

# Fast and Accurate Static Data-Race Detection for Concurrent Programs

Vineet Kahlon<sup>1</sup>, Yu Yang<sup>2</sup>, Sriram Sankarnarayanan<sup>1</sup>, and Aarti Gupta<sup>1</sup>

<sup>1</sup> NEC Labs, Princeton, USA.

<sup>2</sup> University of Utah, Salt Lake City, USA.

**Abstract.** We present new techniques for fast, accurate and scalable static race detection in concurrent programs. Focusing our analysis on Linux device drivers allowed us to identify the unique challenges posed by debugging large-scale real-life code and also pinpointed drawbacks in existing race warning generation methods. This motivated the development of new techniques that helped us in improving both the scalability as well as the accuracy of each of the three main steps in a race warning generation system. The first and most crucial step in data race detection is automatic shared variable discovery. Towards that end, we present a new, efficient dataflow algorithm for shared variable detection which is more effective than existing correlation-based techniques that failed to detect shared variables responsible for data races in majority of the drivers in our benchmark suite. Secondly, accuracy of race warning generation strongly hinges on the accuracy of the pointer analysis used to compute aliases for lock pointers. We formulate a new scalable context sensitive alias analysis that effectively combines a divide and conquer strategy with function summarization and is demonstrably more efficient than existing BDD-based techniques. Finally, we provide a new warning reduction technique that leverages lock acquisition patterns to yield provably better warning reduction than existing lockset based methods.

## 1 Introduction

The widespread use of concurrent software in modern day computing systems necessitates the development of effective debugging methodologies for multi-threaded software. Multi-threaded programs, however, are behaviorally complex involving subtle interactions between threads which makes them hard to analyze manually. This motivates the use of automated formal methods to reason about such systems. Particularly notorious to catch are errors arising out of data race violations. A data race occurs when two different threads in a given program can simultaneously access a shared variable, with at least one of the accesses being a write operation. Checking for data races is often a critical first step in the debugging of concurrent programs. Indeed, the presence of data races in a program typically renders its behavior non-deterministic thereby making it difficult to reason about it for more complex and interesting properties.

In this paper, we develop techniques for data race detection that are efficient, scalable and accurate. In order to identify the practical challenges posed by the debugging of large-scale real-life code, we focused our analysis on detecting data races in Linux device drivers. A careful study of bug reports and CVS logs at `kernel.org` revealed that the two main reasons for the presence of data races in drivers are incorrect locking and timing related issues. Since timing related data races are hard to analyze at the software level, we chose to focus only on locking related bugs.

The classical approach to data race detection involves three steps. The first and most critical step is the automatic discovery of shared variables, i.e., variables which can be accessed by two or more threads. Control locations where these shared variable are read or written determine potential locations for data races to arise. In fact, locking related data races arise if a common shared variable is accessed at simultaneously reachable program locations in two different threads where disjoint sets of locks are held. Since locks are typically accessed via pointers, in order to determine these locksets at program locations of interest, in the second step, a must-pointer alias analysis is carried out. Finally, the main drawback of static analysis is that a large number of bogus data race warnings can often be generated which do not correspond to true bugs. The last step, therefore, is to use warning reduction and ranking techniques in order to either filter out bogus warnings or use ranking to prioritize them based on the degree of confidence.

The challenge lies in carrying out race detection while satisfying the conflicting goals of scalability and accuracy both of which depend on various factors. Key among them are (i) accuracy of shared variable discovery, and (ii) accuracy and scalability of the alias analyses for determining shared variables (must aliases) and locksets (may aliases). Wrongly labeling a variable as shared renders all warnings generated for it bogus. On the other hand, if we miss reporting a variable as shared then we fail to generate warnings for a genuine data race involving this variable.

