Research for concurrent program data race checking algorithm in control system

Hao Liang[1]*, Yunfeng Ai[2]

[1]*Company of Postgraduate Management, the Academy of Equipment, Beijing 101416, China*

[2]*College of Engineering & Information Technology, University of Chinese Academy of Sciences, Beijing 100049, China*

Received 26 September 2014, www.cmnt.lv

Abstract

The designing methods of multithreaded have already been used in control system widely. However the problems of data race, which are brought by multithreaded program, are being the difficulty in control system designing and testing currently. To this end, we optimized the thread-state analysis, designed the conservative lockset analysis. Further, We have introduced the thread-state analysis and conservative lockset analysis methods into Happens-Before relationship algorithm, designed a quick data race detecting method (DHTC) for control system multithreaded program with a certain hardware universality. The DHTC reduces the false alarm rate of Happens-Before relationship detecting methods, meanwhile improves the efficiency of dynamic checking greatly.

Keywords: concurrent program, data race, happens-before, thread state, lockset

1 Introduction

With the increasing degree of automation in control system, there are more and more demands for capacity in data processing, communication and integration nowadays. Therefore the concurrent multithreaded and multitasking programs have been applied and developed widely. However, randomness of path interleaving brings a lot of uncertainties to the system and even leads to data race. Intuitively data race means that two or more threads access one shared object without the synchronization protecting, while at least an operation of thread is writing. Data race brings a lot of difficulties to designing and testing of program with randomness, uncertainty and unpredictability.

So far there are two typical methods in data race checking. One is Happens-Before Relationship method, and the other is lockset analysis method.

Reference [1] presents the algorithm and theory of static data race checking based on Happens-Before Relationship. References [2,3] make use of Dynamic Partial Order Reduction algorithm to reduce the state space in model of concurrent program. Dynamic model checking can avoid the inconsistencies between executable code (including the compilation, runtime libraries) and models. Therefore dynamic model checking is both sound and complete. However there are always inefficient and high false positive rates in application of such method.

The algorithm of lockset first appeared in Eraser [4]. References [5-7] have presented further optimization for algorithm of lockset. The Reference [8] has applied the lockset method on hardware level. It car find the data race in a very short time. But it can only detect the program on the. The basic idea of lockset is to check whether there is a pair of same lock protecting each shared object accessed by different threads. The lockset algorithm, which can find out the shared object without lock protecting more accurately, overcomes disadvantage of false positives in Happens-Before Relationship. But there still is false positive in lockset algorithm; meanwhile an accurate calculation for locksets requires large amounts of resources by either dynamic or static checking method.

We make use of the method of thread state analysis, conservative lockset analysis to optimize the Happens-Before Relationship, and present a quick dynamic data race checking method which we call DHTC method. Compared with typical Happens-Before relationship and lockset methods, our method has higher efficiency, lower false positive rate, and wider range of platforms applicability.

2 Method description

We use Labeled Transition Systems (LTS) [8] as the basic model for concurrent programs and introduce the Happens-Before Relationship based on such model.

Definition 2.1 LTS is a four-tuple: $M = (S, init, T, R)$, where S is the finite state set of concurrent program, $init(S_0)$ is the initial state, T is the finite set of transitions, and $T \subseteq S \times S$, R is the set of relations of transitions and $R \subseteq T \times T$.

Definition 2.2 parallel combination of LTS, given a concurrent program which has n threads, we use

*^*Corresponding author's*^ e-mail: haorenlianghao@126.com

$Tid = [1,2,\ldots,n]$ to represent the set of threads ID. The LTS of one thread can be written as $M_{tid} = (S_{tid}, init_{tid}, T_{tid}, R_{tid})$, where $tid \in Tid$ is the unique identity of a thread. Such that parallel combination of LTS is

$$M_{||} = M_1 \,||\, M_2 \,||\ldots M_n =< (S_1 \times \ldots \times S_n \times LS_1 \times \ldots \times LS_n \times SS), (s_{01}, s_{02,} \ldots s_{0n}), (T_1 \cup T_2 \cup \ldots \cup T_n), R_{||} > .$$

In the rest of this paper, we will take the $M = (S, s_0, T, R)$ instead of $M_{||}$ to denote the model of concurrent program. A global state $s \in S$ is composed by local states of each thread and the shared states of all shared objects. Threads communicate with each other via shared objects. The operations which access global objects are called visible operations, likewise the operations on local objects are called invisible operations. A transition transforms the model from one state to another by performing one visible operation on global objects.

