Synchronization Identification through On-the-Fly Test

Xiang Yuan[1,2], Zhenjiang Wang[1], Chenggang Wu[1,⋆], Pen-Chung Yew[3,4], Wenwen Wang[1,2], Jianjun Li[1], and Di Xu[5,⋆⋆]

[1] SKL of Computer Architecture, Institute of Computing Technology, CAS
[2] University of Chinese Academy of Sciences Beijing, China
{yuanxiang,wangzhenjiang,wucg,wangwenwen,lijianjun}@ict.ac.cn
[3] Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, USA
yew@cs.umn.edu
[4] Institute of Information Science, Academia Sinica, Taiwan
[5] IBM Research - China
xudi@cn.ibm.com

Abstract. Identifying synchronizations could significantly improve testing and debugging of multithreaded programs because it could substantially cut down the number of possible interleavings in those tests. There are two general techniques to implement synchronizations: modularized and ad-hoc. Identifying synchronizations in multi-threaded programs could be quite challenging. It is because modularized synchronizations are often implemented in an obscure and implicit way, and ad-hoc synchronizations could be quite subtle. In this paper, we try to identify synchronizations from a new perspective. We found that if a thread is waiting for synchronizations, the code it executes during the wait is very different from that after the completion of the synchronization. Based on such an observation, we proposed an effective method to identify synchronizations. It doesn't depend on the understanding of source codes or the knowledge of semantics of library routines. A system called SyncTester is developed, and experiments show that SyncTester is effective and useful.

Keywords: synchronization identification, concurrency testing, multithreading.

1 Instruction

Many debugging and testing algorithms use guided or unguided interleaving among threads as the basis of such tests, e.g. Eraser [1], FastTrack [3] and CTrigger [4]. However, the number of interleaves in multi-threaded programs could be enormous. It will make those algorithms very complicated and time consuming. To reduce the number of possible interleavings, and thus reducing the time complexity, these algorithms try to exploit synchronizations. It can also reduce

⋆ To whom correspondence should be addressed.
⋆⋆ This work was done when Di Xu attended Institute of Computing Sciences, CAS.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 4–15, 2013.
© Springer-Verlag Berlin Heidelberg 2013

false positives in the test results. Previous work [12] shows that synchronizations could reduce 43-86% false data races found by Valgrind. Therefore, identifying synchronizations in multithreaded programs is very desirable and important.

In general, there are 2 types of synchronizations: modularized [12] and ad-hoc. Modularized synchronization could be identified by their semantics [5,3,7]. And most approaches identify ad-hoc ones by pattern-matching [7,8,12].

Identifying modularized synchronizations using their semantics could be tedious and error prone. There exist hundreds of libraries. Some of them have many routines. E.g. GLIBC 2.16 has about 1200 routines [14]. Some routines provide implicit synchronization functions (e.g. read()/write()). Moreover, some synchronization pairs could be from different libraries (e.g. pthread_kill/sigwait). Therefore, this work could be quite challenging for programmers.

To identify ad-hoc synchronizations, ISSTA08 [7] focuses on synchronizations that consist of a spinning read and a corresponding remote store. It is dynamic. Helgrind+ [8] and SyncFinder [12] focus on loops whose exit condition cannot be satisfied in their loop bodies. SyncFinder is a static method, while Helgrind+ searches for such loops statically, but identifies the remote stores dynamically. They are all based on the synchronization patterns summarized from various application codes. Because of the complexity and the large amount of codes, this process is time consuming and may miss or misjudge some patterns. To identify complex patterns, elaborate inter-procedural pointer analysis may be needed. However, there are still many cases pointer analysis cannot handle.

In this paper, we try to identify synchronizations from a different perspective. We leverage the essential feature of a synchronization that it forces a thread to wait when the thread may violate the intended order imposed by the programmer. Our scheme depends on neither the patterns nor the knowledge of library routines, thus labor-intensive pattern collection/recognition and learning of library routines are avoided. Moreover, our proposed scheme works on binary executable, which is an advantage when source code is not available. Overall, our work makes the following contributions:

(1) Propose a synchronization identification scheme from a new perspective. It can be used to identify both modularized and ad-hoc synchronizations in multi-threaded programs regardless of their implementation.

(2) Implement a prototype system, SyncTester. Experimental results on real programs show that it is very effective and useful.

