable*) that is loop-invariant locally, and modified by a remote thread.

Note that a sync variable may not necessarily be directly used by a sync condition (e.g., inside a while loop condition). Instead, a sync condition may have data/control-dependency on it like in the examples shown on Figure 2.4 and Figure 2.1(a)(c).

Following the above characteristic, SyncFinder starts from loops in the target programs, and examines their exit conditions to identify those that are (1) loop invariant, (2) directly or indirectly depend on a shared variable, and (3) can be satisfied by a remote thread’s update to this variable. By checking these constraints, SyncFinder filters out most computation loops as shown in our evaluation.

Checking all of the above conditions requires SyncFinder to conduct (1) program analysis to know the exit conditions for each loop; (2) data and control flow analysis to know the dependencies of exit conditions; (3) some static thread analysis to conservatively identify what segment of code may run concurrently; and (4) some simple satisfiability analysis to check whether the remote
As shown on Figure 3.2, SyncFinder consists of the following steps: (1) Loop detection and exit condition extraction; (2) Exit dependent variable (EDV) identification; (3) Pruning computation and condvar loops based on characteristics of EDVs; (4) Synchronization pairing to pair an identified sync loop with a remote update that would break the program out of this sync loop; (5)
Final result reporting and annotation in the target program’s source code.

SyncFinder is built on top of the LLVM compiler infrastructure [LA04] since it provides several useful basic features that SyncFinder needs. LLVM’s intermediate representation (IR) is based on single static assignment (SSA) form, which automatically provides a compact definition-use graph and control flow graph for every function, both of which can be leveraged by SyncFinder’s data-, and control-flow analysis. In addition, SyncFinder also uses LLVM’s loopinfo analysis, alias analysis, and constant propagation tracking to implement the ad hoc sync loop identification algorithm. SyncFinder annotation is done via the static instrumentation interfaces provided in LLVM. In the rest of this section, we focus on our algorithms and do not go into details about the basic analysis provided by LLVM.

3.1.4 Contributions

Based on the insights from the characteristics study in Chapter 2, we invent SyncFinder, an automatic static analysis tool to identify and annotate ad hoc synchronizations in concurrent programs. This would be highly beneficial to programmers to be aware of them and thus convert them into standard synchronization primitives such as cond_wait. Concurrent program analysis tools (e.g., data race or deadlock detection tools, performance analysis tools) can also leverage the annotations of ad hoc synchronizations to improve their effectiveness.

Specifically, if ad hoc synchronizations are annotated in concurrent programs, (1) programmers can port such ad hoc synchronizations to more structured implementations; (2) static or dynamic concurrency bug (e.g., data race and deadlock detection tools, performance analysis tools) can also leverage the annotations of ad hoc synchronizations to improve their effectiveness.

This Chapter also presents two use cases with the annotations of ad hoc synchronizations from SyncFinder. The first case demonstrates a simple wait-inside-critical-section detector, which identifies deadlocks and bad programming practices involving ad hoc synchronizations. Our evaluation shows that the tool detects 5 deadlocks that are missed by previous deadlock detection tools in Apache, MySQL and Mozilla.

In the second use case, we extend a data race detector in Valgrind [NS07] to leverage the ad hoc synchronizations annotations from SyncFinder. With the annotations, Valgrind race detector shows 43–86% decrease of false positive rates of data race report. Above two use cases demonstrates that SyncFinder can help concurrency bug detectors improve their effectiveness by making the implicit synchronizations explicit to them.
### 3.2 Finding Loops

<table>
<thead>
<tr>
<th>Apps</th>
<th>while</th>
<th>for</th>
<th>goto</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>27</td>
<td>4</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>MySQL</td>
<td>33</td>
<td>24</td>
<td>26</td>
<td>83</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 3.1: Loop mechanisms used for real-world ad hoc synchronization. There are a non-negligible number of "goto" loops, which often complicate loop analysis (e.g., Figure 1.5).

As shown in Table 3.1, ad hoc synchronizations are implemented using three primary forms of loops: "while", "for" and "goto". Fortunately, LLVM’s loopinfo pass identifies all those loops based on back edges in LLVM IR.

For each loop identified by LLVM, SyncFinder extracts its exit conditions. Specifically, it identifies the basic blocks with at least one successor outside of the loop, then for each identified basic block, SyncFinder extracts its terminator instruction, from which SyncFinder can identify the branch conditions. Such conditions are the exit conditions for this loop. SyncFinder represents the exit conditions in a canonical form: disjunction (OR) of multiple conditions, and examines each separately.

In addition, since LLVM does not keep the loop context information, e.g., loop headers and bodies, across functions, SyncFinder keeps track of them into its own data structure and uses them throughout the analysis.

### 3.3 Identifying Sync Loops

The key challenge of SyncFinder is to differentiate sync loops from computation loops. To address this challenge, SyncFinder examines the exit conditions of each loop by going through the following steps to filter out computation loops.

#### 3.3.1 Exit Dependent Variable (EDV) Analysis

For each exit condition of each loop in the target program, the first step is to identify all variables that this exit condition depends on—we refer to them as exit dependent variables (EDVs). If a loop is a sync loop, the sync variables should be included in its EDVs. Note that a sync variable is not necessarily used in an exit condition (sync condition) directly. A loop exit condition can be data/control-dependent on a sync variable. Therefore, we conduct data-flow and control-
flow analysis to find indirect EDVs. The EDV identification process is similar to static backward slicing [Wei81, RR95, HRB88].

SyncFinder first starts from variables directly referenced in the exit condition. They are added into an EDV set. Then, as shown in Figure 3.3, it pops a variable out from the EDV set, and finds out new EDVs along this variable’s data/control flow. New EDVs are inserted into the set. It then pops another EDV from the set, and so on so forth until it reaches the loop boundary. For an EDV that does not depend on any other variables inside this loop, we refer to them as a leaf-EDV (similar to “live-in” variables). SyncFinder maintains a separate set for leaf-EDVs. Obviously, leaf-EDVs are the ones we should focus on since they are not derived from any other EDVs in this loop.

During the backward data/control flow tracking process, if the dependency analysis encounters a function whose return value or passed-by-reference arguments affect the loop exit condition, SyncFinder further tracks the dependency via inter-procedural analysis. SyncFinder applies data-and control-flow analysis starting from the function’s return value, and identifies Return/arguments-Dependent Variables (RDVs) in the callee. Such RDVs are also added into the leaf-EDV set. In addition, all RDVs of this function are stored in a summary to avoid analyzing this function again for other loops.

To handle variable and function pointer aliasing, SyncFinder leverages and extends LLVM’s alias analysis to allow it go beyond function boundary.

### 3.3.2 Pruning Computation Loops

For every exit condition of a loop, SyncFinder applies the following two pruning steps to check whether it is a sync condition. At the end, if a loop has at least one sync condition, it is identified as a sync loop. Otherwise, it is pruned out as a computation loop. Most computation loops are filtered in this phase.
Non-shared variable pruning: A sync variable should be a shared variable that can be set by a remote thread. Specifically, it should be either a global variable, a heap object, or a data object (even stack-based) that is passed to a function (e.g., thread starter function) called by another thread, which can be shared by the two threads.

Therefore, if an exit condition has no shared variables in its leaf-EDV set, it is deleted from the loop’s exit condition set. SyncFinder moves to the next exit condition of this loop. If the loop has no exit conditions left, this loop is pruned out as a computation loop.

Loop-variant based pruning: In almost all cases, a sync condition is loop-invariant locally, and only a remote thread changes the result of the sync condition. Based on this observation, SyncFinder prunes out those exit conditions that are loop-variant locally as shown on Figure 3.4. It is possible that some ad hoc synchronizations may also change the sync conditions locally. In all our experiments with 25 concurrent programs, we did not find any true ad hoc synchronizations that SyncFinder missed due to this pruner. Note that some exit conditions, such as expiration time, are separated as different conditions, and we examine each condition separately.

```
while(module){
    next = module->next;
    free(module);
    module = next;
} /* Mozilla */

for (i = 0; i < nlights; i++){
    VecMatMult(lp->pos, m, lp->pos);
    lp = lp->next;
} /* SPLASH */
```

(a) Loop-variant module
(b) Loop-variant condition checking

Figure 3.4: The non-sync variables pruned out by loop-variant based pruning. In the two computation loops, the variables in italic font are shared variable leaf-EDVs.

To check if an exit condition is loop variant, SyncFinder applies a modification (MOD) analysis within the scope of a loop being examined. Specifically, it checks all leaf-EDVs and leaf-RDVs of this loop, and prunes out those modified locally within this loop. The leaf-RDV summary is also updated accordingly.

### 3.3.3 Pruning Condvar Loops

SyncFinder does not consider condvar loops (i.e., sync loops that are associated with cond.wait primitives) as ad hoc loops as they can be easily recognized by intercepting or instrumenting these primitives. As the final step of the ad hoc sync loop identification, SyncFinder checks every loop candidate to see it calls a cond.wait primitive inside the loop. Loops that use primitives are recognized as condvar loops and are thereby pruned out. The names of cond.wait primitives (original pthread functions or wrappers) are provided as input to SyncFinder to identify cond.wait calls.
Table 3.2: The characteristics of writes to sync variables. In the four sampled applications, majority of writes assign constant values, or use simple increase or decrease operations.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>total</th>
<th>constant</th>
<th>inc/dec op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>42</td>
<td>21 (50.0%)</td>
<td>5 (11.9%)</td>
</tr>
<tr>
<td>MySQL</td>
<td>325</td>
<td>125 (38.5%)</td>
<td>110 (33.8%)</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>203</td>
<td>48 (23.6%)</td>
<td>8 (3.9%)</td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>83</td>
<td>41 (49.4%)</td>
<td>31 (37.3%)</td>
</tr>
</tbody>
</table>

3.4 Synchronization Pairing

Once we identify a potential sync loop, we find the remote update (referred to as a sync write) that would “release” (break) the wait loop. To identify a sync write, SyncFinder first collects all write instructions modifying sync variable candidates, and then applies the following pruning steps.

