Ad Hoc Synchronization Considered Harmful

*Weiwei Xiong†, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma**

*University of California, San Diego †University of Illinois at Urbana-Champaign *Intel*

Abstract

Many synchronizations in existing multi-threaded programs are implemented in an ad hoc way. The first part of this paper does a comprehensive characteristic study of ad hoc synchronizations in concurrent programs. 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 part of our work builds 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.

We also build 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%.

1 Introduction

Synchronization plays an important role in concurrent programs. Recently, partially due to realization of multi-core 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) [37, 13, 30, 40] as ways to replace the cumbersome "lock" operations. Similar to TM, some new language constructs [46, 7, 12] such as Atomizer [12] have also been proposed to address the atomicity problem. On a different but related note, various tools such as AVIO [27], CHESS [31], CTrigger [36], ConTest [6] 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 [14, 16, 22].

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 ad hoc ways to do synchronizations. It is usually hard to tell ad hoc synchroniztions 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 *ad hoc synchronization*. If a program defines its own synchronization primitives as functional calls and then uses these functions throughout the program for synchronization, then we do not consider these primitives as ad hoc, since they are well modularized.

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.

Figure 1(a)(b)(c)(d) show four real world examples of ad hoc synchronizations from MySQL, Mozilla, and OpenLDAP. In each example, a thread is waiting for some other threads by repetitively checking on one or more shared variables, i.e. sync variables. Each case has its own specific implementation, and it is also not obviously apparent that a thread is synchronizing with another thread.

Unfortunately, there have been few studies on ad hoc

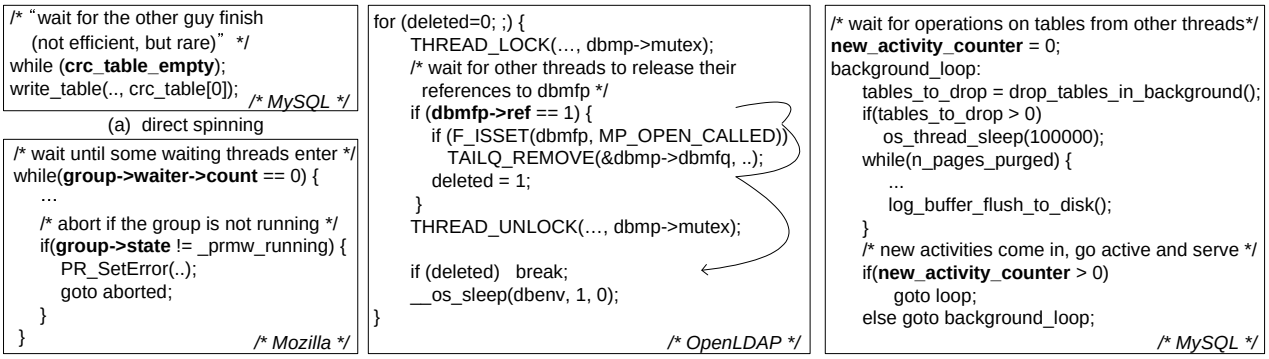


Figure 1: **Real world examples of ad hoc synchronizations**. Sync variables are highlighted using bold fonts. Example (a) directly spins on the sync variable; (b) checks more than one sync variables, (c) takes a certain control path to exit after checking a sync variable, (d) performs some useful work inside the waiting loop.

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 Contribution 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. The fact that programmers often use ad hoc synchronization is likely due to two primary reasons: (i) Unlike typical atomicity synchronization, when coordinating execution order among threads, 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 Section 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 1(a) directly spins on a shared variable; Figure 1(b) has multiple exit conditions; Figure 1(c) shows the exit condition indirectly depends on the sync variable and needs complicated calculation to determine whether to exit the loop; Figure 1(d) 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 Section 2.

Apps.	#ad hoc sync	#buggy sync
Apache	33	7 (22%)
OpenLDAP	15	10 (67%)
Cherokee	6	3 (50%)
Mozilla-js	17	5 (30%)
Transmission	13	8 (62%)

Table 1: **Percentages of ad hoc synchronizations that had introduced bugs according to the bugzilla databases and changelogs of the applications.**

(3) Ad hoc synchronizations are error-prone. Table 1 shows that among the five software systems we studied, signficant 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 2, 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 3 shows another deadlock example in MySQL, which has never been reported previously. More details and the real world examples are in Section 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 [11, 43, 24]. These tools cannot recognize ad hoc synchronizations unless these synchronizations are annotated manually by programmers or automatically by our SyncFinder described in section 1.2. For the same reason, it is also hard for concurrency testing tools such as ConTest [6] to expose these deadlock bugs during testing.