Definition 2.3 given a global state $s = (s_1, s_2, \ldots s_n)$, if and only if the transition t is a enabled transition on local state s_i in global state s, and local state $s_j = s'_j (i \neq j)$. So that the transition t is enabled at state s, written $t \in s.enabled$ and $s \xrightarrow{t} s'$.

Definition 2.4 $R_i \subseteq T \times T$ is an independent relation, if and only if for each $< t_1, t_2 > \in R_i$, it holds the following two properties:

1) If transition t_1 is enabled in state s, and $s \xrightarrow{t_1} s'$, if and only if transition t_2 is enabled in state s', t_2 is also enabled on state s.

2) If t_1, t_2 are enabled in state s, and there is a unique state s', leading to $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.

Definition 2.5 The Happens-Before relationship (R_H) [9] is a smallest relation between two transitions in a sequence of transitions $\pi = (t_1, t_2, \ldots t_n)$, such that:

1) if $i < j$ and t_i, t_j is dependent then $t_i \underset{\pi}{\rightarrow} t_j$;

2) $\underset{\pi}{\rightarrow}$ relation is a transitively close.

The main idea of the Happens-Before Relationship is to search the state space of the concurrent program, and to check a pair of transitions with Happens-Before Relationship $< t_1, t_2 > \in R_H$. According to [11,12], we present the formal description for data race: there is a data race, if the following three properties hold:

1) Two transitions at a state s are enabled simultaneously, $t_1, t_2 \in s.enabled$;

2) Two transitions are dependent with each other, $< t_1, t_2 > \notin R_i$;

3) Three is at least a transition with write operation.

DPOR (Dynamic Partial-order Reduction) was introduced [1,2]. It can improve the efficiency of space state searching based on Happens-Before Relationship. However, the consumption of dynamic checking remained 30-100 times more than run-time execution, and even higher. Although there is completeness of the theory in data race checking methods based on Happens-Before Relationship. DPOR can find out all data race without false negative, but false positive rate is very high. We will introduce a new method to cut down the false positive rate, and improve de efficiency in next section.

3 Analysis method of thread state and conservative lockset

3.1 THREAD STATE ANALYSIS METHOD

Theory of thread state analysis was put out in Eraser [3]. In the dissertation, run-time states of thread were divided into four states: read own state, write own state, shared access state, and race state. The detailed descriptions of these states are present in Table 1.

TABLE 1 Four states of thread

Name of state	Description of states
read\write own state	After shared object was created by create thread, the shared object is owned by create thread only. At this time there cannot be data race, and shared object is in read\write own states.
Shared access state	After shared object was created by create thread, the shared object is accessed by other threads with only read operations. At this time there cannot be data race, and shared object is in shared access state.
Race state	There is at least one write operation in access thread, so the value of shared object would be changed. Different reading and writing sequence may produce unexpected results. At this time the shared object is in race state, and there may by a data race.

The detail flowchart of thread states transition was presented in Eraser, but Eraser did not take the stat transition after the end of thread into account. It is an important reason to for false positive of lockset analysis. Therefore we add the state transition at end of thread into flowchart of thread states transition in Figure 1.

In FIGURE 1 Flowchart of thread state transition the solid part represented thread states transition descripted in the Eraser, and the dotted part represented thread states transition after the end of thread added by us. $OPR(s)$ and $OPW(s)$ are the transition sets of read and write operations at state s.In order to get the state of transition in thread, we have designed the thread state analysis method in Figure 1.