**Pruning unsatisfiable remote updates** For each remote update to the target sync variable candidate, SyncFinder analyzes what value is assigned to this variable, and whether it can satisfy the sync condition. A complicated solution to achieve this functionality is to use a SAT solver. But it is too heavyweight, especially since, according to our observations (shown in Table 3.2), the majority (66%) of sync writes either assign constant values to sync variables, or use simple counting operations like increment/decrement, rather than complicated computations. This is because a sync variable is usually a control variable (e.g. status, flag, etc.) and does not require sophisticated computations.

Therefore, instead of using a SAT solver, we use constant propagation to check if this remote update would satisfy the exit condition. For an assignment with a constant, it substitutes the variable with the constant, and propagates it till the exit condition to see if it is satisfiable or not. For increment based updates, SyncFinder treats it as “sync var > 0” since it obviously does not release the loop that is waiting for an exit condition “(sync var == 0)”.

**Pruning serial pairs** A sync loop and a sync write should be able to execute concurrently. If there is a happens-before relation between such pair, due to thread creation/join, barrier, etc, the remote write does not match with the sync loop. Due to the limitation of static analysis, currently SyncFinder conservatively prunes serial pairs related to only thread creation/join. Specifically, SyncFinder follows thread creation and conservatively estimates code that might be running concurrently.
3.5 SyncFinder Annotation

After the above pruning process, the remaining ones are identified as sync loops, along with their corresponding sync writes. All the results are stored in a file. SyncFinder also automatically annotates in the target software’s source code using LLVM static instrumentation framework. It inserts `/#SyncAnnotation: Sync Loop Begin(&loopId)`, `/#SyncAnnotation: Sync Loop End(&loopId)`, respectively, at the beginning and end of an identified sync loop. In addition, inside the loop, it also annotates the read to a sync variable by inserting `/#SyncAnnotation: Sync Read(&syncVar, &loopId)`. For the corresponding sync write, it inserts `/#SyncAnnotation: Sync Write(&syncVar, &loopId)`. The loopId is used to match a remote sync write with a sync loop. Similar annotations are also inserted into the target program’s bytecode to be leveraged by concurrency bug detection tools as discussed in the next section.

3.6 Two Use Cases of SyncFinder

SyncFinder’s auto-identification can be used by many bug detection tools, performance profiling tools, concurrency testing frameworks, program language designers, etc. We built two use cases to demonstrate its benefits.

3.6.1 A Tool to Detect Bad Practices

It is considered bad practice to wait inside a critical section, as it can easily introduce deadlocks like the Apache example shown on Figure 1.3 and the MySQL example on Figure 1.4. Furthermore, it can result in performance issues caused by cascading wait effects, and may introduce deadlocks in the future if programmers are not careful. As a demonstration, we built a simple detector (referred to as wait-inside-critical-section detector) to catch these cases leveraging SyncFinder’s auto-annotation of ad hoc synchronizations. Our detection algorithm can be easily integrated into any existing deadlock detection tool as well.

To detect such pattern, our simple detector checks every sync loop annotated by SyncFinder to see if it is performed while holding some locks. If a sync loop is holding a lock, then SyncFinder checks the remote sync write to see whether the write is performed after acquiring the same lock or after another ad hoc sync loop, so on and so forth, to see if it is possible to form a circle. If it is, the detector reports it as a potential issue: either a deadlock or at least a bad practice.
3.6.2 Extensions to Data Race Detection

We also extend Valgrind [NS07]’s dynamic data race detector to leverage SyncFinder’s automatic identification of ad hoc sync loops. Valgrind implements a happens-before algorithm [Lam78] using logical timestamps, which was originally based on conventional primitives including mostly lock primitives, and thread creation/join. It cannot recognize ad hoc synchronizations. As a result, it can introduce many false positives (shown in Table 3.7) as discussed in Chapter 2 and illustrated using two examples in Figure 2.8.

We extend Valgrind to eliminate data race false positives by considering ad hoc synchronizations annotated by SyncFinder. It treats the end of a sync loop in a similar way to a cond_wait operation, and the corresponding sync write like a signal operation. This way it keeps track of the happens-before relationship between them. We also extend Valgrind to not consider sync variable reads and writes as data races.

3.7 Evaluation

3.7.1 Effectiveness and Accuracy

We evaluated SyncFinder on 25 concurrent programs, including 12 used in our manually characteristic study and 13 other ones. Table 3.3 shows the overall result of SyncFinder on the 25 programs. On average SyncFinder accurately identifies 96% of ad hoc sync loops in the 12 studied programs and has a 6% false positive rate overall. SyncFinder successfully identified diverse ad hoc order synchronizations, including those we missed during our manual identification. For example, it successfully identifies those complicated, interlocked “goto” sync loops, as shown in Figure 1.5.

For the 12 studied programs, SyncFinder misses a few (1-3 per application) sync loops in large server/desktop applications. Considering the total number of loops (up to 4265) in each of these applications, such a small miss rate does not limit SyncFinder’s applicability to real world programs. SyncFinder fails to identify these sync loops because of the unavailability of the source code for these library functions and inaccurate pointer alias.

SyncFinder also returns a low number of false positives for all 25 programs. As showed in Table 3.3, SyncFinder has 0-6 false positives per program (i.e. a false positive rate of 0-30%). Such numbers are quite reasonable. Programmers can easily examine the reported sync loops to prune out those few false positives. Most of the false positives are caused by inaccurate function pointer analysis. Due to complicated function pointer alias, sometimes SyncFinder cannot further track into callee functions to check if a target variable (leaf-EDV) is locally modified. In these cases, SyncFinder conservatively considers the target variable as a sync variable.
<table>
<thead>
<tr>
<th>Apps.</th>
<th>Total loops</th>
<th>Identified Sync Loops</th>
<th>Missed ones</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>True</td>
<td>FP</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>1462</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>MySQL</td>
<td>4265</td>
<td>48</td>
<td>42</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>2044</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>Cherokee</td>
<td>748</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>AOLServer</td>
<td>496</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Nginx</td>
<td>705</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>BerkeleyDB</td>
<td>1006</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>BIND9</td>
<td>1372</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Desktop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mozilla-js</td>
<td>848</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>PBZip2</td>
<td>45</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Transmission</td>
<td>1114</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>HandBrake</td>
<td>551</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>p7zip</td>
<td>1594</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>wxDFast</td>
<td>154</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Scientific</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radiosity</td>
<td>80</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Barnes</td>
<td>88</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Water</td>
<td>84</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Ocean</td>
<td>339</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>FFT</td>
<td>57</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Cholesky</td>
<td>362</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>RayTracer</td>
<td>144</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>FMM</td>
<td>108</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Volrend</td>
<td>77</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>LU</td>
<td>38</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Radix</td>
<td>52</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Total (Ave.)</td>
<td>-</td>
<td>290</td>
<td>264</td>
</tr>
</tbody>
</table>

Table 3.3: Overall results of SyncFinder: *Every concurrent program uses ad hoc sync loops except LU*. Both true ad hoc sync loops and false positives are showed here. For the 12 programs used in the characteristic study, the numbers of missed ad hoc sync loops are also reported. They are generated by comparing with our manual checking results from the characteristic study. We cannot show the numbers of missed ad hoc sync loops for the unseen programs in the study since we did not manually examine them as we did for the 12 studied programs. To show SyncFinder’s total exploration space, we also show the total number of loops, most of which are computation loops. Note that the total numbers of ad hoc sync loops are different from those numbers shown in Table 2.2 because some code (for other platforms such as FreeBSD, etc) are not included during the compilation.

### 3.7.2 Sync Loop Identification and Pruning

To show the effectiveness of sync loop identification, in Table 3.4, we test SyncFinder on some server/desktop applications and show the results from each of the sync loop identification steps. 37
Table 3.4: EDV Analysis and non-sync variable pruning. After identifying leaf-EDVs for each loop, SyncFinder applies non-shared, loop-variant and condvar-loop based pruning schemes. The final results are the sync variables of the ad hoc sync loops. Some sync variables may be associated with a same sync loop.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1,462</td>
<td>3,120</td>
<td>8,682</td>
<td>184</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>MySQL</td>
<td>4,265</td>
<td>9,181</td>
<td>20,458</td>
<td>377</td>
<td>118</td>
<td>72</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>2,044</td>
<td>4,434</td>
<td>11,276</td>
<td>171</td>
<td>45</td>
<td>27</td>
</tr>
<tr>
<td>PBZip2</td>
<td>45</td>
<td>278</td>
<td>799</td>
<td>130</td>
<td>16</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3.5: False synchronization pair pruning. Note that the numbers shown here are synchronization pairs. In all the other results, we show “synchronization loops” (regardless how many setting statements for an ad hoc sync loop)

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Initial pairs</th>
<th>w/ Remote update pr.</th>
<th>w/ Serial pair pr.</th>
<th>With both</th>
<th>True pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>27</td>
<td>22</td>
<td>27</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>MySQL</td>
<td>251</td>
<td>204</td>
<td>178</td>
<td>141</td>
<td>123</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>168</td>
<td>134</td>
<td>146</td>
<td>115</td>
<td>96</td>
</tr>
<tr>
<td>PBZip2</td>
<td>19</td>
<td>15</td>
<td>11</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

From the total loops identified, SyncFinder extracts exit conditions, and identifies all leaf-EDVs (the third column in Table 3.4). From the leaf-EDVs, SyncFinder prunes out non-shared variables (95% of leaf-EDVs), and applies loop-variant based pruning, which further prunes 80% of shared leaf-EDVs. SyncFinder then applies the final pruning step to prune out sync variables that are associated with condvar loops. The remains are sync variable candidates and those loops using them are potential sync loops.