Furthermore, ad hoc synchronizations also have problems interacting with modern hardware's weak memory

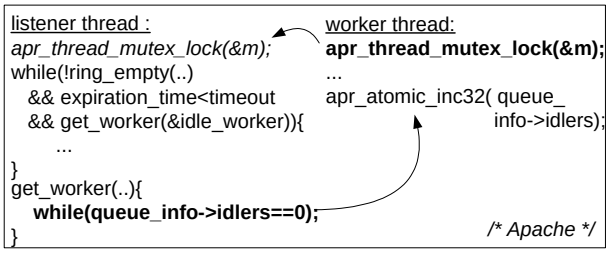


Figure 2: **A deadlock introduced by an ad hoc synchronization in Apache.**

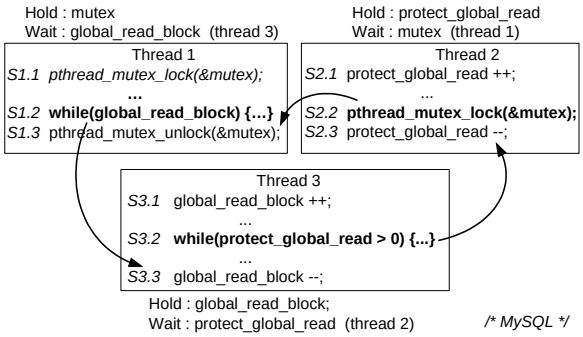


Figure 3: **A deadlock caused by a circular wait among three threads (This is a new deadlock detected by our deadlock detector leveraging SyncFinder's auto-annotation)**. Thread 2 is waiting at S2.2 for the lock to be released by thread 1; thread 1 is waiting at S1.2 for thread 3 to decrease the counter at S3.3; and thread 3 is waiting at S3.2 for thread 2 to decrease another counter at S2.3.

consistency model and also with some compiler optimizations, e.g. loop invariant hoisting (discussed further in Section 2).

By studying the comments associated with ad hoc synchronizations, we found that some programmers 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 Section 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 Contribution 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 [9] and SafeDrive [50] 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' Lock_Lint [2] have been available for several years, they are rarely used, even in Sun's own code [35]. Furthermore, manual examination is also error-prone. Figure 4 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.

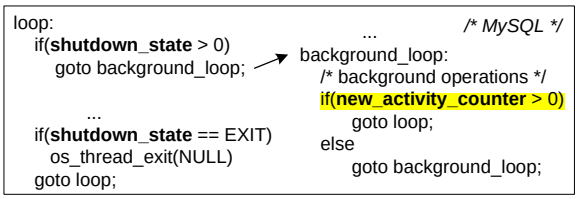


Figure 4: **An ad hoc synchronization missed in our manual identification process of our characteristic study but is identified by our auto-identification tool, SyncFinder**. The interlocked "goto" loops can easily be missed by manual identification (Figure 1(d) shows more detailed code).

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 characteristic study and 13 others. 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 iden-

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

Table 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.

tify 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 [33] 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.

2 Ad Hoc Synchronization Characteristics

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. Two inspectors separately investigated almost every line of source code and compared the results with each other. As shown on Table 3, 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 4). Fortunately, our automatic identification tool, SyncFinder, discovers them, and we were able to extend our manual examination to include such complicated types.

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.

Apps.	#sync loops	I_a	I_b	both
Apache	33	4	2	2
MySQL	83	12	8	7
OpenLDAP	15	3	3	2
PBZip2	7	1	0	0

Table 3: **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 1(d).

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++.

Finding 1: 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, 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 ensurance 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.