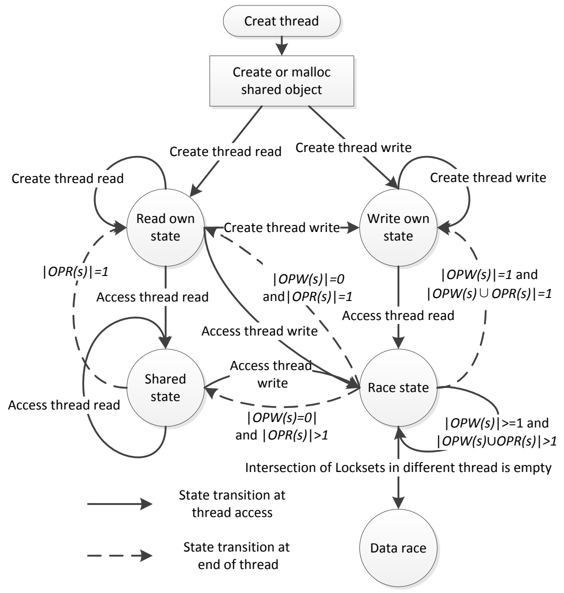

FIGURE 1 Flowchart of thread state transition

In Figure 2 lines 4-11 show that when there are write and read operations at a global state *s* before the end of thread if the thread with write operation is end. The thread state will change to read own state or shared state. If the thread with read operation is end and there is only one thread with write operation. The thread state will change to write own state. If there are at least two threads with more than one write operation still running, the thread state will still keep in race state. From lines 13-22 shows that the state changing during the executing of thread, according position and operation of transition *t*.

```
1.  GetThreadstate(transition $t$){
2.    let $T_\tau = \{t_1, t_2, \dots, t_n\}$ be the consequence of the transitions with thread $\tau = tid(t)$;
3.    let $t_h = t(1 \le h \le n)$;
4.    if($h = n$){
5.      let $t'$ be a transition such that: $tid(t') \ne tid(t_n)$and $t'$is coenabled with $t_n$;
6.      if($\exists t''$: $tid(t) \ne tid(t'')$ and $t''$is coenabled with $t'$)
7.        if($op(t'') = read$ and $op(t')=read$) return $threadshare$;
8.        else if($op(t'') = write$ or $op(t') = write$) return $threadrace$
9.      else If($t'' = null$)
10.       if($op(t') = read$) return $threadreadnly$;
11.       else if($op(t') = write$) return $threadwriteonly$;
12.   }
13.   for ($j = 1, t_j \in T_\tau, j++$){
14.     let $t'$ be a transition such that: $tid(t') \ne tid(t_j)$and $t'$is coenabled with $t_j$;
15.     if ($t' \ne null${
16.       if($op(t') = write$ or $op(t_j) = write$) and $h = j$)   return $threadrace$;
17.       else if($op(t') = read$ and $op(t_j) = read$) and $h = j$)   return $threadshare$;
18.     }
19.     else if ($op(t_j) = op\_write$ and $(\exists t_i(i < j))$, $t_i \in T_\tau$, $op(t_i) = object\_create$)
20.       if ($h \le j$ and)   return $threadwriteonly$;
21.     else if ($op(t_j) = op\_read$ and $(\exists t_i(i < j))$, $t_i \in T_\tau$, $op(t_i) = object\_create$)
22.       if ($h \le j$)   return $threadreadnly$;
23.   }
24.   return $threadrace$;
25. }
```

FIGURE 2 Thread state analysis method

3.2 CONSERVATIVE LOCKSET ANALYSIS METHOD

In classical lockset analysis method, if there is shared object without protecting by a pair of locks, race condition will be triggered. However classical lockset analysis method is still going with large consumption and high false positive rate. Prototype versions of Eraser runs 10-30 times lower efficient than running target purely. The system of HARD makes use of Bloom technology to implement lockset analysis at level of hardware with more efficiency but about 0.89 false positive rates.