(3) Introduce helper processes to do contrast test. With their help, SyncTester can avoid perturbation to the program execution.

2 Synchronization Testing

Our scheme tries to identify synchronization by monitoring the execution of a program. A multi-threaded program is tested for several executions. In each execution, we select a different thread as the *testing thread* and all other are *executing threads*. We propose two algorithms: a *forward test* and a *backward test*. They identify synchronizations according to the intended order between the testing thread and the executing threads. Fig.1 gives an overview of it.

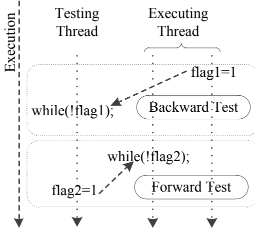

Fig. 1. Test Scheme Overview

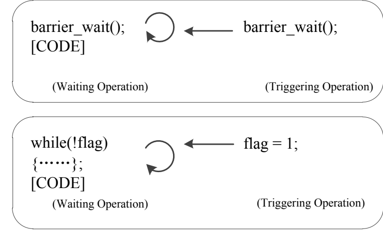

Fig. 2. Behavior of waiting operations

2.1 Forward Test

Synchronizations are used to control the orders among operations in different threads. A synchronization involves at least two threads: one is waiting (*waiting thread*) for a certain operation from another (*triggering thread*). We refer to the waiting action in the waiting thread as *waiting operation*, and the operation that wakes up the waiting thread as *triggering operation.* A waiting operation and its corresponding triggering operation form a *synchronization pair (sync pair).*

Waiting operations have two forms, as shown in Fig.2. The waiting thread may be blocked by a library call or spinning on a code segment. The triggering operation should be a library call or a shared memory store instruction respectively. So we take all the library calls and shared memory store instructions as *potential triggering operations* (PTO). When a thread is waiting for a certain operation, we say that it is in a *suspended state*, and such a thread is executing a *potential waiting operation* (PWO). When the corresponding triggering operation occurs, waiting thread will exit the suspended state and go on to execute its subsequent codes, as [CODE] shown in Fig.2. The code it executes during a suspended state is different from that after it exits this state.

Then we design a sync-pair identification scheme. Algorithm 1 shows its details. When testing thread encounters a PTO, we suspend its execution until all executing threads enter suspended states (Line 14-16). Then we execute the PTO. If it makes an executing thread exit a suspended state, we probably find a sync pair (Line 23-29).The italic light-grey codes are the specific techniques to improve the scheme.

This algorithm is “safe” in the sense that we allow only one testing thread in each test. Hence, if all the PTOs in the testing thread are not real triggering operations, executing threads will continue running to their completion without being suspended. If none of the suspended executing threads exits its suspended state after a PTO is executed, the testing thread will continue executing until a real triggering operation releases an executing thread.

It is important to identify whether a thread is in a suspended state or not. A basic block vector (BBV) [9] records the executing frequencies of all basic blocks in a time quantum. Every thread builds its own BBVs during execution. When a thread is in a suspended state, the BBVs of its continuous quanta should be very similar. When a threads BBVs in continuous quanta have small differences [9] and contain same blocks, this thread is marked as in a suspended state.

Algorithm 1 Forward Test

```
1: Input: Set<Thread> Threads, including all threads in a multi-
   threaded program
2: Map<Thread, PWO> pwOp := Φ;
3: Map<Thread,Process> helpProc := Φ;
4: Set<Thread> testingThreads = Threads;
5: Set<Thread> execThreads;
6: //Algorithm begins
7: testingThreads = ThreadGroup(testingThreads);
8: for testThread∈testingThreads do
9:    //Each iteration of this for loop is a pass of execution
10:   while testThread is not finished do
11:      execThreads:= Threads \ {testThread}
12:      //execThreads are executing freely.
13:      nextOp := NextPTO(testThread);
14:      while not AllThreadSuspended(execThreads) do
15:         sched_yield();
16:      end while
17:      if not RepeatOp(nextOp) then
18:         execThreads:= TypeMatching(execThreads, nextOp);
19:         for t∈ execThreads do
20:            pwOp[t] := t's current PWO;
21:            helpProc[t] := CreateHelperProc(t);
22:         end for
23:         Execute(testThread, nextOp);
24:         for t∈ execThreads do
25:            if not Suspended(t) && Suspended(helpProc[t]) then
26:                  RecordSyncPair(nextOp, pwOp[t]);
27:            end if
28:            Terminate(helpProc[t]);
29:         end for
30:      else
31:         Execute(testThread, nextOp);
32:      end if
33:   end while
34: end for
```