Considerable research have been devoted to automatic shared variable discovery [7, 14]. However, most existing techniques are based on the underlying assumption that when accessing shared variables concurrent programs almost always follow a *locking discipline* by associating with each shared  $v$  variable a lock  $l_v$  which needs to be acquired before any access to  $v$ . Existing techniques focus on computing this association between locks and variables. Towards that end, various correlation based techniques have been developed – both statistical [7] and constraint based [14]. An advantage of statistical techniques is that they are scalable and do not depend on an alias analysis which can often be a bottleneck. However, the failure of correlation based techniques to detect the shared variable responsible for data races in our suite of Linux drivers exposed the fact that their main weakness turns out to be this very reliance on the existence of locking discipline. Indeed, many data races arise precisely when the locking discipline is violated. Furthermore, it turns out that in most of the drivers that we considered, the original implementations correctly followed lock discipline. Data race bugs were introduced only when the programs were later modified by adding new code either for optimization purposes or in order to fix bugs. Typically, this newly added code was a “hack” that introduced lock-free accesses to shared variables that weren’t present in the original code. Since the only occurrences of these variables were in regions unguarded by locks, no meaningful correlations could be developed for them and was the key reason why correlation-based techniques [7, 14] didn’t work.

In order to ensure that no shared variable fails detection, we use a very liberal criterion to categorize variables as shared. Our shared variable detection routine is based on the premise that all shared variables are either global variables of threads, aliases thereof, pointers passed as parameters to API functions or escape variables. Furthermore, we are only interested in identifying precisely the subset of variables from the above set that are written to in the given program as only these can participate in a data

race. The main challenge here is that since global variables can be accessed via local pointers we need to track aliasing assignments leading to such local pointers. An additional complication is that not all assignments to aliases of global variables result in meaningful updates to global variables. Indeed, in a sequence of pointer assignments  $p_1 = p, \dots, q = p_k$ , starting at a pointer  $p$  to a global structure  $S$ , say, we see that assignments in the above sequence merely pass aliasing information without updating the value of any (scalar) variable. If, however, the above sequence is followed by an assignment of the form  $p \rightarrow f = exp$  to a field  $f$  of  $S$ , then it is a genuine update to  $f$  thus making it a variable of interest. We show that such update sequences can be detected via an efficient dataflow analysis. In fact, in most Linux drivers, data global to a thread is usually stored as global structures having a large number number of fields – typically 50 to 100. Only a small fraction of which are actually used for storing shared data which our new algorithm was able to isolate with high precision. Declaring all the field of a global structure as shared would simply generate too many bogus warnings.

The second step in static race detection is to accurately determine locksets at program locations where shared variables are accessed. Since locks are typically accessed via pointers, this requires computation of must-aliases of these lock pointers. The accuracy of warning generation is therefore directly dependent on the accuracy of the must-alias pointer analysis. Moreover, for the sake of accuracy it is imperative that lock aliases be computed context sensitively. This is because most must-aliases in C arise from parameter passing of pointer arguments in functions, which alias to different pointers in different contexts. The result is that a context sensitive alias analysis produces drastically lesser bogus warnings than a context insensitive one. However, the key drawback of a context sensitive analysis is scalability as the number of possible contexts in a large program can easily explode. In recent years, considerable research has been devoted to ameliorating this problem by storing contexts symbolically using data structures like BDDs. Implementation of BDD-based context sensitive pointer analysis like BDDBDDDB [18] have been shown to give good results for Java programs [13, 12]. However, C programs, which are less structured than Java programs, typically have too many pointer variables and complex aliasing relations between them which become hard to handle using BDDBDDDB as the program size grows. This was borne out by our experiments with a downloaded version of BDDBDDDB. We therefore propose a new technique for scalable context sensitive pointer analysis that combines:

(i) **Divide and Conquer** which leverages the fact that we can partition the set of all pointers in a program into disjoint classes such that each pointer can only alias to a pointer within its class. While, in general, aliasing is not an equivalence relation, many widely used pointer analyses like Steensgaard[16] generate equivalence relations that are over-approximations of aliasing. Since we use this initial pointer analysis only for partitioning, scalability is more critical than accuracy and this is precisely what Steensgaard’s analysis offers. There are two important consequences of this partitioning. First, since we are only interested in lock pointers, and since lock pointers can only alias to other lock pointers, we can ignore non-lock pointers. This drastically cuts down on the number of pointers we need to consider for our analysis. Secondly, since a given lock pointer can, in general, be aliased to a small subset of the total set of lock pointers, Steensgaard analysis provides us with a further decomposition of the set of lock pointers

into yet smaller partitions. A second and more accurate context-sensitive alias analysis is then carried out on these final partitions and even though expensive in general, it becomes scalable on these small classes.