In this paper, we propose a conservative dynamic lockset analysis. In typical lockset analysis, given a shared object v accessed by two threads $tid(t_1)$ and $tid(t_2)$, if the sets of locks on shared object *v*, $set_1 = lockset(v, tid(t_1))$, $set_2 = lockset(v, tid(t_2))$ meet $set_1 \cap set_2 \ne \phi$. There is not potential data race condition between $tid(t_1)$ and $tid(t_2)$ as the shared object v is protected by same lock. But it is very difficult to precisely compute the sets of lock during run-time, and it is the main reason for false positive rate. We show a motivating example in Figure 3.

1	Thread_A{	Thread_B{
2	Lock(L1)	Lock(l1);
3	v1++;	v3=v1;
4	Unlock(L1);	Unlock(l1);
5	Lock(l2)	if(v3==0)
6	v2++;	v2++;
7	Unlock(l2)	else
8	}	lock(l2)
9		v2=0;
10		unlock(l2)
11		}

FIGURE 3 Motivating example for data race

If we use the typical lockset analysis method, we will get that $set_1(v2) = lockset(v2, tid(t_1)) = \{L2\}$ and $set_2(v2) = lockset(v2, tid(t_2)) = \phi$, so that $set_1 \cap set_2 = \phi$ at FIGURE 2 line 6. But line 6 in thread B cannot be reachable in practical executing. So it is a false positive. We introduce branch path analysis method into lockset method. Such method instruments the source code to record the operation of different shared object acquiring and releasing locks in every branching path.

Consider the example in Figure 3 again. We build the code like in Figure 4.

```
if(v3==0)
    branch-begin();
    otherbranch_objectaccess(v2,l2,l2);
    otherbranch_locksetupdate(l2,l2);
    v2++;
else
    branch-begin();
    otherbranch_objectaccess(v2,null,null);
    otherbranch_locksetupdate(null,null);
    lock(l2)
        v2=0;
    unlock(l2)
```

FIGURE 4 The instrumenting for branching path

The functions branch-begin() and branch-end() are used to record branching begin and end to inform the scheduler. In function otherbranch_objectaccess ($object$, $Lockset_{acq}$, $Lockset_{rel}$) for each shared $object$, $Lockset_{acq}$ is the set of acquired locks, $Lockset_{rel}$ is the set of released locks. In function otherbranch_locksetupdate($Lockset_{acq}$, $Lockset_{rel}$), $Lockset_{acq}$ is the acquired set of acquired locks and $Lockset_{rel}$ is the set of released locks in the other branching path.

The set of acquired locks is represented as $t.L_{acq}$, the set of released locks is $t.L_{rel}$ at the transition t . We can get the subset of practical acquired set of locks $t.mayl_{acq}$ and $t.mayl_{acq} \subseteq t.L_{acq}$, the superset of practical released set of locks $t.mayl_{rel}$ and $t.mayl_{rel} \supseteq t.L_{rel}$ with the branching path lockset method. Further we will get the subset of practical held set of locks $t.mustlockset$ by $t.mustlockset = t.mayl_{acq} \setminus t.mayl_{rel}$. $t.mustlockset$ is a subset of the set of practical held locks. The conservative lockset analysis method with branching path proposed by us is focused on checking such set.

```
GetLockSet(transition $t_\tau$ ){
    let $T_\tau = \{t_1, t_2, \ldots, t_n\}$ be the consequence of the transitions with thread $\tau$;
    let $t_1.lockset = s.lockset_\tau$;
    for($i = 1, i < n, i++$){
        if ($op(t_i)$ = branch − begin){
            $t_i.lockset = t_{i-1}.loctset$;
            $TempSet.pop(\emptyset)$;
        }
        if ($op(t_i) = lock(L)$) $t_i.lockset = t_{i-1}.loctset \cap \{L\}$;
        else if ($op(t_i) = unlock(L)$) $t_i.lockset = t_{i-1}.loctset \setminus \{L\}$;
        else if ($op(t_i)$ = otherbranch − objectaccess($v, L_a, L_r$)){
            let $t_j (j < i)$ be the last branch − begin that precedes $t_i$;
            $t_j.maylockset = t_j.maylockset \cup \{s_j.lockset \cup L_a \setminus L_r\}$;
        else if ($op(t_i)$ = otherbrach − locksetupdate($L_a, L_r$))
            let $t_j (j < i)$ be the last branch − begin that precedes $t_i$;
            $TempSet.top() = TempSet.top() \cup \{s_j.lockset \cup L_a \setminus L_r\}$;
        else if ($op(t_i)$ = branch − end)
            $t_i.lockset = TempSet.pop() \cup t_{i-1}.loctset$;
        else $t_i.lockset = t_{i-1}.loctset$;
        }
    }
}
```