Apps.	Total loops	Total Ad hoc	Single exit condition					Multiple exit cond.			Total func	async
			sc -dir	sc -df	sc -cf	sc -func	total	mc -all	mc -Nall	total		
Apache	1462	33	4	0	1	3	8	22	3	25	16	25
MySQL	4265	83	23	5	4	11	43	13	27	40	32	64
OpenLDAP	2044	15	2	0	0	2	4	4	7	11	9	15
Cherokee	748	6	0	2	0	1	3	0	3	3	1	5
Mozilla-js	848	17	2	4	1	4	10	4	1	5	5	15
PBZip2	45	7	0	0	0	1	1	0	6	6	7	7
Transmission	1114	13	6	0	0	1	7	0	6	6	3	2
Radiosity	80	12	5	5	1	0	11	1	0	1	0	1
Barnes	88	7	6	1	0	0	7	0	0	0	0	0
Water	84	9	9	0	0	0	9	0	0	0	0	0
Ocean	339	20	20	0	0	0	20	0	0	0	0	0
FFT	57	7	7	0	0	0	7	0	0	0	0	0

sc : single exit cond.
-dir: directly depends on a sync var
-df : has data dependency
-cf : has control dependency

mc : multiple exit cond.
-all: all exit conditions depend on sync vars
-Nall : not all, but at least one does

func : inter-procedural dependency
async: useful work while waiting

Table 4: **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*).

```
while(1) {
    int oldcount = (global->barrier).count;
    ...
    If(updatedcount == oldcount)  break;
}                                  /* SPLASH2 */
```

(a) sc-df (data dependency)

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

(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;
}
```

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

(c) Function call

Figure 5: **Examples of various ad hoc synchronizations.** A sync variable is highlighted using a bold font. An arrow shows the dependency relation from a sync variable to a loop-exit condition. The examples of other ad hoc categories are shown on Figure 1.

Such performance justifications are frequently mentioned in programmers' comments associated with ad hoc synchronization implementations.

While ad hoc synchronizations are seemly 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.

Finding 2: Ad hoc synchronization is diverse.
Table 4 further categorizes ad hoc synchronizations from several perspectives. Some real world examples for each category can be found in Figure 1 and Figure 5.

(i) Single vs. multiple exit conditions: 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 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 1(a). 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 5(a)), control dependencies (referred to as *cf*, (Figure 1(c)), even inter-procedural dependencies (referred to as *func*, Figure 5(c)).

(iii) Asynchronous synchronizations (referred 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 1(d), a MySQL master thread does background tasks like log flushing until a new SQL query arrives (by checking *new_activity_counter*).

[1] A condition that can break the execution out of a loop.