(ii) **Procedure Summarization** which exploits locality of reference, viz., the fact that locks and shared variables are accessed in a small fraction of functions. Our new summarization based must alias analysis procedure therefore needs to compute summaries only for these small number of functions thereby making our approach applicable to large programs. We emphasize that procedure summarization is extremely important in making any static analyses scalable. Indeed, typical real-life code has a large number of small functions that can be called from many different contexts. A non-summarization based technique like BDDBDDDB can easily be overwhelmed as the program size grows. It is important to note that it is the synergy resulting by combining the two techniques that enables us to achieve scalability. Indeed, it is divide and conquer which allows us to exploit locality of reference making summarization viable.

Finally, one of the main weaknesses of using static race detection techniques is that a large number of (bogus) race warnings can often be generated. In this paper, we show that tracking lock acquisition patterns, instead of locksets, results in a warning reduction technique that is more accurate than the existing lockset based warning reduction technique [8] in two ways. First, by leveraging acquisition histories in addition to locksets we can filter out warnings generated by lockset based technique at the warning generation stage itself. Secondly, once the warnings are generated, we can use a dominator-based technique that leverage acquisition histories to give provably better warning reduction than [8]. Additionally, by using ranking, we can guarantee that our reduction technique is sound, viz., will not drop real data races in favor of bogus ones.

## 2 Shared Variable Discovery

So as not to miss any shared variable we use a very liberal definition of when a variable is declared as such. Essentially, we are interested in all *genuine* modifications to global variables, aliases thereof, pointer passed as parameters to API functions and escape variables. A global variable of a thread that is directly written to is declared as shared. Such variables can determined merely by scanning through each program statement of the given driver. However, a global variable may also be accessed via a local pointer. Such a pointer  $q$  could occur at the end of a chain of pointer assignments  $p_1 = p, p_2 = p_1, \dots, q = p_k$  starting at a pointer  $p$  to, say, a global structure  $S$ , which is either global or passed as a parameter to an api function. Then any variable  $v$  modified via an access through  $p$  is also a variable of interest. However, simply declaring all pointers occurring in such sequence as shared could lead to a lot of bogus warnings. Indeed, in the above assignment sequence, the assignments are not genuine updates but merely serve to propagate the value of fields  $S$ . If, however, the above sequence is followed by an assignment of the form  $p_{k+1} \rightarrow f = exp$ , where  $exp$  is either a local variable or an expression other than simple propagation of a data value, it is a genuine update and should be declared a shared variable of interest.

**Definition 1.** A sequence of assignments to pointers  $p_1 = p, p_2 = p_1, \dots, q = p_k$  is called a complete update sequence from  $p$  to  $q$  iff for each  $i$ , there do not exist any assignments to pointers  $p_i$  after it is written and before it is read in the sequence.

Thus our goal is to detect complete update sequences from  $p$  to  $q$  that are followed by the modification of a scalar variable accessed via  $q$ , where  $p$  either points to a global variable or is passed as a parameter to an API function. We determine such sequences using our new interprocedural dataflow analysis formulated as algorithm 1. Essentially the procedure propagates the assignments in complete update sequences as discussed above till it hits a genuine update to a locally global variable which it declares as shared. The algorithm keeps track of the potentially shared variable as the set  $P_{sh}$ . Given a thread  $T$ , to start with  $P_{sh}$  contains variables of  $T$  that are pointers to global variables or passed as parameters to API functions. A separate variable  $V_{sh}$  keeps track of variables involving genuine updates which are therefore shared. Each assignment of the form  $v = w$  is a propagation if  $w \in P_{sh}$ . Thus if  $v \notin P_{sh}$  it is added to  $P_{sh}$  (lines 4-6). A variable  $v \in P_{sh}$  is included in  $V_{sh}$  only if there is an assignment of the form  $v = exp$ , where  $exp$  is a genuine update as discussed above (lines 7-12).