FIGURE 5 Lockset analysis method with branching path

The light-weight and conservative lockset analysis method is shown in Figure 5. Note that we have not computed the precise held set of locks, but an over-approximated in the method. For a conservative checking, an over-approximated set is sufficient and light-weight. The method first gets the set $s.lockset_\tau$ of thread τ at the state s during the dynamic execution. As this set is computed from actual running information, so it is a precise set. Then we use deep-first method to walk the thread τ which contains the transition t_τ .

4 The overall method

We take the thread state and conservative lockset analysis method with branching path into the Happens-Before method to improve checking efficiency, to reduce the searching state space, and to increase checking accuracy. The improved Happens-Before relationship data race checking method (DHTC) was shown in Figure 6.

```
1.  Statestack $S$; state $s$;
2.  Exploer(){
3.    $S = \emptyset$;
4.    $S.push(s_0)$;
5.    if($s_0.enabled != \emptyset$) UpdateBackTraceset ($s_0$);
6.    while($S != \emptyset$){
7.      $s = S.top()$;
8.      if($\exists q \in s.backtrack \backslash s.done$){
9.        $s.done \leftarrow s.done \cup \{q\}$;
10.       $s.sleep \leftarrow \{t \in s.enabled \mid tid(t) \in s.done\}$;
11.       let $t \in s.enabled$ such that $tid(t) = q$;
12.       let $s'$ be a state such that $s \xrightarrow{t} s'$;
13.       $s'.sleep \leftarrow \{t' \in s.sleep \mid (t,t')$is indepantent$\}$;
14.       $s'.enabled \leftarrow s'.enabled \backslash s'.sleep\}$;
15.       $S.push(s')$;
16.       if(RaceDetect(s)=True) report a Datarce;
17.       if ($s'.enabled = \emptyset$)
18.         for ($i = 1, i < S.zise, i++$) UpdateBackTraceset ($s_i$);
19.       if($\exists \tau \in Tid: t \in s'.enabled$ and $tid(t) = \tau$){
20.         $s'.backtrack \leftarrow \{\tau\}$;
21.         $s'.done \leftarrow \emptyset$;
22.       }
23.     }
24.     else{
25.       $s = S.pop()$;
26.       if ($s.enabled = \emptyset$)
27.         for ($i = 1, i < S.zise, i++$) UpdateBackTraceset ($s_i$);
28.     }
29.   }
30. }
```

FIGURE 6 DHTC data race checking method

The DHTC in Figure 6 is different from DPOR [13,14] based on Happens-Before relationship at 3 points:

1) The sleep set is introduced into DHTC. The *s.sleep* is the set of transitions which are enabled at state *s*, but unnecessary to be executed. The transitions which are independent and have been already executed will be added into *s.sleep* (lines 10, 13). Meanwhile it will take the transitions in *s.sleep* away from *s.enabled*. On one hand the number of backtrack transitions can be reduced, on the other hand the explosion of transition space duo to the infinite loop can be limited with the reducing of set *s.enabled*.

2) The selection of trackback point is different from the DPOR algorithm, which updates the trackback set at each state *s*. The trackback set is computed when the *s'.enabled* is empty line 16 in Figure 6. That is this algorithm computes the trackback set for the whole trace in execution, instant of computing at each state.