```
/* get tuple Id of a table */
do {
  ret= m_skip_auto_increment ?
       readAutoIncrementValue(...):
       getAutoIncrementValue(···);
} while(ret== -1 && --retries && ..);
                                /* MySQL */
```

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

Figure 6: **An ad hoc synchronization in MySQL was revised by programmers to solve a performance problem.**

Finding 3: Ad hoc synchronizations can easily introduce bugs or performance issues.
After studying the 5 applications listed in Table 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 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 uses 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 2 and 3), 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 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 [5, 28, 45]. These modern microprocessors can reorder two writes to different locations, making ad hoc synchronizations such as the one in Figure 1(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 [5, 28, 45].

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 [3]. All of these just complicate programming as well as software testing and debugging.

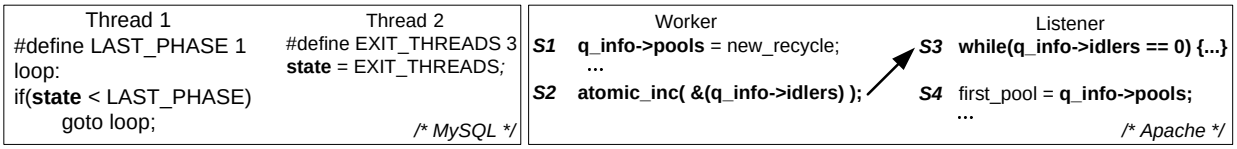
Comment examples
Programmers are aware of better design but still use ad hoc implementation (8%)
/* This can be built in *smarter way, like pthread_cond*, **but we do it** since the status can come from.. */ /* By doing.. applications will get *better performance and avoid the problem entirely*. **Regardless,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

Table 5: **Observations in programmers' comments on ad hoc synchronization from Apache, Mozilla, and MySQL**. We study 63 comments associated with ad hoc synchronizations.

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 5, 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.

Finding 4: 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 2 and 3).

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*



(a) Benign data race on **state**

(b) False data race on **q_info->pools**

Figure 7: **False positives in Valgrind data race detection due to ad hoc synchronizations.**

data races on sync variables: typically an ad hoc synchronization is implemented via an intended data race on sync variables. Figure 7(a) shows such a benign data race reported by Valgrind [33] 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 7(b), the two threads are synchronized at S2 and S3, which guarantees the correct order between S1 and S4's accesses to *q_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 [25, 32] and performance profiling [29] 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.

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 8 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.

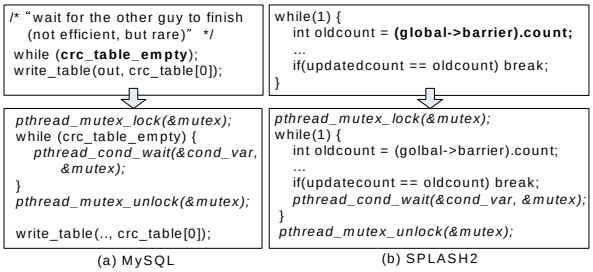


Figure 8: **Replacing ad hoc synchronizations with synchronization primitives using condition variables.** (a) shows the re-implementation of ad hoc synchronization in Figure 1(a); (b) is for Figure 5(a).

3 Ad hoc Synchronization Identification

3.1 Overview

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, the second part of our work builds 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 Section 1.2.

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 [27]) 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 after only one iteration, making it hard to identify them as ad hoc synchronization loops [18]. 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 9). 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 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 condi-

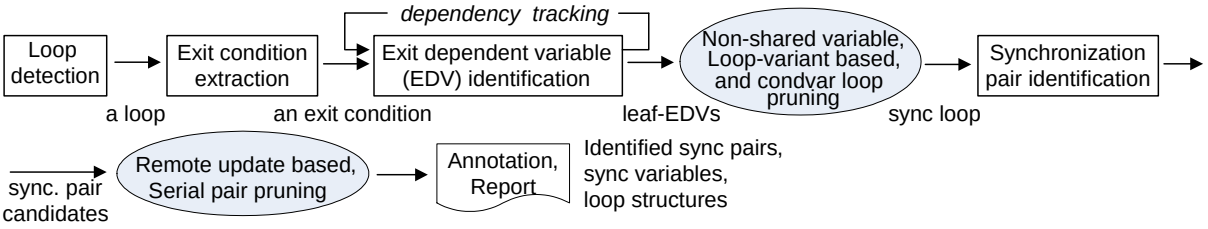


Figure 10: **SyncFinder design to automatically identify and annotate ad hoc synchronization**

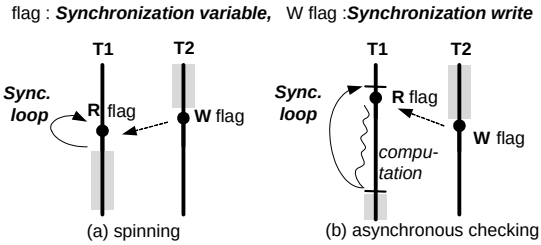


Figure 9: **Ad hoc synchronization abstract model.** The loop exit condition (i.e., sync condition) either directly or indirectly depends on a sync variable.

tion). Instead, a sync condition may have data/control-dependency on it like in the examples shown on Figure 1(c) and Figure 5(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 update to the sync variable can satisfy the sync condition.

As shown on Figure 10, 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 [23] 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.

3.2 Finding Loops

Apps.	while	for	goto	Total
Apache	27	4	2	33
MySQL	33	24	26	83
OpenLDAP	7	4	4	15
Mozilla-js	12	4	1	17

Table 6: **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 4).

As shown in Table 6, 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.

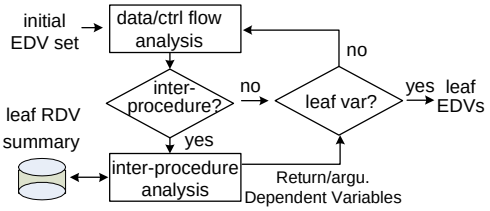


Figure 11: **Leaf-EDV identification.** SyncFinder recursively tracks Exit Dependent Variables(EDVs) along the data-, control-flow, until it reaches a leaf-EDV.

(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 [48, 38, 15].

SyncFinder first starts from variables directly referenced in the exit condition. They are added into an EDV set. Then, as shown in Figure 11, 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 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.

(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 12. 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 */
```
(a) Loop-variant *module*

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

Figure 12: **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) 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.