---

**Algorithm 1 Dataflow Analysis for Shared Variable Detection**

---

```

1: Initialize  $V_{sh} = \emptyset$ ,  $P_{sh}$  to the global variables of thread  $T$  and  $S$  to the entry statement of  $T$ .
2: repeat
3:   for each statement  $st \in S$  do
4:     if  $st$  is of the form  $v = w$  where  $v$  and  $w$  are variables then
5:       if  $w \in P_{sh}$  then
6:         set  $P_{sh} = P_{sh} \cup \{v\}$ 
7:       else if  $v \in P_{sh}$  but  $w \notin P_{sh}$  then
8:         set  $V_{sh} = V_{sh} \cup \{v\}$ .
9:       end if
10:    else if  $st$  is of the form  $v = exp$  where  $exp$  is an expression other than a simple
        variable then
11:      if  $v \in P_{sh}$  then
12:        set  $V_{sh} = V_{sh} \cup \{v\}$ .
13:      end if
14:    end if
15:    Add each successor statement  $st'$  of  $st$  not already processed to  $S$ .
16:  end for
17: until  $S$  is empty
18: return  $V_{sh}$ 

```

---

### 3 Scalable Context Sensitive Alias Analysis

As noted in the introduction, once the shared variables have been identified, the key bottleneck in generating accurate lockset based warnings is a scalable context-sensitive must alias analysis to determine locksets at control locations where shared variables are accessed. In this section, we propose a new technique for scalable context sensitive alias analysis that is based on effectively combining a divide and conquer strategy with function summarization in order to leverage the benefits of both techniques.

**Divide and Conquer via Partitioning.** We exploit the fact that, even though aliasing is not, in general, an equivalence relation, many alias analyses like Steensgaard’s compute relations that are over-approximations of aliasing but are, importantly, equivalence

relations. Additionally, an equally critical feature of Steensgaard’s analysis is that it is highly scalable. This makes it ideally suitable for our purpose which is to partition the set of all the pointers in the program into disjoint classes that respect the aliasing relation, i.e., a pointer can only be aliased to pointers within the class to which it belongs. A drawback of Steensgaard’s analysis is lack of precision. However, this can be achieved by then focusing a more refined analysis on each individual Steensgaard partition. Indeed partitioning, in effect, decomposes the pointer analysis problem into much smaller sub-problems where instead of carrying out the pointer analysis for all the pointers in the program, it suffices to carry out separate pointer analyses for each equivalence class. The fact that the partitioning respects the aliasing relation guarantees that we will not miss any aliases. The small size of each partition then offsets the higher computational complexity of a more precise analysis. As noted in the introduction, the Steensgaard generated equivalence class for a lock pointer typically contains only a small subset of lock pointers of the given program thus ensuring scalability of a context-sensitive alias analysis on each such partition.

**Exploiting Locality of Reference via Summarization.** Using decomposition, once the set of pointers under consideration have been restricted to small sets of lock pointers, we can further exploit locality of reference which then allows us to effectively leverage procedure summarization for scalable context sensitive pointer analysis. Indeed, typically in real-life programs, shared variables, and as a consequence locks, are accessed in a very small number of functions. Thus instead of following the BDDDBDD approach that pre-computes aliases for all pointers in all contexts, it is much more scalable to instead use procedure summarization to capture all possible effects of executing a procedure on lock pointers. The reason it is more scalable is that we need to compute these summaries only for the small fraction of functions in which lock pointers are accessed. Once we have pre-computed the summaries, the aliases for a lock pointer at a program location in a given context can be generated efficiently on demand. We emphasize that it is the above decomposition that allows us to leverage locality of reference. Indeed, without decomposition we would have to compute summaries for each function with a pointer access, viz., practically every function in the given program. Additionally, for each function we would need to compute the summary for all pointers modified in the function not merely the lock pointers which could greatly increase the termination time of the algorithm. Thus by combining divide and conquer with summarization we can exploit the synergy between the two techniques.