3.7.3 Synchronization Pairing and Pruning

During synchronization pairing, SyncFinder applies two pruning schemes, unsatisfiable remote update pruning and serial pair pruning. Table 3.5 shows the effect of those pruning steps on the same set of server/desktop applications in Table 3.4. First, remote update based pruning eliminates 51.8% of false sync pair candidates on average. It is especially effective on Apache, since the majority of sync writes are just simple assignments with constant values, so it is easy to determine whether such values would satisfy the corresponding sync exit conditions.

Second, the effectiveness of serial pair pruning depends on application characteristics. While it prunes out almost all false positives in simple desktop/scientific programs (e.g., PBZip2), it is less effective in servers like Apache, where many function pointers are used. Due to the limitation of
function pointer analysis, it is hard to know in all cases whether two certain regions cannot be con-
current. To be conservative, SyncFinder does not prune the pairs inside such regions. Fortunately,
the remote update based pruning helps filtering them out.

### 3.7.4 Two Use Cases: Bug Detection

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Deadlock (New)</th>
<th>Bad practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1 (0)</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>2 (2)</td>
<td>13</td>
</tr>
<tr>
<td>Mozilla</td>
<td>2 (0)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.6: Deadlock and bad practice detection

Table 3.6 shows that our simple deadlock detector (leveraging SyncFinder’s ad hoc synchronization annotation) detects five deadlocks involving ad hoc order synchronizations, including those shown in Figure 1.3 and Figure 1.4. Previous tools would fail to detect these bugs since they cannot recognize ad hoc synchronizations. Besides deadlocks, our detector also reports 16 bad practices, i.e. waiting in a sync loop while holding a lock, which could raise performance issues or cause future deadlocks.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Original Valgrind</th>
<th>Extended Valgrind</th>
<th>%Pruned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>30</td>
<td>17</td>
<td>43%</td>
</tr>
<tr>
<td>MySQL</td>
<td>25</td>
<td>10</td>
<td>60%</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>7</td>
<td>4</td>
<td>43%</td>
</tr>
<tr>
<td>Water</td>
<td>79</td>
<td>11</td>
<td>86%</td>
</tr>
</tbody>
</table>

Table 3.7: False positive reduction in Valgrind

Table 3.7 shows that SyncFinder auto-annotation could reduce the false positive rates of Valgrind data race detector by 43-86%.

### 3.8 Summary

This Chapter describes a first effort to solve the problems introduced by ad hoc synchronizations, a tool called SyncFinder to automatically detect and annotate ad hoc synchronizations in multithreaded programs. The observation of SyncFinder is that ad hoc synchronizations are implemented by incooperating shared variables in loop exit conditions which depend on setting operations on the variables from remote threads. Driven by this observation, SyncFinder uses static analysis
to identify ad hoc synchronization pairs and finally annotate them in the programs. By evaluating 25 multithreaded programs including 14 applications and 11 SPLASH-2 programs, SyncFinder successfully identifies 96% of ad hoc synchronization loops with a 6% false positive rate.

With the annotated ad hoc synchronizations, this Chapter further demonstrates that SyncFinder can help detect deadlocks missed by conventional deadlock detection and also reduce data race detector’s false positives. Many other concurrency program analysis tools can also leverage the annotations of ad hoc synchronizations to improve their effectiveness or efficiency.

In the next Chapter, I discuss the address transfer, a propose to reduce false positives in data race detectors caused by implicit synchronizations.
Chapter 4

Address Transfer

This Chapter presents Address Transfer. Concurrent programs are prone to bugs, such as data races. Recently much work has been devoted to detecting data races in multi-threaded programs. Most tools, however, require the accurate knowledge of synchronizations in the program, and may otherwise suffer from false positives in race detection, limiting their usability. To address this problem, some tools such as Intel® Inspector provide mechanisms for suppressing false positives and/or annotating synchronizations not automatically recognized by the tools. However, they require users’ input or even changes of the source code.

4.1 Overview

4.1.1 Motivation

Various solutions have been proposed to detect data races. They are mostly based on one of the two classic algorithms, the happens-before algorithm [SJ91, M.P06, MJ03, ADJ09, L.L78] and the lockset algorithm [SMG+97, DK03, PRY07], or a hybrid of the two [JKA+02, EA03b, YTW06]. The former claims two conflicting memory accesses as a data race if they are not ordered by synchronization operations, and is usually implemented as dynamic tools such as Intel® Inspector (hereinafter referred to as ”Inspector”) [Int11] and Valgrind [NS07], whereas the latter reports a data race if there is no common lock held to protect conflicting accesses to a shared memory location, and is implemented as both static [DK03, MAJ06, Ste93] and dynamic tools [SMG+97]. In order to overcome the large overhead associated with software-only implementations, hardware designs have also been proposed: hardware happens-before bug detectors [SJ91, M.P06, MJ03, ADJ09] leverage existing cache coherence protocol, and a lock-set data race detector [PRY07] makes use of the hardware bloom filters.

Unfortunately, most existing data race detectors, including even commercial strength race detectors such as Inspector, require accurate knowledge of synchronizations in the program, and may otherwise report false races (referred to as Not-A-Race in this dissertation for clarification) or benign races, of which the definitions are explained in Table 4.1. The former is not a race but is in-
### Class | Definition
--- | ---
**Not-A-Race** | It is NOT a true data race but reported by a data race detector. If the detector is dynamic, it is usually introduced by non-recognized happens-before relation guaranteed by a program (e.g., through synchronization).
Benign data race | It is a true data race satisfying the data race definition, but does not affect the program correctness.
Harmful data race | It satisfies the data race definition and results in incorrect program behavior once it is exposed.

* data race definition: more than one threads access the same memory location without proper synchronization, and at least one of them is a write.

Table 4.1: The cases reported by existing data race detectors. Besides harmful data races, many existing tools report both Not-A-Race and benign data races as well.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Total</th>
<th>Not-A-Race</th>
<th>Benign Race</th>
<th>Harmful Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>592</td>
<td>555</td>
<td>30</td>
<td>7</td>
</tr>
<tr>
<td>Berkeley-DB</td>
<td>686</td>
<td>601</td>
<td>65</td>
<td>20</td>
</tr>
<tr>
<td>Ocean</td>
<td>116</td>
<td>94</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>Asterisk</td>
<td>29</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Qt</td>
<td>30</td>
<td>26</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1453</strong></td>
<td><strong>1289</strong></td>
<td><strong>133</strong></td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

Table 4.2: The accuracy of Inspector. Inspector provides mechanisms to leverage users’ input for annotating synchronizations not automatically recognized or suppressing false positives, but we didn’t use them in our experiments since domain knowledge or extra effort from the users are required. For classification, we manually checked the cases reported by Inspector. Since it is not always clear whether a reported race is benign or harmful due to the lack of program knowledge, we conservatively count benign races only when it is certain that both possible orders of two race instructions will produce exactly the same execution state, following the idea of an existing automatic benign data race classifier [SZJ+07].

Correctly reported when a detector fails to recognize certain happens-before relations in a program. For example, in Figure 4.1(a), the address of data is transferred from thread 1 to thread 2 and then used by thread 2 to conduct a task written in data. Because of the causal relationship between S2 and the address transfer process, the two shared accesses at S1 and S2 would never race with each other, yet may be reported as a race if a detector fails to recognize the implicit happens-before relation. On the other hand, a benign race is indeed a data race but is not a bug, i.e. the data race is intended or allowed as shown in the example in Figure 4.1(b).

Table 4.2 shows the results from Inspector by using 5 real-world applications including Apache, Berkeley-DB, etc. It shows that the accuracy in detecting harmful races is only around 2%. Inter-
Figure 4.1: Real-world cases reported by the Intel® Inspector [Int11]. The highlighted memory accesses are reported as data races. Although (a) is not a data race, and (b) does not affect execution correctness, both are reported.
estingly, Not-A-Race is the most dominant source of the inaccuracy, accounting for 36%–94% of the total reports. In order to separate harmful data races from Not-A-Race reports and benign races, developers have to manually check all the reports, which is tedious and very time consuming. As a result, the wide use of data race detectors has been limited in practice [AKBea10].

To improve the accuracy of data race detection, there have been two main approaches: separating benign races from harmful ones [SZJ+07, JMSO10], and relying on manual annotation for synchronizations [NS07, Int11, san, suna]. In the first approach, [SZJ+07] identifies benign data races by trying to flip the execution order of a race pair and comparing the two execution results. [JMSO10] uses heuristics to identify certain code patterns that may introduce benign data races. Since their target is limited to benign data races, these methods are not very helpful in identifying Not-A-Race among the reports. Some tools including Inspector adopt the second approach by providing mechanisms to leverage users’ input to annotate synchronizations that are not automatically recognized. However the annotation can be time consuming and may need source code change which requires the developers to put extra effort on it. Recently a few solutions [XPZ+10b, JT10b] have been proposed to automatically identify ad hoc synchronizations (e.g., a while-flag). They can help eliminate benign data races used in ad hoc synchronization such as the one shown in Figure 4.1(b), and some of Not-A-Race caused by the ad hoc synchronization. However, our study reveals that such type of Not-A-Race is not the dominant factor (details in Table 4.3 in Section 4.2.1).

4.1.2 Summary of Contributions

This chapter first studies why an existing data race detector generates Not-A-Race reports. Our study makes a unique observation of address transfer, which contributes to a majority of Not-A-Race reports. Based on this observation, the second part of our study builds a tool called ATDe-tector to effectively identify address transfer related to data race candidates and eliminate them from the data race reports. As a result, we improve the accuracy of Inspector by eliminating all Not-A-Race reports caused by unrecognized address transfer.