3.4 Synchronization Pairing

Once we identify a potential sync loop, we find the remote update (referred as *a sync write*) that would "release" (break) the wait loop. To identify a sync write, SyncFinder

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

Table 7: **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.

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 7), 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.

4 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.

4.1 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 2 and the MySQL example on Figure 3. 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.

4.2 Extensions to data race detection

We also extend Valgrind [33]'s dynamic data race detector to leverage SyncFinder's auto-identification of ad hoc sync loops. Valgrind implements a happens-before algorithm [21] 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 12) as discussed in Section 2 and illustrated using two examples in Figure 7.

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.

	Apps.	Total loops	Identified Sync Loops			Missed ones
			Total	True	FP	
Server	Apache	1462	17	15	2	1
	MySQL	4265	48	42	6	3
	OpenLDAP	2044	18	14	4	1
	Cherokee	748	6	6	0	0
	AOLServer	496	6	6	0	-
	Nginx	705	12	11	1	-
	BerkeleyDB	1006	15	11	4	-
	BIND9	1372	5	4	1	-
Desktop	Mozilla-js	848	16	11	5	1
	PBZip2	45	7	7	0	0
	Transmission	1114	14	12	2	1
	HandBrake	551	13	13	0	-
	p7zip	1594	10	9	1	-
	wxDFast	154	6	6	0	-
Scientific	Radiosity	80	12	12	0	0
	Barnes	88	7	7	0	0
	Water	84	9	9	0	0
	Ocean	339	20	20	0	0
	FFT	57	7	7	0	0
	Cholesky	362	8	8	0	-
	RayTracer	144	3	3	0	-
	FMM	108	8	8	0	-
	Volrend	77	9	9	0	-
	LU	38	0	0	0	-
	Radix	52	14	14	0	-
	Total (Ave.)	-	290 (11.6)	264 (10.6)	26 (1.0)	-

Table 8: **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 because some code (for other platforms such as FreeBSD, etc) are not included during the compilation.

5 Evaluation

5.1 Effectiveness and Accuracy

We evaluated SyncFinder on 25 concurrent programs, including 12 used in our manually characteristic study and 13 other ones. Table 8 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 4.

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 8, 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.

5.2 Sync Loop Identification and Pruning

Apps.	Total loops	Exit cond.	Leaf-EDVs	Aft non--shared pr.	Aft loop-var. pr.	Aft cond-var pr.
Apache	1,462	3,120	8,682	184	24	20
MySQL	4,265	9,181	20,458	377	118	72
OpenLDAP	2,044	4,434	11,276	171	45	27
PBZip2	45	278	799	130	16	9

Table 9: **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.

To show the effectiveness of sync loop identification, in Table 9, we test SyncFinder on some server/desktop applications and show the results from each of the sync loop identification steps. From the total loops identified, SyncFinder extracts exit conditions, and identifies all leaf-EDVs (the third column in Table 9). 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.

5.3 Synchronization Pairing and Pruning

During synchronization pairing, SyncFinder applies two pruning schemes, unsatisfiable remote update pruning and

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

Table 10: **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)

serial pair pruning. Table 10 shows the effect of those pruning steps on the same set of server/desktop applications in Table 9. 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 concurrent. To be conservative, SyncFinder does not prune the pairs inside such regions. Fortunately, the remote update based pruning helps filtering them out.

5.4 Two Use Cases: Bug Detection

Apps.	Deadlock (New)	Bad practice
Apache	1 (0)	1
MySQL	2 (2)	13
Mozilla	2 (0)	2

Table 11: **Deadlock and bad practice detection**

Table 11 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 2 and Figure 3. 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.

Apps.	Original Valgrind	Extended Valgrind	%Pruned
Apache	30	17	43%
MySQL	25	10	60%
OpenLDAP	7	4	43%
Water	79	11	86%

Table 12: **False positive reduction in Valgrind**

Table 12 shows that SyncFinder auto-annotation could reduce the false positive rates of Valgrind data race detector by 43-86%.