3) The rule for computing of trackback transition is different from the DPOR, which add a transition into trackback set *s.backtrack* only following the rule that the transitions are dependent with each other. But in our algorithm, we introduce the thread state and lockset analysis to reduce the trackback transition further in Figure 7.

```
1.  UpdateBackTraceSet (state $s$){
2.    let $\pi$ be the sequence of transitions associated with $S$;
3.    for each thread $\tau \in Tid$ {
4.      let $t_\tau \in s.enabled$   $tid(t_\tau) = \tau$
5.      let $t_d$ be the lasted transition in $\pi$ that is dependent and co_enabled with $t_\tau$;
6.      if($t_d$=$null$) continue;
7.      led $s_d$ be the state in $S$ from which $t_d$ is executed from;
8.      if (LocksetDetectDatarace ($s_d$, $t_d$)= True) continue;
9.      let $E$ be $\{q \in s_d.enabled \mid tid(t) = tid(q)$, or $q$ in $\pi$ , $t_d \underset{\pi}{\rightarrow} q$ and there is $q \underset{\pi}{\rightarrow} tid(t_d)$ $\}$
10.     if ($E != \emptyset$) choose any $q$ in E, add $tid(q)$ to $s_d.baktrack$;
11.     else $s_d.baktrack \leftarrow s_d.baktrack \cup \{tid(q) | s_d.enabled\}$
12.   }
13. }
```

FIGURE 8 Method for computing trackback set

Line 8 in Figure 7 is the function LockSetDetectDatarace(s_d,t_d) is to reduce the trackback transition with thread state and conservative data race checking. It first gets the thread state of current state *s*, according to the thread state analysis in section 3.1. Then it computes the conservative set of locks at the concurrent sate in Figure 8 line 5. *t.maylockset* is the set of locks that may be hold at state *s* and *t.lockset* is the set that must be hold. Line 8 in Figure 8 is judgment of two dependent transitions. Only if the intersection of sets of locks is empty, there is no potential data race at trackback point. So it is unnecessary to add the transition into trackback set.

```
LocksetDetectDatarace(state $s$, transiton $t$){
    let $t_\tau = pre(s,t)$;
    $t_\tau.threadstate$ = GetThreadstate($t_\tau$);
    if($t_\tau.threadstate \neq threadrace$)  return true;
    for each($thead\ \tau$ such that $t \in s.enabled$ and $tid(t) = \tau$)  GetLockSet($s, \tau$);
    for each  ($t \in s.enabled$){
        if ( $\exists t' \in s.enabled$  s.t.  $t'$  is dependent with  $t$  and ( $op(t)$  or  $op(t')$)= $write$)
        if( $(t.lockset \cup t.maylockset) \cap (t'.lockset \cup t'.maylockset) = \emptyset$) return False;
        return True;
    }
}
```

FIGURE 8 LockSetDetectDatarace function

The function DetectDatarace(s) is to check whether there is data race between two transitions at same shared object according to Happens-Before relationship in **Error! Reference source not found.**line 16. Note that such checking is applied after the thread state and conservative lockset analysis. The method in Figure 9 is to check whether there are two dependent transitions on the same shared object with at least a write operation of transition.

```
DetectDatarace($s$){
    if($\exists t, t' : t, t' \in s.enabled$ and $(tid(t') \neq tid(t)$ and $(t, t')$ is dependent){
        if($op(t') = write$  or  $op(t) = write$) return False;
    }
    else if return true;
}
```

FIGURE 9 DetectDatarace function

Data race detecting method based on Happens-Before relationship is both sound and complete [1], but it cannot distinguish between benign and malignant race, so there are many cases of false positive. We introduce the thread state and conservative lockset analysis method into the Happens-Before relationship. The DHTC uses thread state and lockset analysis to check target program conservatively and gets the potential data race state, and finally it uses Happens-Before relationship algorithm to accurately detect data race.