Unique Observation of Address Transfer: In multi-threaded applications, there are two common ways for multiple threads to access shared data. The memory address may be globally known to all threads from the beginning, and thus they can immediately access the data with its own knowledge of the location. Alternatively, the location to access may be made known to one thread by another at a certain execution point. In this case, one thread explicitly passes a memory address to another thread to give the location information. We refer to this as address transfer. We further identified three use cases of address transfer: (i) Task dispatching, (ii) Memory recycling, and (iii) Producer-consumer communication for data sharing. Their details are in Section 4.2.2.
Figure 4.2: The real-world examples of address transfer. The addresses transferred are highlighted. In addition to these two, another use case of address transfer is task dispatching, which is shown in Figure 4.1(a). Due to the happens-before relation introduced by address transfer, S1 and S2 never race each other.
Address transfer implicitly introduces a happens-before relation between the sender’s and the receiver’s memory accesses to the transferred address. For instance, in all the three examples in Figure 4.1(a) and Figure 4.2, the memory access at S1 by thread 1 always happens before the same memory access at S2 by thread 2. Since S2 uses the received address either directly or indirectly, it causally depends on the address transfer process. Again, the address transfer can happen only after S1.

Importantly, however, existing data race detectors, including commercial strength tools such as Inspector, may not be aware of the happens-before relations introduced by address transfer. It is because address transfer is often implemented with application-specific methods, such as lock-free data structure (e.g., the Apache example in Figure 1.2(a)), customized locks (e.g., the Berkley-DB example in Figure 1.2(b)), system calls (e.g., TransmissionBT example in Figure 4.1(a)), and other synchronization primitives recently developed such as transactional memory (e.g., Intruder example in Figure 1.2(c)). In general, existing data race detectors recognize only standardized synchronization primitives (e.g., pthread_mutex_lock). Consequently, they incorrectly report the pairs of S1 and S2 in all the examples as data races, while all of them are Not-A-Race.

Specifically, according to our investigation of the results from Inspector with 17 diverse open-source applications, the majority of Not-A-Race reports (62% of 1420) are caused by the unrecognized address transfer (the details are shown in Table 4.3).

Address Transfer Detection (ATDetector): Based on the observation of address transfer, we propose a method called ATDetector that automatically identifies address transfer and uses the information to prune Not-A-Race reports. ATDetector focuses on identifying only those address transfers that are related to data race candidates collected by data race detectors.

Given a data race candidate and its memory address, starting from the later race instruction candidate, ATDetector backtracks all the relevant address propagation paths within a thread and also across multiple threads. Meanwhile, it collects all potential address receiving and sending sites, which can be shared memory read/write or certain system calls. In this way, we can recognize address transfer regardless of the customized way of implementation. Finally, if one of the address transfer candidates indeed breaks the potential race condition by enforcing happens-before relation, the data race candidate is pruned out.

To implement this idea, ATDetector conducts binary instrumentation with Pin [CRRea05] to trace memory and register accesses, and performs online dependency analysis with the traces when it encounters potential race candidates.

Improving the accuracy of Inspector, a state-of-the-art commercial data race detector: We used ATDetector to improve the accuracy of Inspector by eliminating Not-A-Race reports caused by unrecognized address transfer without any users’ input. Since the source code of Inspector is not available, instead of integrating ATDetector to the tool, we run it by feeding Inspector’s data
race reports to ATDetector as inputs.

We evaluated ATDetector with 8 diverse open-source server, desktop, and scientific programs, including Berkeley-DB, Apache, TransmissionBT, etc. As a result, ATDetector can help eliminate all of the Not-A-Race reports caused by unrecognized address transfer, which are 62% of the total Not-A-Race reports from Inspector. More importantly, ATDetector does not eliminate any true data races that are originally reported. It has a modest memory overhead ranging from 2KB to 2MB per thread for most of the evaluated applications.

4.2 Address Transfer

This section studies the reasons an existing data race detector introduces Not-A-Race reports. Our study makes a unique observation of memory address passing among multiple threads, which is referred to as address transfer. The observation includes (i) unrecognized address transfer is the major cause of Not-A-Race reports (Section 4.2.1); (ii) address transfer is commonly used for task dispatching, memory recycling, and producer-consumer communications (Section 4.2.2); (iii) address transfer is implemented with various customized methods, such as lock-free structure (Section 4.2.3).

Our study is conducted by using Intel® Inspector [Int11], and 17 open-source multi-threaded programs. To understand the Not-A-Race reports and address transfer, we manually examined a total of 1775 cases reported by Inspector, including 1420 Not-A-Race reports.

4.2.1 Causes of Not-A-Race Reports

As shown in Table 4.2, majority (88%) of the cases reported by Inspector are Not-A-Race reports. This problem becomes more severe in some large server applications such as Apache and Berkeley-DB. Therefore, it is critical to eliminate Not-A-Race reports to improve the accuracy of data race detection.

Table 4.3 further shows the three causes of Not-A-Race reports:

(1) **Address transfer:** Instead of being a global variable, data can alternatively be shared by passing its address from one thread to another. We refer to the process of passing memory address as address transfer. For example, in Figure 4.1(a), thread 1 dispatches a new task to thread 2 by transferring the address of data, which describes a new task. Other uses of address transfer are explained in Section 4.2.2.

Similar to lock synchronization, address transfer implicitly introduces a happens-before relation between the address sending thread’s and the receiving thread’s accesses to the same memory location, which is mistakenly reported as a data race. It guarantees that the sender’s memory access
happens before the receiver’s access to the same memory location. The address of the memory accessed may be either exactly the same as the transferred address, or calculated from the transferred address, as shown in Figure 4.1(a). In this example, S1 is guaranteed to happen before S2 due to the data’s address transfer, and the accessed memory address (address of data->session) is calculated from the transferred address (address of data).

Unfortunately, most of the address transfers examined are implemented with various customized methods (details are in Section 4.2.3), and existing data race detectors including Inspector mainly focus on standardized synchronization primitives such as POSIX pthread_mutex_lock, in order to recognize happens-before relation or/and lock-set. Because of this, it results in many (62% of 1420) Not-A-Race reports, as shown in Table 4.3.

(2) Customized Synchronization: Programmers often implement their own methods of synchronization instead of using the ones provided by the standard libraries. It can be due to many reasons such as performance or convenience [XPZ+10b]. This is especially true in server applications. The synchronization can be a customized lock or an ad-hoc synchronization such as a while loop waiting for an event from the remote site (e.g., the while loop in Figure 4.1(b) waits until session->event is set).

<table>
<thead>
<tr>
<th>Apps.</th>
<th>Not-A-Race</th>
<th>Address Transfer</th>
<th>Customized Synchronization</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley-DB</td>
<td>601</td>
<td>272</td>
<td>329</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>555</td>
<td>555</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Squid</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Asterisk</td>
<td>13</td>
<td>3</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>FreeSwitch</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>64</td>
<td>27</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>HandBrake</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>TransmissionBT</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Qt</td>
<td>26</td>
<td>0</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>WxdFast</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>intruder</td>
<td>14</td>
<td>8</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>bayes</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>genome</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>labyrinth</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>yada</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Ocean</td>
<td>94</td>
<td>0</td>
<td>94</td>
<td>0</td>
</tr>
<tr>
<td>Radix</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1420</strong></td>
<td><strong>883(62.2%)</strong></td>
<td><strong>533(37.5%)</strong></td>
<td><strong>4(0.3%)</strong></td>
</tr>
</tbody>
</table>

Table 4.3: The causes of Not-A-Race reports from Intel Inspector.
Typical data race detectors only recognize standard primitives such as those from POSIX, and cannot deal with customized synchronizations without user annotations. This is the same in Inspector, and it results in 37.5% of Not-A-Race reports. A few previous works including our own [XPZ+10b] deal with the problem of ad-hoc synchronizations. However, it is less significant compared with Address Transfer and is not the focus of this chapter.

(3) **Others**: For conditional shared data access, a special type of order synchronization can be implemented with a flag and an if-statement checking it. Specifically, it guarantees that certain memory accesses happen only if another access to the same memory from a different thread already happened. Otherwise it skips the memory access without waiting for the remote peer. Inspector fails to recognize such condition checking and generates a few (0.3% of 1420) Not-A-Race reports.

### 4.2.2 Use Cases of Address Transfer

In general, there are two ways for a thread to get the memory address of shared data. First, a memory address can be global (e.g., global variables) so that every thread has access to them during its lifetime. In this case, a thread can immediately access the data without communicating with other threads. Alternatively, the addresses of shared data may only be made known to one thread by another at a specific execution point of the program. For example, a thread may produce data for other threads to consume by explicitly passing the location information of the data first. Similarly, address transfer can be used to pass the address of a memory pool that is freed by one thread and later acquired by another thread.

Specifically, from Not-A-Race reports of Inspector, we observed the following three major use cases of address transfer in the 17 real-world multi-threaded programs.

(1) **Task dispatching**: Especially in multi-threaded server applications, it is common for one thread to create tasks and dispatch them to other threads to execute. The address of a task structure can be either directly transferred to a specific worker thread or passed through a shared task queue. The TransmissionBT example in Figure 4.1(a) shows that thread 1 directly dispatches the task produced to thread 2 via transferring the address of the task structure, data.

(2) **Customized memory recycling**: Many applications implement their own memory allocators for efficiency. It is possible for a thread to reuse a chunk of memory just returned by another thread to a shared memory pool. In the example in Figure 4.2(b), after thread 1 ends its task using mpool, the address of mpool is inserted to a global queue. Then thread 2 reuses the pool by fetching the address from the queue.