6 Related Work

Spin and hang detection Some recent work has been proposed in detecting *simple* spinning-based synchronizations [32, 25, 18]. For example, [25] proposed some new hardware buffers to detect spinning loops on-the-fly. [18] 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 as "sc-dir" in Table 3 in Section 2). As shown in Table 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 Section 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 [4, 2, 1, 41] 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.

Concurrent bug detection tools Much research has been conducted on concurrency bug detection [47, 20, 31, 6, 17, 11, 43]. These tools usually assume that they can recognize all synchronizations in target programs. As we demonstrated using deadlock detection and race detection, SyncFinder can help these tools improve their effectiveness and accuracy by automatically annotating ad hoc synchronizations that are hard for them to recognize.

Transactional memory Various transactional memory designs have been proposed to solve the programmability issues related to mutexes [39, 30, 19, 44] and also condition variables [10]. Our study complements such work by providing ad hoc synchronization characteristics in real world applications.

Software bug characteristics studies Several studies have been conducted on the characteristics of software bugs [8, 42, 34], including one of our own [26] on concurrency bug characteristics. This paper 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 this paper 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.

7 Conclusions and Limitations

In this paper, we provided a quantitative characteristics study of ad hoc synchronization in concurrent programs and built a tool called SyncFinder to automatically identify

and annotate them. By examining 229 ad hoc synchronization loops from 12 concurrent programs, we have 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 detect these ad hoc synchronizations, we developed SyncFinder, a tool that successfully identifies 96% of ad hoc synchronization loops with a 6% false positive rate. 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 [31]) 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.

All work has limitations, and ours is no exception: (i) 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. (ii) 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.) (iii) 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 [49] to help identify false positives. (iv) For our characteristic study, we can always study a few more applications, especially of different types.

8 Acknowledgments

We would like to express our deepest appreciation to our shepherd, Professor George Candea, who was very responsive during our interactions with him and provided us with valuable suggestions, which have significantly improved our paper and strengthened our work. Moreover, we would also like to thank the anonymous reviewers whose comments and insightful suggestions have greatly shaped and improved our paper and have taught us many important lessons. Finally, we greatly appreciate Bob Kuhn, Matthew Frank and Paul Petersen for their continuous support and encouragement throughout the whole project, as well as their insightful feedback on the project and the paper. This work is supported by NSF-CSR 1017784, NSF CNS-0720743 grant, NSF CCF-1017804 grant, NSF CNS-1001158 (career award) and Intel Grant.

References

[1] Java annotations. `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`.

[2] Lock_Lint - Static data race and deadlock detection tool for C. `http://developers.sun.com/sunstudio/articles/locklint.html`.

[3] Miscompiled volatile-qualified variables. `https://www.securecoding.cert.org/confluence/display/seccode/DCL17-C.+Beware+of+miscompiled+volatile-qualified+variables`.

[4] MSDN run-time library reference – SAL annotations. `http://msdn2.microsoft.com/en-us/library/ms235402.aspx`.

[5] BOEHM, H.-J., AND ADVE, S. V. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 68–78.

[6] BRON, A., FARCHI, E., MAGID, Y., NIR, Y., AND UR, S. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA* (2005), ACM, pp. 206–212.

[7] BURNS, B., GRIMALDI, K., KOSTADINOV, A., BERGER, E. D., AND CORNER, M. D. Flux: A language for programming high-performance servers. In *USENIX Annual Technical Conference, General Track* (2006), USENIX, pp. 129–142.

[8] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. R. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (18th SOSP'01)* (Banff, Alberta, Canada, Oct. 2001), ACM SIGOPS, pp. 73–88.

[9] CONDIT, J., HARREN, M., ANDERSON, Z. R., GAY, D., AND NECULA, G. C. Dependent types for low-level programming. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007* (2007), Springer, pp. 520–535.

[10] DUDNIK, P., AND M.SWIFT, M. Condition variables and transactional memory: Problem or oppertunity? In *4th ACM SIGPLAN Workshop on Transactional Computing(Transact)*.

[11] ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (Oct. 19–22 2003), pp. 237–252.

[12] FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2004), ACM, pp. 256–267.

[13] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA* (1993), pp. 289–300.

[14] HOARE, C. A. R. Monitors: an operating system structuring concept. *Communications of the ACM 17* (1974), 549–557.