Algorithm 2 Backward Test

```
1: Input: Set<Thread> Threads, including all threads in a multi-
   threaded program
        Set<SyncPair> SyncPairs, including sync pairs
   found by forward test
2: Set<Thread> targetThreads = Threads;
3: Set<Thread> execThreads;
4: targetThreads = ThreadGroup(targetThreads);
5: for testingThread∈targetThreads Do
6:    //Each iteration of this for loop is a pass of execution
7:    execThreads := Threads \ {testingThread}
8:    //execThreads are executing freely
9:    while testingThread is not finished do
10:      nextOp := NextReadOp(testingThread);
11:      if not AllThreadSuspended(execThreads) then
12:         WaitForSuspending(execThreads);
13:      end if
14:      wrOp = LastWriteOp(nextOp);
15:      if MatchHappenBefore(nextOp, wrOp, SyncPairs) &&
   not RepeatedOp(nextOp, wrOp) && not LockOp(nextOp,
   wrOp) && wrOp is not in testingThread then
16:         CreateHelperProcBT(nextOp, wrOp);
17:      end if
18:      Execute(testingThread, nextOp);
19:   end while
20: end for
21: //Define <rdOp, wrOp> as a Backward Test Pair
22: CreateHelperProcBT(OP rdOp, OP wrOp) begin
23:   Fork(); //Create Helper Process
24:   if this is helper process then
25:      //The following codes execute concurrently with main
   process
26:      RestoreToPreviousValue(this, wrOp);
37:      if Suspended(this) then
28:         helperProc2 = CreateHelperProcess(this);
29:         ResetToCurrentValue(helperProc2, wrOp);
30:         if Suspended(this) != Suspended(helperProc2) then
31:            RecordSyncPair(wrOp, rdOp);
32:         endif
33:       end if
34:      Terminate(this, helperProc2);//Kill helper processes
35:   end if
36: end
```

In the following cases, the differences of a threads continuous BBVs will be small. The thread is (1) blocked by a system call, (2) has exited or (3) spinning on a code segment. In the first two cases, the thread is no longer executing, and will not build BBVs, and all the elements of its BBVs are 0.The differences of its BBVs are 0.

When a thread exits its suspended state, it starts to execute other code segments. So, if we find a thread executes blocks different from those in the suspended state, it has exited its previous suspended state.

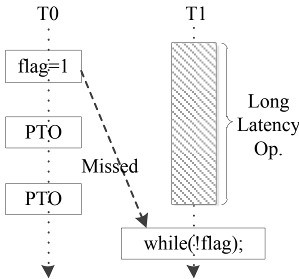

Fig. 3. Motivation of Backward test

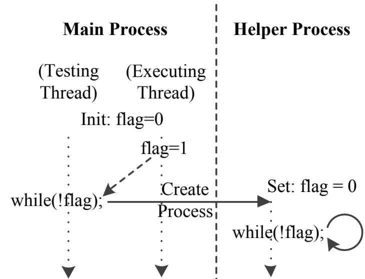

Fig. 4. Schematic of Backward Test

2.2 Backward Test

Algorithm 1 can identify many sync pairs. However, in some special cases, it may miss some sync pairs. For example, in Fig.3, “flag=1” and “while(!flag)” form a sync pair. Assume there is a *long latency operation (LLO)* in T1. We refer to an operation as a LLO if it lasts for several time quanta and can continue to execute without being triggered by other threads. When T1 is executing a LLO, Algorithm 1 may mistake T1 as entering a suspended state and miss this sync pair. To counter this difficulty, we propose a *backward test*.

Note that the *forward test* suspends the testing thread at each potential triggering operation. It allows executing threads to run ahead of the testing thread and enter suspended states. However, LLOs can distort it. The proposed *backward test* targets sync pairs whose triggering operations execute before the waiting operations. The sync pair shown in Fig.3 can be identified if we use a *backward test* and select T1 as the testing thread.