(3) **Producer-consumer communications for data sharing**: In some cases, multi-threaded programs need to dynamically manage shared objects by adjusting the data structure when encountering new program states or input, and the shared objects are passed in a producer-consumer
manner. For instance, in the Berkeley-DB example in Figure 4.2(a), all active cursors are organized as B-Tree by using index field. Thread 1 creates the database cursor dbc and inserts its address into a shared queue. Thread 2 iterates all the active cursors and adjusts index to re-organize the B-Tree. The address of index is derived from the address of dbc transferred via the shared queue.

4.2.3 Address Transfer Implementation

Address transfer needs a synchronized communication between the address sender and the receiver, as other data sharing. However, according to our investigation of the data race reports, in many cases (62% of all Not-A-Race reports in Table 4.3), address transfer is implemented without standard synchronization primitives such as pthread_mutex_lock and thus becomes problematic for existing data race detectors. Specifically, the followings discuss the diverse customized methods implementing address transfer, with real-world examples in Figure 1.2.

(1) Lock-free implementation: Programmers often do address transfer with their own lock-free structures. One popular way is to use atomic read and write operations with a shared queue. For example, Figure 1.2(a) shows Apache’s address transfer for memory recycling. By using the gcc’s builtin function for atomic compare and swap, thread 1 posts an address of available memory location p to recycled_p, and thread 2 reuses it by reading the address atomically.

Note that while atomic operations can be used to implement synchronizations, they are not synchronizations themselves and can be used for other purposes. Therefore it is hard to automatically detect lock-free structures, though the atomic operations themselves are not difficult to identify.

(2) Customized synchronizations: Some synchronization primitives used for address transfer are customized ones that are specifically implemented, and are difficult for typical data race detectors to recognize [XPZ+10b] without users’ inputs. The Berkeley-DB example in Figure 1.2(b) shows that dbc’s address is transferred via a shared queue which is protected by a customized lock primitive, _db_tas_lock, and thus cannot be recognized.

(3) New parallel programming models: With the recent development in diverse concurrent programming models such as transactional memory [UT10, COD+07] and Intel Thread Building Blocks [Inta], new synchronization primitives are used to implement address transfer. Figure 1.2(c) shows the implementation with transactional memory.

(4) System calls: Address transfer can also be done using certain types of system calls such as read() and write(). The TransmissionBT example in Figure 4.1(a) uses the read/write system calls to transfer the address of data via a pipe. Without being aware of such system calls, data race detectors may fail to recognize the related address transfer.
4.3 Address Transfer Detection

4.3.1 Overview

Since address transfer severely affects the accuracy of data race detectors, it would be beneficial to identify them. Manually annotating address transfer is tedious and error-prone, as discussed in the beginning of the chapter. Therefore, we develop a tool called ATDetector to dynamically identify the address transfer related to data race candidates, and eliminate the Not-A-Race reports. It can be integrated with a data race detector to filter out Not-A-Race reports automatically. Alternatively, as a stand-alone detector, it can also take data race candidates as input, and validates them by running the application and monitoring its execution.

As we have observed from real-world address transfer cases in Section 4.2.3, an address transfer can be implemented in various ways, e.g., using lock-free structure, which is not easy to automatically identify. Therefore, ATDetector does not try to identify such synchronizations used to implement address transfer. Instead, it tries to detect the dependency between the memory address involved in a race candidate and the transferred address. ATDetector performs backtracking[SP03, ADKD11] on all relevant address propagation paths both intra- and inter-thread, trying to locate
the source of the accessed memory address. Once the address transfer is identified along the path, the data race candidate is determined to be Not-A-Race and pruned. The backtracking process is shown in Figure 4.3. This approach makes the identification independent of how address transfer is synchronized.

However, it is not trivial to explore address propagation paths due to the following reasons. First, the memory address used in data race candidates may not be exactly the same as the address transferred. The address accessed could be calculated from the address transferred with other values. Second, address transfers may be implemented using system calls as well as shared memory reads/writes. Third, address transfers can be performed across more than two threads.

To address the above challenges, ATDetector conducts the followings: (i) binary analysis to extract memory address, registers in an instruction, and dependency between the operands; (ii) based on the extracted information, building traces for both memory and registers; (iii) dependency analysis on the traces in order to backtrack from instr2 to the address receiving site (Figure 4.3); (iv) leveraging the semantic knowledge of certain system calls, library calls, and instructions that affect the propagation path of the address.

Figure 4.4 shows the detection steps. ATDetector first identifies all the possible candidates of address receiving sites (i.e., ATr set) using the dependency analysis explained above. Then, for each ATr, the tool finds its remote peer (i.e., ATs), which is an address sending site that either writes to the same shared memory or is a system call corresponding to ATr. Finally, it checks whether the ATs happens later than instr1 on the same thread. If not, ATDetector moves on to check another ATr from the ATr set. Otherwise, a valid address transfer is identified and the inspected data race candidate is determined to be Not-A-Race. In addition, to handle the case where more than two threads are involved in the address transfer, we recursively perform the similar steps by treating an intermediate ATs as the second race (inst2 in Figure 4.3) instruction candidate and begins a new detection from it.

In order to compare the timing of address transfer related events, ATDetector maintains a global timestamp that ticks at each shared memory access as well as each invocation of potential address transfer related system calls such as read and write.

4.3.2 Address Receiving Site Identification

In this step, ATDetector tries to find all possible address receiving sites that a given memory address depends on. Figure 4.5 describes the process of address receiving site identification. The input of this process is the memory access instruction that contains the target memory address. In the first iteration, the input is the second instruction involved in the candidate data race pair. ATDetector then extracts the relevant register to be tracked. The memory access instruction usually
Figure 4.5: The process of address receiving site identification. Based on the memory and register trace, ATDetector backtracks the address propagation path in order to locate potential address receiving sites. When an ATr is output, it continues to track from the ATr in order to locate further possible receiving sites. The termination conditions are discussed in Section 4.3.2.

utilizes two registers, one for memory address calculation, referred to as memory base register, the other for storing the value written or read from the address. For the first iteration, we get the memory base register as it is the starting point of our analysis. For the later instructions on the tracking path, we extract the register used in the source operand in order to backtrack the propagation path by following the dependency.

After the register is identified, ATDetector searches backward along the register trace and locates the last change of this register. Then it decides the next step based on the source of register modification. The source can be of several types:

Local variable: the simplest case is a register changed by reading from a local variable, via which the memory address is propagated. In this case, ATDetector further identifies the last change to this local variable by searching backward the memory access trace. The newly identified change record contains the register which holds the value to be written to the local variable, and the register becomes the next relevant register and participates in the next iteration.

Shared memory: if the source is shared memory, then it is already a candidate address receiving site, and we put it to the output list. However, the tracking process does not terminate here because we need to find all candidate address receiving sites. To do this, ATDetector extracts the memory base register of this instruction and continues the local tracking.

Others: there can be other sources of registers along the local tracking process, such as library calls and arithmetic computation. ATDetector addresses each of them specifically as follows:

Library calls and system calls affect the detection in two ways: they are actually used to perform the transfer as the example in Figure 4.1(a), or they are in the address flow paths. ATDetector deals with these function calls as long as it knows their semantics. For example, if the function can be a data receiver such as ”read”, it outputs the call sites as ATr. On the other hand, if the function does no communication but just passes on the data between the arguments and return value, such as memcpy, ATDetector continues the local tracking by attaining the source arguments. Semantics of more library and system calls can be added to ATDetector on demand.
Arithmetic computations may be also used to generate memory addresses, especially in the case of arrays or complex data structures. For example, the LEA and ADD instructions are often used in this case. ATDetector deals with them by forking the tracking path according to the operands, and tracks each of the source operands.

The backtracking process terminates when (1) a valid address transfer that orders the two instructions of a given data race candidate is identified, or (2) no more record can be found from the trace, or (3) the backtracking encounters certain function calls, or (4) the record has a timestamp smaller than the first access in the race candidate. Among them, conditions (1) and (2) are intuitive. For (3), some function calls can also terminate the backtracking such as malloc and read, since their return values do not depend on previous data flow. And for (4), this is because it is impossible to locate a valid address transfer if the current instruction already happened before the first instruction in the race candidate.

Note that in the backward tracking based on dynamic execution trace, inter-procedure analysis is guaranteed automatically. This is because ATDetector observes the actual execution sequence of instructions, and it does not need to care about whether the modifications are made inside some routines or not.

### 4.3.3 Address Sending Site Identification

After the candidate address receiving sites are identified, ATDetector attempts to find the corresponding address sending site for each receiving site. The challenge of this process comes from the wide variety of address transfer implementations. If the address receiving site is a shared memory read, ATDetector simply checks the memory trace and looks for the last memory write to the shared memory. However, if the candidate receiving site is the read end of a system call, ATDetector needs to consult the knowledge base to get the semantics of the functions involved in the transfer. For example, if the receiving site is a call to read() that reads a previously created pipe, as in Figure 4.1(a), ATDetector has to pair it with the corresponding write() call which is the sending site candidate. Based on the semantics of pipe, it first finds the matching file descriptors that form the pipe, and then instead of finding the latest writing site, ATDetector searches the execution trace and finds the writing end based on the syscall invocation order.
4.3.4 Address Transfer Candidate Validation

Once the candidate address sending site is also identified, ATDetector checks if the pair enforces a happens-before relation between the candidate race instructions. Specifically, it checks (i) whether the first access involved in the data race report has a smaller timestamp (happens earlier) than the identified address sending site, and (ii) both of the first access and the address sending site are from the same thread. These two conditions guarantee that the first access happens before the address transfer. Since the backtracking step explained in earlier sections naturally reveals that the second access happens after the address transfer, satisfying the two conditions above finally guarantees that the given two accesses have happens-before relation, so it is a Not-A-Race pair.