5 Implementation and experiment

In this section, we have conducted experimental comparison of our DHTC with the DPOR algorithm.

5.1 FRAMEWORK OF CHECKING TOOL

The DHTC checking platform consists of program analyzer, thread analyzer and scheduler. All three parts was shown in Figure 10.

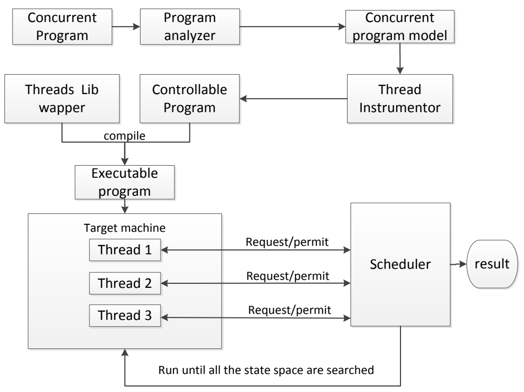

FIGURE 10 Framework for data race checking tool

Given a concurrent program, it first uses program analyzer to build LTS model, then instruments the program with the code on threads and shared objects to register the threads and objects information to scheduler. Instrumented code can communicate with scheduler during runtime. Thereafter it executes the program on the target machine, while scheduler controls threads to be executed or blocked. The platform of dynamic checking tool is shown in Table 2.

TABLE 2 Platform of Dynamic Checking Tool

Platform	Processor	RAM	OS
Scheduler	Intel Core i7 4770	8GB	Windows 7
Target machine (ARM)	Exynos 4412	2GB	RT-Linux
Target machine (x86)	Intel I5 3337	4GB	Linux

5.2 EXPERIMENTAL RESULT

We select a multi-thread coding and communication program on ARM platform, and a launch control program on x86. The launch control program consists of many task threads. We modify the number of threads to verify efficiency of our dynamic checking algorithm. The result is shown in Figure 12.

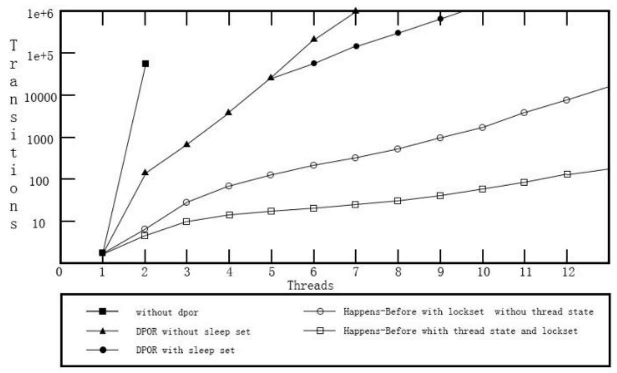

FIGURE 12 Experimental result of launch control program

From Figure 12, we can conclude that Happens-Before relationship with thread state and conservative lockset analysis can largely reduce the state space of concurrent program. Especially with the number of threads more or less growth, the increasing of transitions is not obvious.

In coding and communication program on ARM platform, we compare the DPOR with sleep set with our algorithm to verify the efficiency and Completeness. In Figure 12 "data conflict" means potential data race which have not be confirmed (Table 3).

Table 3 Experimental result for comparison

Threads	SDPOR with sleep				Our Algorithm			
	Transitions	Time (s)	Data conflict	Data race	transitions	Time (s)	Data conflict	Data race
2	0.9k	1.1	26	3	0.7k	1.8	16	2
3	3.7k	5s	103	3	0.71k	1.9	21	2
4	12k	19s	398	6	0.73k	1.9	23	2
5	50k	82s	1640	11	1k	3.7	25	3
6	212k	375s	1479	19	2k	9.9	31	4

The careful readers will find that DPOR with sleep set has higher efficiency than DHTC in only 2 threads situation. The main cause is that DPOR with sleep set needs 145ms while DHTC needs 2460ms in only once execution of data race checking. Since DHTC consists of the Happens-Before relationship, thread state and conservative lockset analysis, it runs much more time than DPOR. So when scale of program is small, the DPOR has higher efficiency. However with increasing of threads, advantage of our algorithm is becoming gradually obviously. Using thread state and conservative lockset analysis makes the number of trackback points largely reduced, and also makes up for the disadvantage of Happens-Before relationship by reducing the false positive rate greatly.