If triggering operation executes before waiting operation, the waiting operation will not spin or be blocked. This is because the triggering operation has removed the wait condition. Therefore, when a *shared variable* is read by the main program or library, we restore the shared variable to its *previous value*. If this thread is blocked or spinning on a shared variable, we treat the corresponding library routine (or code segment) and the operation that performs the previous store to the shared variable as a sync pair.

However, restoring a shared variable to its previous value will cause errors to the programs execution. Therefore, we perform such a test in a separate process, called *helper process*. Helper processes are created (by *fork()* syscall in Linux) when we encounter read operations to shared variables or library calls. They communicate with the main process via pipes. Because they need to inherit the current context of the main process, they can't be created in advance.

This is called a *backward test*. Fig.4 is its schematic, and Algorithm 2 is its details. Note that Algorithm 2 defines the backward test pair in Line 21.

Helper Process Isolation. Helper processes inherit the context of the main process. They share some resources with the main process, such as file descriptors and shared memory. If they access these resources, they will interfere with the execution of main process. Hence we need to handle system calls and shared memory in helper processes appropriately.

We classify system calls into 3 types and handle them in helper processes differently. (Type 1) System calls that don't access such shared resources, invoke them as usual, e.g. *gettimeofday()*, *futex()*; (Type 2) System calls that get data from shared resources only, e.g. *read()*, get these data from the main process. (Type 3) System calls that store data or change status of shared resources, ignore them and get their return values from main process, e.g. *open()*, *write()*.

Processes may use shared memory to communicate with other processes. We transform shared memory in the helper process into private. We back up the data in shared memory, detach it and then map private one. The range of shared memory can be got by system call instrumentation in main process.

2.3 Perturbation Elimination

Long Latency Operations (LLOs). Take the forward testing as an example. When an executing thread enters a LLO, our scheme may misidentify the executing thread as entering a suspended state. When the testing thread executes a potential triggering operation, if this executing thread happens to exit its misidentified suspended state, the forward test will mistakenly identify the LLO as a waiting operation. A false sync pair will be identified, i.e. a *false positive.*

The key to distinguish a LLO from a *true waiting operation* is the manner it exits the suspended state. A LLO can exit the suspended state by itself after finishing its work. But a waiting operation cannot exit the suspended state until another thread executes its corresponding triggering operation.

Fig.5 shows how to distinguish LLOs in forward test (Line 21 & 25 Algorithm 1). When testing thread (T0) encounters a potential triggering operation (flag=1;), a helper process is created by the executing thread (T1). Note that the helper process does not interact with the testing thread (i.e. 'flag' is always 0 in the helper process). After the testing thread has executed "flag=1;", we compare the state of T1 and helper process. If T1 and the helper process both exits the suspended state (Fig.5(b)), it means that the operation that made T1 enter a suspended state is a LLO. Otherwise we find a real sync pair (Fig.5(a)).

In the backward testing (Line 28-30 Algorithm 2), we take a similar approach to handle LLOs. When the helper process enters a suspended state, we create a second helper process using the first helper processs context. In the second helper process, we reset the value of the shared variables restored by the first helper process to the same as that in main process. If one helper process exits the suspended state while the other dose not, it means that there is a helper process waiting for the new values of these shared variables. We could treat this backward test pair as a sync pair.

Lock/Unlock Operations. When the helper process tests a lock operation in backward test, it restores the values of the lock variables to those before an unlock operation. These values make the helper process have held the lock and will be blocked by this lock operation. In order to avoid identifying lock/unlock operations as sync pairs, when we encounter a library routine at the first time, we create a helper process to test whether it is a lock operation or not. This helper process creates two threads and each of them invokes this library routine with the same arguments. From the definitions of ABI, we will know where the arguments are (e.g. register or stack). If one of these two threads is blocked and the other is not, this library routine is treated as a lock operation and will not form sync pairs. (LockOp() in Line 15 Algo.2)

2.4 Efficiency Issues

Efficiency is always a concern in testing multi-threaded programs. There are two ways to improve it. One is to reduce the number of testing threads, and the other is to reduce the number of testing points in testing threads. Testing points include potential triggering operations and backward test pairs.