If the first condition is not satisfied, ATDetector drops the address transfer candidate. If the second one is not satisfied, the candidate sending site is either in the local thread where the backtracking started, or another thread. ATDetector needs to continue to backtrack from the identified candidate sending site in both cases. In the first case, the tracking is still on the same thread and needs to continue looking for further possible address transferring sites. In the second case, our observation is that address may be transferred between more than 2 threads. Figure 4.6 shows such an example, where tracking continues in the listener thread.

4.3.5 Implementation

ATDetector is implemented using Pin[CRRea05]. Pin provides a comprehensive runtime inspection platform to analyze a program’s behavior, and plenty of APIs to inspect each instruction. ATDetector records the necessary traces by instrumenting memory accessing instructions as well as instructions and system/library calls that could affect the address propagation, as described in Section 4.3.2 and 4.3.3. ATDetector leverages the instruction inspection APIs to analyze binary code. However, it is difficult to accurately decide whether a memory is shared or not. Therefore ATDetector conservatively treats all heap memory as potentially shared. Heap memory can be recognized by checking if the address falls into thread stack address range.

Since we do not have the source code of Inspector and cannot directly integrate ATDetector with it, we use its data race report as the input. The program runs under ATDetector and when a candidate data race is triggered, ATDetector starts the address transfer detection. If an address transfer is detected, the candidate race is classified as a Not-A-Race report.

In order to traceback the execution and to find how the memory address propagates, ATDetector records every instruction that may lead to data passing. A typical data race detector keeps the record of memory accesses, which can be optionally utilized by ATDetector when tracing back the memory modifications. However, a unique additional requirement in ATDetector is the trace of the register modifications. For example, ATDetector needs to record instructions that affect the
address propagation path, such as MOV, ADD, and LEA, etc. Besides memory and register trace, ATDetector also needs to record the relevant system calls and library calls as described earlier.

The trace is maintained in a per-thread manner to reduce the contention. As the program runs, the trace may grow larger than the memory capacity especially when the trace size is not limited. ATDetector periodically flushes the traces to disks, and fetches them when needed during address transfer detection.

### 4.3.6 Memory Usage

A critical concern for tools that perform runtime analysis is the memory overhead. If too much memory is consumed by the tool itself, the target application may suffer from extremely low performance (frequent page-swaps). This is especially true when it needs to record the execution trace, which is usually very large and needs to be minimized or compressed for high performance [SP03, MHC10, XR04].

An important observation of address transfer is that the address receiving site and the second memory access show strong spatial locality with regard to the instructions. For example, in the Berkeley-DB code shown in Figure 4.2(a), the program immediately accesses the data after taking from the list. Other examples in Figure 4.1(a), Figure 4.2, and Figure 1.2 share the same characteristic. Based on this observation, we can limit the amount of memory for recording the register trace from the address receiving site to the memory accessing instruction without significant accuracy loss. ATDetector allows users to specify the amount of memory for recording, and the sensitivity of the trade-off between memory consumption and detection effectiveness is shown in Section 4.4.3.

Note that loops may hurt the memory efficiency by producing large amount of memory accessing records. Currently ATDetector detects and skips simple spin loops. Complex loops can also be detected by building the control flow graph in the pre-execution phase [JT10b]. We leave it to the future work.

<table>
<thead>
<tr>
<th>Apps.</th>
<th>LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>248K</td>
<td>Apache web server</td>
</tr>
<tr>
<td>Berkeley-DB</td>
<td>388K</td>
<td>Database library</td>
</tr>
<tr>
<td>Asterisk</td>
<td>530K</td>
<td>Telephony server</td>
</tr>
<tr>
<td>TransmissionBT</td>
<td>96K</td>
<td>Bit-torrent cliente</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>20K</td>
<td>Memory allocator</td>
</tr>
<tr>
<td>intruder-STAMP</td>
<td>1.3K</td>
<td>Network intrusion detector</td>
</tr>
<tr>
<td>genome-STAMP</td>
<td>1.3K</td>
<td>Gene sequencing algorithm</td>
</tr>
<tr>
<td>labyrinth-STAMP</td>
<td>1K</td>
<td>Maze routing algorithm</td>
</tr>
</tbody>
</table>

Table 4.4: Applications used in the evaluation
Table 4.5: Effectiveness of ATDetector in pruning out Inspector’s Not-A-Race reports related to address transfer. Note that the number of such reports from Inspector is different from Table 4.3 (i.e., “Address Transfer” column), since some data race candidates reported by Inspector are not even appeared in ATDetector’s run.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Not-A-Race related to Address Transfer</th>
<th>True Race</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reported by Inspector</td>
<td>Pruned by ATDetector</td>
</tr>
<tr>
<td>Apache</td>
<td>531</td>
<td>531</td>
</tr>
<tr>
<td>Berkeley-DB</td>
<td>247</td>
<td>247</td>
</tr>
<tr>
<td>Asterisk</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>TransmissionBT</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>intruder</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>genome</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>827</strong></td>
<td><strong>827(100%)</strong></td>
</tr>
</tbody>
</table>

Table 4.6: The size of register traces per thread. It shows ATDetector’s memory overhead when it prunes all Not-A-Race reports caused by address transfer.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Trace size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1.6MB</td>
</tr>
<tr>
<td>Berkeley-DB</td>
<td>40MB</td>
</tr>
<tr>
<td>Asterisk</td>
<td>2KB</td>
</tr>
<tr>
<td>TransmissionBT</td>
<td>2MB</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>8KB</td>
</tr>
<tr>
<td>intruder</td>
<td>2KB</td>
</tr>
<tr>
<td>genome</td>
<td>2KB</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2KB</td>
</tr>
</tbody>
</table>

Currently ATDetector uses the same-sized trace buffer for each thread. However, different threads may require different amount of space for trace, depending on their tasks. Even though our evaluation results show that ATDetector has moderate memory overhead, using simple optimization techniques like dynamically adapting buffer size will help further reduce the overhead.

4.4 Evaluation

4.4.1 Experimental Methodology

In our experiments, we evaluate the accuracy of ATDetector using Inspector’s data race reports on 8 real-world applications including 3 server applications, 2 desktop application, and 3 scientific
benchmarks using transactional memory from STAMP [MCKO08]. Table 4.4 shows the description of each application. Since the source code of Inspector is not available for us, it is not possible to directly integrate our proposed idea to Inspector. Therefore, we first run Inspector alone to get data race reports, then separately run ATDetector with the data race candidates as its inputs. The application inputs are the same in both phases.

### 4.4.2 Effectiveness of ATDetector

Table 4.5 shows the effectiveness of ATDetector in pruning Not-A-Race reports that are caused by unrecognized address transfer. First, ATDetector successfully eliminated all of the Not-A-Race reports from Inspector, as long as the data race candidates are encountered during ATDetector’s standalone execution. Please note that, due to some timing issues, the data race candidates encountered during Inspector’s detection may not all appear again in ATDetector’s runs. It is not an issue originated from ATDetector, and can be avoided by integrating our idea directly into a data race detector, or by dynamically enforcing the target data race candidates to execute during a standalone run, just like many testing tools [PLZ09, K.S08]. Second, ATDetector eliminates no true data races that are originally reported by Inspector.

The above results indicate that ATDetector is highly accurate to identify address transfer, and is thus practical to improve widely-used commercial data race detectors.

### 4.4.3 Memory Overhead

The memory overhead may be concerned especially due to tracing register writes, as discussed in Section 4.3.5. Table 4.6 shows the maximum size of register trace per thread. It shows that the size varies from 2KB to 40MB, depending on applications. But except for Berkeley-DB, most applications use modest amount of memory during the entire detection run.

Since some applications such as Berkeley-DB require a large amount of trace buffers, the size of memory used for trace should be limited and is better to be adjustable by users. To further study the sensitivity of the algorithm to the size of register trace, we measure ATDetector’s pruning rates by changing the size of register trace buffer with Apache and Berkeley-DB, which are the top two applications showing high number of Not-A-Races related to address transfer.

As the result shown in Figure 4.7, even with the small size of trace buffers, ATDetector can show high pruning effectiveness. In the case of Apache, with only a 160KB trace buffer per thread, ATDetector can still prune around 50% of the total targets. The pruning rate eventually reaches 98% on 800KB, and then 100% on 1.6MB, which are not shown in this graph due to space limit. In the case of Berkeley-DB, the pruning rate increases even faster at the beginning: 90% of the
Figure 4.7: The sensitivity to the size of register trace buffer. Due to the space limit, we are not showing all x-axis. At the maximum trace size shown in Table 4.6, the graph will eventually reach 100%, meaning that ATDetector can eliminate all the Not-A-Race reports related to address transfer.
total targets are eliminated only with 100KB trace buffer per thread.

This results reveal that there exists a strong spatial locality of the instructions while backtracking to address transfer sites. It is intuitive in that, after a thread receives a memory address, it usually accesses the memory right away, as we discussed in Section 4.3.6.

Nevertheless, there are some extreme cases where ATDetector’s backtracking requires a large register trace buffer. For instance, once an Apache thread receives the address of memory pool, long time later, the thread can do a final access to the memory to cleanup the memory pool, happening to be a race candidate. To identify the address transfer for memory recycling, ATDetector needs to keep all the register traces collected for the long duration.

### 4.4.4 Output to Aid Inspection

For each eliminated Not-A-Race report, ATDetector outputs the detailed information in order to help users understand why the pruned pair is not a race, and how the memory address is passed from one thread to the other. Figure 4.8 shows the ATDetector’s output for one Not-A-Race report prunned on Berkeley-DB.