[15] HORWITZ, S., REPS, T. W., AND BRINKLEY, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, 1988), pp. 35–46.

[16] HOWARD, J. H. Proving monitors. *Commun. ACM 19*, 5 (1976), 273–279.

[17] INTEL CORPORATION. Intel thread checker. http://software.intel.com/en-us/articles/intel-thread-checker-documentation/.

[18] JANNESARI, A., AND TICHY, W. F. Identifying ad-hoc synchronization for enhanced race detection. In *IPDPS* (April 2010), IEEE.

[19] JAYARAM BOBBA, NEELAM GOYAL, M. H.-M. S. D. W. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *International Symposium on Computer Architecture* (June 2008), pp. 127–138.

[20] JULA, H., TRALAMAZZA, D. M., ZAMFIR, C., AND CANDEA, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA* (2008), pp. 295–308.

[21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[22] LAMPSON, B., AND REDELL, D. Experience with processes and monitors in mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[23] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[24] LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX Annual Technical Conference, General Track* (2005), pp. 31–44.

[25] LI, T., LEBECK, A. R., AND SORIN, D. J. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems PDS-17*, 6 (June 2006), 508–521.

[26] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2008).

[27] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 37–48.

[28] MANSON, J., PUGH, W., AND V.ADVE, S. The java memory model. In *POPL* (January 2005), ACM.

[29] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. *Special issue on performance evaluation tools for parallel and distributed computer systems 28* (November 1995), 37–46.

[30] MINH, C. C., TRAUTMANN, M., CHUNG, J., MCDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, NY, USA, 2007), ACM, pp. 69–80.

[31] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings* (2008), USENIX Association, pp. 267–280.

[32] NAKKA, N., SAGGESE, G. P., KALBARCZYK, Z., AND IYER, R. An architectural framework for detecting process hangs/crashes. In *EDCC: EDCC, European Dependable Computing Conference* (2005).

[33] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (2007), 89–100.

[34] OSTRAND, T. J., WEYUKER, E. J., AND BELL, R. M. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Eng 31*, 4 (2005), 340–355.

[35] PADIOLEAU, Y., TAN, L., AND ZHOU, Y. Listening to programmers taxonomies and characteristics of comments in operating system code. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 331–341.

[36] PARK, S., LU, S., AND ZHOU, Y. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009* (2009), ACM, pp. 25–36.

[37] RAJWAR, R., AND HILL, M. Transactional memory bibliography. http://www.cs.wisc.edu/trans-memory/biblio/.

[38] REPS, T., AND ROSAY, G. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1995), ACM, pp. 41–52.

[39] ROSSBACH, C. J., HOFMANN, O. S., PORTER, D. E., RAMADAN, H. E., ADITYA, B., AND WITCHEL, E. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), ACM, pp. 87–102.

[40] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), ACM Press.

[41] STERLING, N. WARLOCK - A static data race analysis tool. In *USENIX Winter Technical Conference* (1993), pp. 97–106.

[42] SULLIVAN, M., AND CHILLAREGE, R. A comparison of software defects in database management systems and operating systems. In *FTCS* (1992), pp. 475–484.

[43] SUN MICROSYSTEMS INC. Thread analyzer user's guide. http://dlc.sun.com/pdf/820-0619/820-0619.pdf.

[44] TIM HARRIS, SIMON MARLOW, S. P.-J. M. H. Composable memory transactions. In *ACM SIGPLAN symposium on Principles and practice of parallel programming* (2005).

[45] V.ADVE, S., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. In *computer* (1996), IEEE.

[46] VAZIRI, M., TIP, F., AND DOLBY, J. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 334–345.

[47] WANG, Y., KELLY, T., KUDLUR, M., LAFORTUNE, S., AND MAHLKE, S. A. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA* (2008), pp. 281–294.

[48] WEISER, M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), IEEE Press, pp. 439–449.

[49] ZAMFIR, C., AND CANDEA, G. Execution synthesis: a technique for automated software debugging. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), ACM, pp. 321–334.

[50] ZHOU, F., CONDIT, J., ANDERSON, Z. R., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G. C., AND BREWER, E. A. Safedrive: Safe and recoverable extensions using language-based techniques. In *OSDI* (2006), pp. 45–60.