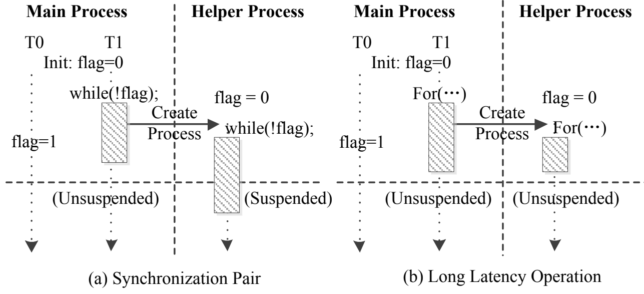

Fig. 5. Handling LLOs in forward test

Reducing Testing Threads. To reduce the number of testing threads, we divide the threads in a program into groups. If two threads are similar in their executions, the sync pairs they executed are likely to be the same. Hence, we put such two threads in a group. In each group, we select only one thread as testing thread (ThreadGroup() in Line 7 Algo.1 and Line 4 Algo.2).

In order to group threads, we calculate the difference between the BBVs of their executions. If the difference is smaller than a threshold, the executions of the two threads are likely to be similar. They are put into the same group.

This is heuristic. To reduce the probability of missing sync pairs due to thread grouping, we build a sync-op set. It contains triggering and waiting operations of sync pairs we found and their calling contexts. After all testing threads are tested, if we find that a thread executes an operation in the *sync-op* set with a different calling context and this operation doesn't belong to a sync pair, we select this thread as testing thread and do another pass of test.

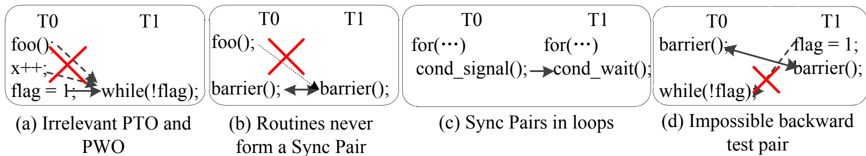

Fig. 6. Cases to reduce number of testing points

Reducing Testing Point. Fig.6 shows some cases, in which we can reduce the number of testing points. We propose the following schemes targeting them.

(1)Match the type/address of synchronization operations. (Type-Matching() in Line 18 Algo.1 and LastWriteOp() in Line 14 Algo.2)

In Fig.6(a), the waiting operation and triggering operation of a modularized sync pairs should be library routines, while an ad-hoc sync pair should consist of a spin loop and a store operation. So, when we encounter a library call, we can skip testing store operations to shared variable, and vice versa.

Furthermore, for an ad-hoc sync pair in the forward test, its waiting operation and triggering operation should access the same shared variable. According to

the shared variables read by executing threads during a suspended state, we can skip store operations that access other shared variables in the testing thread.

(2)Use history of routines. (RepeatOp() in L.17 Algo.1 & L.15 Algo.2)

After analyzing various popular library routines, we found that a library routine with blocking function can only be unblocked by a few specific library routines. If a pair of library routines is tested for several times and they never form a sync pair, as shown in Fig.6(b), it is most likely that they are not a sync pair at all. So, when we encounter the same library pair, skip the test on them.

(3)Accelerate test in loops. (RepeatOp() in L.17 Algo.1 & L.15 Algo.2)

There exist sync pairs that are in loops as shown in Fig.6(c). Such sync pairs appear in every iteration and we need not to test them every time.

If a sync pair is found repeatedly, we assume it is in a loop. In forward test, we record the potential triggering operations (excluding its triggering operation) appear between two occurrences of its waiting operation, and skip them when the waiting operation appears again. In the backward test, we skip testing it.

(4)Use results of the forward test. (MatchHappenBefore() in L.15 Algo.2)

Sync pairs define happens-before relations between code segments. We can utilize sync pairs identified in a forward test to reduce testing points in a backward test. The happen-before relation defined by barrier() in Fig.6(d) ensures that the store (flag=1;) in a backward test pair executes before its corresponding read (while(!flag);). Then the read can't get the old value before the store and we don't need to test such cases in a backward test.

3 Evaluation

SyncTester is implemented using Pin [16]. It uses Pins API to instrument instructions that may access shared variables, code blocks, library routines and system calls. It then collects information and controls the execution of target programs. For example, SyncTester instruments store instructions and syscalls. It then records a shared variables previous value for the backward test.

We evaluated SyncTester on a series of multi-threaded programs. The test programs are from benchmark suites, such as SPLASH2 [10] and STAMP [11], or applications, such as PBZIP2, PFSCAN, and Apache Httpd. We con-figured SPLASH2 using POSIX thread library and configured STAMP with a simple software TM provided by its web site. Our experiments are run on a server with two 2.27GHz Intel Xeon E5520 quad-core processors and 8GB DRAM.