### 4.4.5 Result Discussion

As an automatic tool, ATDetector efficiently helps avoid human mistakes in reasoning about data races, especially for complicated multi-thread programs. In our study, there are several cases where we have mistakenly determined as true races because of some “common senses” such as “if one object is the last one accessed in a critical section, it is very likely to be protected by the surrounding lock [DK03]”. There is one case in Berkeley-DB where the write to a global variable
is the last statement in a critical region, and the read is not protected. Therefore we identify it as a true data race. During the experiment, however, ATDetector reports all of its dynamic instances as Not-A-Race. By checking the output as Figure 4.8, we find it is actually well ordered by address transfer. And the lock surrounding the write is meant to protect other fields of the same data structure. Without knowing this, one may fix the problem by adding a lock to the read, and thus degrade the software performance by adding potential contention.

Similar to all the dynamic program analysis tools, ATDetector analyzes the program behavior based on a specific execution, and does not consider the different behaviors in the others. Therefore it is possible that some instances of a candidate data race are ordered by address transfer but others are not. An instance is the recycled data structure may contain pointers to the same globally shared variable which might be accessed simultaneously by multiple threads and thus forms true race. This is a well-known limitation of happens-before relation based analysis, and would not be prevented even if the programmers manually and correctly annotate the synchronization points. In order not to prune true races, ATDetector only reports address transfer if all the dynamic instances of a candidate race in the execution can be ordered. Our experiments show that there are some true races that can be ordered by address transfer in certain instances. However none of them has all the instances detected as Not-A-Race in our study.

### 4.5 Future Work

Chapter 6 presents detailed future work of the dissertation. This section describes the future work related to address transfer.

As the next steps of address transfer, there are mainly three further work we can do. First, for more practical use, we can integrate the idea of ATDetector into a data race detector and help suppress Not-A-Race reports directly in the race detection process. Second, we could always do a more comprehensive characteristics study for address transfer cases in concurrent programs. With this part, we plan to investigate more diverse address transfer cases from other applications. Third, triggered by the high spatial locality related to address transfer and the address use, we plan to see if small hardware support for event tracing can help fast on-the-fly detection.

### 4.6 Summary

In this Chapter, we studied false data race reports (Not-A-Race reports) from a state-of-the-art commercial data race detector, Intel® Inspector, with 17 applications from various fields. By manually examining all the 1420 Not-A-Race reports, we identified two major sources of Not-A-Race
reports: *address transfer* and customized synchronizations. Particularly, the happens-before relations introduced by address transfer are often unrecognized during data race detection due to its customized way of implementation, causing many (62%) Not-A-Race reports. To address this problem, we developed an automatic tool ATDetector to identify address transfer, and use it to prune Not-A-Race reports. We evaluated ATDetector with Inspector using 8 real-world applications including server, client/desktop, and scientific applications. With modest memory overhead, ATDetector could successfully prune all Not-A-Race reports caused by address transfer, without eliminating any true data races originally reported by Inspector.

In the next Chapter, I will discuss the previous work related to this dissertation.
Chapter 5

Background and Related Works

This Chapter describes previous works that are related to this dissertation. Section 5.1 discusses synchronizations in concurrent systems; Section 5.2 discusses existing concurrent program analysis tools; Section 5.3 discusses program analysis background; Section 5.4 describes other characteristic studies on synchronizations; and Section 5.5 further discusses other related work.

5.1 Synchronization

The synchronization in this dissertation is mainly referred to as synchronization in multi-threaded or multi-processes environments to guarantee no two threads or processes execute same portion of code at the same time. For different purposes in concurrent execution environments, there are different types of synchronizations.

5.1.1 Mutex lock

Mutex lock is a synchronization mechanism to enforce that at one period of time there is only one process or thread access the shared resource [Dijer]. This enforcement is also called mutual exclusion. Mutex lock usually relies on hardware supporting instructions like “test-and-set” or “compare-and-swap” for efficient implementation. Not using locks to protect shared resources in a multi-thread or -process environment can result in racing conditions between different threads, causing programs to generate incorrect results during execution. Improper use of locks can also cause deadlock or livelock problem, in which cases multiple threads or processes are blocked by or waiting for each other without making progress in the programs. The popular pthread implementation for mutex lock can be referred to from [IG04a].

5.1.2 Waiting and Signaling

Besides mutual exclusion, there is also another mechanism multithreaded or multiprocess programs need, i.e., waiting for some specific conditions to be met before continuing execution. Also
there is a way for threads or processes to signal other threads or processes that such conditions have been met so that they can resume their executions. Monitors proposed in [Hoa74, Han75] provide above functionality. The solution provided in monitors that implements waiting and signaling is called condition variables. A condition variable is associated with a group of threads, each of which can wait for a condition to become true before proceeding. While any other threads can also signal the waiting threads to notify them that the condition has become true, so that they can continue executing. There are also many other important works on monitors, such as [How76, LR80] The widely used pthread_cond_wait and pthread_cond_signal can be referred to from [IG04b, IG04c].

5.1.3 Semaphore

A semaphore [Dij68] is an abstract data type that provides a system level abstraction to coordinate operations on shared resources from multiple processes. Semaphore can also be used to guarantee mutual exclusion but provides more functionality than that. A semaphore can provide a counting number of resources to be shared by multiple processes, instead of just only one shared access in a mutual exclusion case. Semaphore is widely used in a variety of operating systems. The primitives of semaphore include P(or wait()) and V(also known as signal()). Operation P decrements the semaphore, and pause a process if reaching 0, while V increases the semaphore to give access to other processes.

5.2 Concurrent Program Analysis Tools

5.2.1 Concurrency Bug Detection

As discussed in 5.1, improper use of synchronizations introduces bugs in concurrent programs, like data races or deadlocks.

Data race happens when multiple threads or processes access a shared memory/resources at the same time, and one of the accesses is a write operation. While data race is because of missing proper synchronizations, deadlock happens when there are improper synchronizations in the programs to cause threads or processes wait for each other to release the synchronization resources(e.g., locks).

Much research has been conducted on concurrency bug detection [WKK\textsuperscript{+}08, JTZC08, MQB\textsuperscript{+}08, BFM\textsuperscript{+}05a, Intb, EA03a, Sunb]. These tools usually assume that they can recognize all synchronizations in target programs [SSSD06, DK03, LELS05a]. Data race detection has been under intensive study for a long time, and there are a lot of work using both software and hardware to perform the detection [DK03, SMG\textsuperscript{+}97, Int11, PRY07, ADJ09, SJ91]. In order to improve the
accuracy of the data race detectors, some tools require users’ annotation in order to annotate the synchronizations in the programs [Int11, NS07, san, suna], and some others classify benign data races from harmful data races [SZJ+07, JMSO10].

RaceFuzzer [K.S08] uses the reported data race information as a heuristic to test “error-prone” schedules in multi-threaded programs. It produces the scenarios of real data races by controlling the scheduling randomly. However it suffers from pruning large amount of real races (as much as 50%) as the schedules randomly produced by their scheme cannot cover all bug-triggering interleaving especially when the data race set is large.

There is another type of concurrency bugs called atomicity violation. Atomicity bugs are often introduced when programmers assume atomic code regions, while in the real implementation the guarantee failed and leads to run-time failure. Many works have been done to improve atomicity property in concurrent programs [FQ03, SAWS05, FM06, LTQZ06, FF04, PLZ09, FQ03, vPG03, KRDV07, HDVT08]. Same with other race detectors and deadlock detectors, a common limitation of these works is that they require programmers to specify all synchronizations because it is hard for those tools to identify all of them, especially implicit synchronizations.

As we demonstrated using deadlock detection and race detection with ad hoc synchronization annotation, SyncFinder can help these tools improve their effectiveness and accuracy by automatically annotating ad hoc synchronizations that are hard for them to recognize.

### 5.2.2 Spin and Hang Detection

Some recent work has been proposed in detecting simple spinning-based synchronizations [NSKI05, LLS06, JT10a]. For example, [LLS06] proposed some new hardware buffers to detect spinning loops on-the-fly. [JT10a] also provides similar capability but does it in software. Both can detect only simple spinning loops, i.e. those sync loops with only one single exit condition and also directly depend on sync variables (referred to as “sc-dir” in Table 2.3 in Chapter 2). As shown in Table 2.3 such simple spinning loops account for less than 16% of ad hoc sync loops on average in server/desktop applications we studied.

Besides, both of them are dynamic approaches and thereby suffer from the coverage limitation of all dynamic approaches (discussed in Chapter 3). In contrast, SyncFinder uses a static approach and can detect various types of ad hoc synchronizations. Additionally, we also conduct an ad hoc synchronization characteristic study.
5.2.3 Concurrent Program Testing and Failure Diagnosis

There are many related work focusing on in-house testing to expose concurrency bugs before software release. Some of the work rely on exercising data flow relationships in the program [YSP98, HM92]. And some others try to explore all kinds of possible interleavings in the concurrent programs [TLK92, KT96, LC06, HcTIH95] to expose bugs, such as data races. There are also some recent work which provide more practical interleaving testing [LW05, BFM±05b, Sto02, MQ07, MQBB08, LCKK07]. These works, however, can not recognize implicit synchronizations in their tools, limiting their usability. As a complementary tool, SyncFinder’s annotations of ad hoc synchronizations can help them identify those implicit synchronizations and improve their effectiveness.

Despite much research effort on bug detections and testing, concurrency bugs still escape to production. Therefore there are also many work related to concurrency failure diagnosis after the bugs manifested during production run. The diagnosis process often starts with reproduce of concurrency bugs, such as those deterministic replay works [KDC05, SKAZ04, VMw, PXY±09, BS95, WCG04, CS98, MQBB08]. The successful replay is highly dependent on the recording of the executions during run-time. In some of the record and replay systems(e.g., [PXY±09]), implicit synchronizations can be a headache and affect the efficiency of the replay dramatically. Because the record and replay system does not recognize implicit synchronizations, quite some time could be spend on record and replay ad hoc synchronizations which is not necessary. With the annotation from SyncFinder, those unnecessary replay could be eliminated thus improve the efficiency and effectiveness a lot.