6 Conclusion

We have proposed a new algorithm DHTC for data race checking, which combines the Happens-Before relationship with thread state and conservative lockset analysis. Our method is sound and complete due to the completeness of Happens-Before relationship and lockset algorithm. It greatly reduces the transition space of target program, improves the efficiency of Happen-Before relationship checking method and makes up for the disadvantage of Happens-Before relationship.

References

[1] Wu P, Chen Y, Zhang J 2006 Static data- race detection for multithread programs *Journal of Computer Research and Development* **43**(2) 329-35 *(in Chinese)*

[2] Flanagan C, Godefroid P 2005 Dynamic Partial-order Reduction for Model Checking Software *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 110-21

[3] Yi X, Wang J, Yang X 2008 Stateful Dynamic Partial-Order Reduction *The 8th International Conference on Formal Engineering Methods ICFEM 2006* Macao China 149-67

[4] Savage S, Burrows S, Nelson G, Sobalbarro P, Anderson T 1997 Eraser: a dynamic data race detector for multithreaded programs *ACM Transactions on Computer Systems* **15**(4) 391-411

[5] Zhang L, Zhang F, Wu Y 2003 A lockset-Based Dynamic Data Race Detecting Approach *Chinese Journal of Computer* **26**(10) 1217-23 *(in Chinese)*

[6] Zhang L, Wu S, Zhang F 2004 Data race detection of software Distributed shared memory system *Mini-Micro System* **25**(12) 2070-4 *(in Chinese)*

[7] Zhou P, Teodorescu R, Zhou Y 2007 HARD: hardware-assisted lockset-based race detection *Proceedings of HPCA 2007* IEEE 121-32

[8] Holey A, Mekkat V, Zhai A 2013 HAccRG: Hardware-Accelerated Data Race Detection in GPUs *In International Conference on Parallel Processing – ICPP* 60-9

[9] Liang Z, Luo G, Kuang H 2009 On-the-fly Deadlock Detection with Partial-order Reduction Based on CEGAR *Computer Engineering* **19**(35) 65-8 *(in Chinese)*

[10] Chaki S, Clarke E M, Ouaknine J, Sharygina N 2004 Automated, Compositional and Iterative Deadlock Detection *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-design* 201-10

[11] Lamport L 1978 Time Clocks and the Ordering of Events in a Distributed System *Communications of the ACM* **21**(7): 558-65

[12] Silberschatz A, Galvin P B, Gagne G 2012 Operating Ststem Concepts (9th Edition) *John Wiley & Sons* Inc

[13] Netzer R H B 1991 Race Condition Detection for Debugging Shared-Memory Parallel Programs *University of Wisconsin-Madison*

[14] Yang Y, Chen X, Gopalakrishnan G, Kirby R M 2009 Inspect A Runtime Model Checker for Multithreaded C Programs *School of Computing University of Utah Salt Lake City* UT 84112 USA

[15] Yi X, Wang J, Yang X 2006 Stateful Dynamic Partial-Order Reduction *8th International Conference on Formal Engineering Methods ICFEM 2006* Macao China 149-67

Authors	
	Hao Liang, March 1981, Shanxi, Taiyuan, China. **Current position, grades**: PhD candidate at Academy of Equipment. **Scientific interests**: computer, automation, embedded systems design, real-time embedded systems, and models for complex systems **Publications**: 6
	Yunfeng Ai, September 1979, Shandong, Jinan, China. **University studies**: PhD degree in Control Engineering at Institute of Automation in Chinese Academy of Science. **Scientific interests**: computer, automation, embedded systems design, real-time embedded systems, intelligent transportation systems, intelligent vehicles driver's modeling and behavior analysis.