3.1 Effectiveness

In order to evaluate SyncTesters effectiveness, we compare it with two existing dynamic test schemes, ISSTA08 and Helgrind+. We implement ISSTA08's algorithm [7] and use Helgrind+'s newest version. These two approaches are designed to identify ad-hoc sync pairs. So, we compare them only for ad-hoc sync pairs. The results show that SyncTester found more sync pairs than ISSTA08 and Helgrind+ did, and it introduced very few false positives as shown in Table 1. In fact, SyncTest covered all sync pairs found by ISSTA08 and Helgrind+.

Table 1. ‘All’ column shows the number of identified sync pairs. ‘FP’ column shows the number of false positives and we verify them manually. The results of SyncTester are in the form of X(modularized)/Y(ad-hoc). FT and BT columns show the results of forward test and backward test, respectively.

Benchmarks	ISSTA08		Helgrind+		SyncTester				Pruned LLOs		Pruned Lock/ Unlock
	All	*FP*	*All*	*FP*	*All*	*FP*	*FT*	*BT*	*FT*	*BT*	
BARNES	1	0	2	2	5/2	3	5/2	5/1	4	235	4
FMM	4	0	8	2	10/8	1	10/4	10/5	12	197	23
OCEAN-C	0	0	0	15	20/0	0	20/0	20/0	0	0	6
OCEAN-NC	0	0	0	40	19/0	0	19/0	19/0	0	0	6
RADIOSITY	0	0	0	7	3/2	0	3/2	3/0	3	11	10
RAYTRACE	0	0	0	9	1/0	1	1/0	1/0	0	0	3
VOLREND	2	0	2	5	5/2	0	5/2	5/1	5	16	5
WATER-S	0	0	0	15	7/0	0	7/0	7/0	0	0	7
WATER-N	0	0	0	5	9/0	0	9/0	9/0	0	0	9
INTRUDER	1	1	2	5	3/2	0	3/0	0/2	3	0	3
LABYRINTH	0	0	1	3	2/1	0	2/0	0/1	2	0	3
PFSCAN	0	0	1	3	3/2	0	3/0	0/2	0	0	4
PBZIP2	0	0	0	0	5/5	0	5/0	0/5	17	0	13
Apache Httpd	0	0	*	*	1/7	1	1/2	0/5	0	0	21

* Maybe the version of Apache HTTPD is not fit for Helgrind+, Helgrind+ exits unexpectedly during our test to Apache Httpd. So we don’t get such data.

FPs in Helgrind+ are probably because it searches for spinning loops on binary codes statically. And it is not accurate enough. For SyncTester, backward test restores a shared variable’s value and don’t know the relationship among different shared variables. This inconsistency brings FPs in some regular loops. And no FP is found in forward test.

The last 3 columns of Table 1 shows the results of perturbation elimination. In those experiments, we found that LLOs include some long loops and library routines. We can prune both of them no matter how they are implemented.

However, because we can’t restore the states of system kernel, the backward test may miss some modularized sync pairs due to system calls, e.g. pthread_kill() and sigwait(). Such cases can be found by the forward test.

3.2 Efficiency Issues

Reducing the Number of Testing Threads. We set a threshold on the difference of BBVs in thread grouping. Different thresholds will result in different thread groupings, as shown in Table 2. “Real Thread Groups” column shows the best groupings found manually. It gives the minimum number of testing threads. We check the results in Table 2 and find that if the number of thread group is no fewer than this amount, the testing threads selected will contain all threads in the “Real Thread Groups”. This happens to be the most prevalent case. It shows that our thread grouping scheme is quite effective. Finally, we choose 0.4 as the threshold. It is chosed as a tradeoff between accuracy and efficiency. In this case, labyrinth misses a sync pair, but it is found in the sync-op set.

The last 4 columns of Table 2 shows the results of thread grouping when the number of worker threads changes. For programs with many threads, it is likely that most threads execute similar codes, and this scheme will also reduce the number of testing threads. If we test a program with different testing threads, these tests can run concurrently. This will also save testing time.