5.2.4 Synchronization Annotation

Many annotation languages [sal, loc, jav, Ste93] have been proposed for synchronizations in concurrent programs. Unfortunately, annotation is not frequently used by programmers since it is tedious. SyncFinder is complementary to these work by providing automatic annotation for ad hoc synchronizations.

5.3 Program Dependency Analysis

Several works exist for program dependency analysis such as dynamic dependency profiling, memory tainting, and backtracking. Dynamic dependency profiling [XR04] builds the program’s dependency graph by analyzing the execution traces, and is widely used to automatically parallelize sequential code. SD³ [MHC10] accelerates this process and reduces the memory overhead by
parallelizing the dependency profiling steps and compressing the traces with stride patterns. Memory tainting marks some initial memory locations, and tracks the data flow by tainting the memory that depends on the tainted set [JD05, AH09, AMC+06]. It is often used to detect security attacks. LIFT [FCZea06] reduces the overhead by eliminating unnecessary dynamic information flow tracking aggressively, and running instrumented code as short as possible by efficiently switching between the target program and the instrumented code. Backtracking techniques are often used in intrusion analysis [SP03, ADKD11], and are aimed to detect the sequence of events that lead to certain consequences. When intrusion is detected, it traces back the system logs and tries to locate the origin of the intrusion.

Dynamic dependency analysis is not suitable for ATDetector because ATDetector does not need the information of whole program dependency. Furthermore, dependency analysis in parallel programs is even more complicated [RJRS08]. ATDetector cannot utilize memory tainting because it does not know the initial tainting locations to begin with. Compared with backtracking technique, ATDetector needs to trace back the program memory and register trace, instead of the system logs used for the intrusion detection purpose.

5.4 Software Bug Characteristics Studies

Several studies have been conducted on software bug characteristics [CYC+01, SC92, CC00, Z.06, LJZ07, YMZ+11]. These previous works identified insightful findings and provide inspiring guidelines for following research work to follow and solve the reliability problems in software systems.

Some of the empirical studies also collect(e.g., [CC00, FNU03]) concurrent bugs. [LPSZ08] is a first comprehensive study mainly on characteristics of concurrent bugs. The study in this dissertation is different from those studies by focusing on ad hoc synchronizations instead of bugs, even though many of them are prone to introducing bugs. The purpose of the ad hoc synchronizations study in this dissertation is to raise the awareness of ad hoc synchronizations, and to warn programmers to avoid them when possible. Also we developed an effective way to automatically identify those ad hoc synchronizations in large software.
5.5 Other Related Work

5.5.1 Lock-free and Wait-free Algorithm

Lock causes blocking of threads or processes. This introduces overhead from synchronization between threads and processes. Sometimes the overhead is not necessary because contentions for the shared resources may not happen all the time, actually in many cases the collision is quite rare. On the other hand, improper uses of locks can also lead to deadlocks or livelocks.

A lock-free or wait-free algorithm relaxes threads or processes from being blocked from competing and waiting for shared resources. There are several wait-free or lock-free research and implementations [Her88, Her91, MS96, KP12]. A lock-free algorithm guarantees system wide throughput, i.e., some thread could be blocked but the whole software system is not. Wait-free algorithms put stronger guarantee on execution progress. A wait-free algorithm provides thread/process starvation-freedom as well as system wide progress. A wait-free algorithm also guarantees lock-free property. This property is especially critical for real-time systems. However, these algorithms suffer from performance problems.

As discussed in Chapter 4, address transfer is often implemented with application specific synchronizations such as lock-free data structure (e.g., the Apache example in Figure 1.2(a)). It causes false positives to data race detectors because they cannot recognize this type of synchronizations. Our ATDetector thus detects address transfer instead of recognizing those implicit synchronizations to prune out false positives.

5.5.2 Concurrent Programming Language Design

To combat with concurrency bugs, researchers also try to propose better programming language supports to make it easier to write concurrent programs and less error-prone. Various transactional memory designs have been proposed to solve the programmability issues related to mutexes [RHP+07, AAK+05, RH, HM93, MTC+07, ST95, KCH+06, HF03, JB08, TH05, DDS06, ATLM+06, PDN06, YN09] and also condition variables [DM]. Transactional memory aims to provide an easier way of concurrent programming. Programmers only need to specify which code regions should be atomic but do not need to reason about when to put synchronization primitives like mutex locks. The implementation of transactional memory (either by hardware or software) automatically protects the atomicity of critical sections in the code. Our study complements such work by providing ad hoc synchronization characteristics in real world applications.
Chapter 6
Conclusions and Future Work

The dissertation starts with the characteristics of implicit synchronizations, and finds out that those synchronizations are either error prone or introduce difficulty to make the multi-threaded program correct. To address the issues from implicit synchronizations, the dissertation makes two other contributions, including 1) detection and annotation of ad hoc synchronizations, and 2) improving the false positive rate caused by implicit synchronizations in a commercial data race detector by proposing an address transfer happens before relation.

For the characteristics study of ad hoc synchronizations, we examined 229 ad hoc synchronization loops from 12 concurrent programs, and found several interesting and alarming characteristics. Among them, the most important results include: all concurrent programs have used ad hoc synchronizations and their implementations are very diverse and hard to recognize manually. Moreover, a large percentage (22-67%) of ad hoc loops in these applications have introduced bugs or performance issues. They also greatly impact the accuracy and effectiveness of bug detection and performance profiling tools.

In an effort to solve the problems introduced by these ad hoc synchronizations, we developed SyncFinder to automatically detect and annotate ad hoc synchronizations in multithreaded programs. The observation of SyncFinder is that ad hoc synchronizations are implemented by incorporating shared variables in loop exit conditions which depend on setting operations on the variables from remote threads. Driven by this observation, SyncFinder uses static analysis to identify ad hoc synchronization pairs and finally annotate them in the programs. By evaluating 25 multithreaded programs including 14 applications and 11 SPLASH-2 programs, SyncFinder successfully identifies 96% of ad hoc synchronization loops with a 6% false positive rate.

With the annotated ad hoc synchronizations, SyncFinder helps detect deadlocks missed by conventional deadlock detection and also reduce data race detector’s false positives. Many other tools and research projects can also benefit from SyncFinder. For example, concurrency testing tools (e.g., CHESS [MQB+08]) can leverage SyncFinder’s auto-annotation to force a context switch inside an ad hoc sync loop to expose concurrency bugs. Similarly, performance tools can be extended to profile ad hoc synchronization behavior.

Not only ad hoc synchronizations cause problems to multithreaded program reliability, other
customized synchronizations also introduce difficulty to the effectiveness of concurrent program analysis tools. This dissertation further studies false data race reports (Not-A-Race reports) from a state-of-the-art commercial data race detector, Intel® Inspector, with 17 applications from various fields. By manually examining all the 1420 Not-A-Race reports, we identified two major sources of Not-A-Race reports: address transfer and customized synchronizations. Particularly, the happens-before relations introduced by address transfer are often unrecognized during data race detection due to its customized way of implementation, causing many (62%) Not-A-Race reports. To address this problem, in this dissertation we develop an automatic tool ATDetector to identify address transfer, and use it to prune Not-A-Race reports. By evaluating ATDetector with Inspector using 8 real-world applications including server, client/desktop, and scientific applications, AT-Detector could successfully prune all Not-A-Race reports caused by address transfer with modest memory overhead, and without eliminating any true data races originally reported by Inspector.

All work has limitations, and ours is no exception. Some of the limitations open new directions for future work.

For ad hoc synchronizations, firstly SyncFinder requires source code. However, this may not significantly limit SyncFinder’s applicability since it is more likely to be used by programmers instead of end users. Secondly due to some implementation issues, SyncFinder still misses 1-3 ad hoc synchronizations. Eliminating them would require further enhancement to some of our analysis (such as alias analysis, etc.) Thirdly even though SyncFinder’s false positive rates are quite low, for some use cases that are sensitive to false positives, programmers would need to manually examine the identified ad hoc synchronization or leverage some execution synthesis tools like ESD [ZC10] to help identify false positives. And finally, for our characteristic study, we can always study a few more applications, especially of different types.

As we can see from the previous discussions in the dissertation that the ad hoc synchronizations are often used to enforce particular event orders in concurrent programs. Until now, much concurrency work has been conducted on one type of synchronization, atomicity. However, another major type of synchronization, order synchronization, has significantly been under-addressed. With the study on ad hoc synchronizations, this dissertation for the first time reveals that these order related synchronizations are common in concurrent programs and need improvements because they are often error-prone. More efforts on order synchronizations, e.g., ad hoc synchronizations could be put into research to improve them. We also expect that with the detection and annotation of ad hoc synchronizations, new performance tools and debugging tools can be studied to capture issues related to ad hoc synchronizations;

Along with the contribution of address transfer, we plan to extend it in three ways. First, for more practical use, we plan to integrate our idea into a data race detector and help suppress Not-A-Race reports directly in the race detection process. Second, for more comprehensive characteristic
study, we plan to investigate more diverse address transfer cases from other applications. Third, triggered by the high spatial locality related to address transfer and the address use, we plan to see if small hardware support for event tracing can help fast on-the-fly detection.

One open question we can ask for ad hoc synchronizations is *why do developers use them in practice?* Based on our study on real world comments from developers, easy to use and also performance are the most provided reasons. While they were not aware of the harmfulness of ad hoc synchronizations. And one broader question we can ask is can we design better synchronizations to fulfill developers’ need for ad hoc synchronizations? A more fundamental work should be done on ad hoc synchronizations is to design better synchronization primitives or libraries that are easy to use and give better performance.
References