Reducing the number of Testing Points. Table 3 is the results of testing point reduction. It shows that there are not many testing points in most programs. It also shows the number of testing points pruned by optimizations. For most benchmarks, more than 97% testing points can be pruned. If we have to test all of them, the efficiency of SyncTester will become quite unacceptable. In the backward test, we perform test in helper processes. Because helper processes can run concurrently, this can reduce the time overhead caused by them.

Table 2. Results of Thread Grouping

Benchmarks	#ths	# thread groups under different threshold						Real Thread Groups	#total threads	# worker threads (threshold = 0.4)		
		0.1	*0.2*	*0.3*	*0.4*	*0.5*	*1*			4	8	16
BARNES	4	1	1	1	**1**	1	1	1	N	1	1	1
FMM	4	4	4	2	**2**	2	1	1	N	2	6	4
OCEAN-C	4	2	1	1	**1**	1	1	1	N	1	1	1
OCEAN-NC	4	1	1	1	**1**	1	1	1	N	1	1	1
RADIOSITY	4	4	3	1	**1**	1	1	1	N	1	2	3
RAYTRACE	4	4	2	2	**1**	1	1	1	N	1	2	4
VOLREND	4	3	2	2	**2**	2	1	1	N	2	2	4
WATER-S	4	2	1	1	**1**	1	1	1	N	1	1	1
WATER-N	4	2	2	2	**1**	1	1	1	N	1	1	2
INTRUDER	4	3	3	2	**2**	1	1	2	N	2	3	3
LABYRINTH	4	1	1	1	**1**	1	1	2	N	1	1	1
PFSCAN	5	4	4	4	**4**	4	4	2	1+N	4	4	6
PBZIP2	8	6	6	6	**6**	6	5	5	4+N	6	5	5
Apache HTTPD	7	4	4	4	**4**	4	4	4	3+N	4	4	4

Table 3. Testing Points Reduction. Columns 2 and 3 are the real testing points during tests. Columns 4 to 7 are testing points pruned by our 4 schemes.

Benchmarks	Testing Points		Pruned Testing Points				% Pruned
	FT	*BT*	*Type Match*	*RTN History*	*Loop Accel*	*Use FTs Results*	
BARNES	36	412	160K	27K	153K	3.8M	99.9892%
FMM	63	463	39K	2363	53K	125K	99.7608%
OCEAN-C	105	250	360K	1798	1186	36K	99.9110%
OCEAN-NC	146	251	364K	808	1157	53K	99.9055%
RADIOSITY	23	74	25	47	31	0	51.5000%
RAYTRACE	2	4	102K	514	0	0	99.9941%
VOLREND	41	33	25K	34	39	26K	99.8557%
WATER-S	111	21	230K	352K	0	2.26M	99.9980%
WATER-N	48	75	70K	1828	14K	3.61M	99.9975%
INTRUDER	60	6	87K	158K	0	379	99.9731%
LABYRINTH	156	15	22K	461	0	4735	99.3833%
PFSCAN	53	11	41	15K	0	0	98.2511%
PBZIP2	151	38	7062	51	0	0	97.4117%
Apache HTTPD	90	224	1566	385	0	0	86.1369%

3.3 Running Time

Finally, we measured the running time of SyncTester, which is shown in Fig.7. Among these benchmarks, Apache HTTPD is a server. It is not easy to measure its running time. So we instead measure its throughput, i.e. the number of processed requests per second. On average, the slowdown factor is 32X.

ISSTA08 claims that its slowdown factor is 9X [7]. Although SyncTester is slower, it identifies more sync pairs. And SyncTester can identify ad-hoc sync

pairs whose triggering operations execute before waiting operations and modularized sync pairs.Helgrind+ is a race detector. It identifies ad-hoc sync pairs to prune false races. For SPLASH2 benchmarks, its slowdown factor is more than 2000X. However, we don't compare it with our results because its main function is to detect data races, not sync pairs.

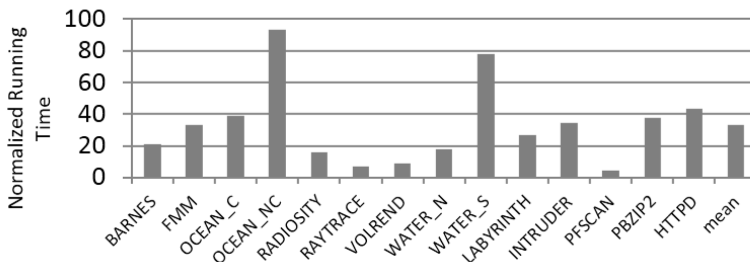

Fig. 7. Running Time of SyncTester

4 Related Work

ISSTA08[7] treats spinning reads and remote stores as a common pattern of ad-hoc synchronizations. However, if a remote store is executed before a spinning read, the spinning read will execute only once and it will miss such synchronizations. Helgrind+[8] tries to overcome such weaknesses of ISSTA08[7]. It searches for spinning loops whose exit conditions depend on loop invariant variables in the binary code and remote stores on the fly. However, in some complex cases, it may be difficulty to find spinning loops in binary codes. There is also a hardware scheme [15]. It uses some hardware buffers and detects spinning loops on the fly. SyncFinder [12] searches for loops with loop-invariant exit conditions. It is a static approach. All of its analysis is done on source code. It uses constant propagation to identify remote stores. Because the source codes are not always available, and pointer analysis is often not very precise, it may introduce some false positives and false negatives.

ATDetector [13] finds that address transfer (i.e. passing memory blocks' address between threads) also imposes implicit happens-before relation and could prune false races. Address transfer ensures that accesses to the memory block in the address sending thread happen before accesses in receiving thread.

5 Conclusion

In this paper, we showned that if a thread was held up in a synchronization by another thread, the code it executes during the wait is very different from that after the completion of the synchronization. From this observation, we propose a new approach to identify sync pairs in multi-threaded programs, called SyncTester. SyncTester can identify both modularized and ad-hoc sync pairs. It doesn't depend on the details of their codes and software implementation. Therefore, it has a great flexibility and is often more accurate than many existing approaches. Experimental results show that SyncTester is quite effective and practical.

Acknowledgements. This research is supported by the National High Technology Research and Development Program of China under the grant 2012AA010901, the National Natural Science Foundation of China (NSFC) under the grant No.61100011, the Innovation Research Group of NSFC under the grant 60921002, and the U.S. National Science Foundation under the grant CNS-0834599.

References

1. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. J. ACM TOCS 15(4), 391–411 (1997)
2. Zhang, W., Sun, C., Lu, S.: ConMem: detecting severe concurrency bugs through an effect-oriented approach. In: Proc. of 15th ASPLOS, pp. 179–192. ACM, NY (2010)
3. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the PLDI, pp. 121–133. ACM, NY (2009)
4. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th ASPLOS, pp. 25–36. ACM, NY (2009)
5. Cui, H., Wu, J., Tsai, C.C., Yang, J.: Stable deterministic multithreading through schedule memorization. In: Proceedings of the 9th OSDI. USENIX Association, Berkeley (2010)
6. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the 22nd SOSP, pp. 177–192. ACM, NY (2009)
7. Tian, C., Nagarajan, V., Gupta, R., Tallam, S.: Dynamic recognition of synchronization operations for improved data race detection. In: Proceedings of the ISSTA, pp. 143–154. ACM, NY (2008)
8. Jannesari, A., Tichy, W.F.: Identifying ad-hoc synchronization for enhanced race detection. In: Proceedings of the IPDPS, pp. 1–10. IEEE Press, NY (2010)
9. Sherwood, T., Perelman, E., Calder, B.: Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In: Proceedings of the PACT, pp. 3–14. IEEE Computer Society, Washington, DC (2001)
10. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd ISCA, pp. 24–36. ACM, NY (1995)
11. Minh, C.C., Chung, J.W., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Proceedings of the 2008 IISWC, pp. 35–46. IEEE Press, NY (2008)
12. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of the 9th OSDI. USENIX Association, Berkeley (2010)
13. Zhang, J., Xiong, W., Liu, Y., Park, S., Zhou, Y., Ma, Z.: ATDetector: improving the accuracy of a commercial data race detector by identifying address transfer. In: Proceedings of the 44th MICRO, pp. 206–215. ACM, NY (2011)
14. The GNU C Library manual, `http://www.gnu.org/software/libc/manual/`
15. Li, T., Lebeck, A.R., Sorin, D.J.: Spin detection hardware for improved management of multithreaded systems. J. IEEE PDS 17(6), 508–512 (2006)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the PLDI, pp. 190–200. ACM, NY (2005)