**Computing Procedure Summaries for Context-Sensitive Pointer Analysis.** We propose a new summarization based technique for demand driven context sensitive pointer analysis. Given a context, viz., a sequence of function calls,  $con = f_1 \dots f_n$ , pointers  $p$  and  $p'$  are aliased in  $con$  iff there is a complete update sequence  $p_1 = p', p_2 = p_1, \dots, p_m = p$  during the execution of the functions in  $con$

To compute the must-alias set of a pointer  $p$  at control location  $l$  of function  $f_n$  in context  $con$ , we first compute the set  $A_n$  of pointers such that for each  $q \in A_n$ , there is a complete update sequence from  $q$  to  $p$  along each path starting at the entry location of function  $f_n$  to control location  $l$ . Having obtained  $A_n$ , we start traversing backwards along the sequence of functions in  $con$ . This involves computing for each pointer  $q \in A_n$ , the set  $A_{n-1}^q$ , of all pointers  $r$  such that there is a complete assignment sequence

from  $r$  to  $q$  along each path starting from the entry location of  $f_{n-1}$  to the location where  $f_n$  was called. Set  $A_{n-1} = \bigcup_{q \in A_n} A_n^q$ . Then repeating the above procedure for each function in  $con$  till we reach the root function  $f_1$  gives us the must-aliases of  $p$ . The key step, therefore, is to compute summaries that capture the effect of each function on lock pointers.

**Definition 2.** *The summary for a function  $f$  in a given program is the set of all tuples of the form  $(p, l, A)$ , where  $p$  is a pointer written to at location  $l$  in  $f$  and  $A$  is set of all pointers  $q$  such that there is a complete update sequence from  $q$  to  $p$  along each path starting at the entry location of  $f$  and ending at  $l$ . The set  $A$  is denoted by  $Sum(f, p, l)$ .*

As an example, consider the program in figure 3 with global pointers  $p$  and  $q$ . First, since no pointer is modified in `bar`, its summary is empty. Also  $g_3 \in Sum(goo, 2c, p)$  and  $g_4 \in Sum(goo, 2c, q)$ . Similarly,  $g_4 \in Sum(goo, 5c, q)$  but  $g_5 \notin Sum(goo, 5c, p)$ . This is because the control flow branches at location `3c` with  $p$  being set to  $g_5$  in one branch and retaining the old value  $g_3$  in the other. Statically, there is no way of deciding whether  $g_3$  and  $g_5$  are the same pointer. Thus  $Sum(goo, 5c, p) = \emptyset$ . Thus,  $Sum(foo, 2a, p) = \{g_1\}$  and  $Sum(foo, 2a, q) = \{g_2\}$ , whereas  $Sum(foo, 3a, p) = \emptyset$  and  $Sum(foo, 3a, q) = \{g_4\}$ .

```

foo(){
  1a: p = g1;
  2a: q = g2;
  3a: bar();
  4a: ... ;
}

bar(){
  1b: goo();
}

goo(){
  1c: p = g3;
  2c: q = g4;
  3c: if(global_var)
  4c:   p = g5;
  5c: u = 1 ;
}

```

**Fig. 1.** An Example Program

Note that we do not need to cache the summary tuples for each program location of a function. Indeed, for function  $f_i$ , where  $1 \leq i \leq n - 1$ , all we need are the summary tuples for the locations where  $f_{i+1}$  is called. In addition, we also need to cache the summary tuple for the exit location as it might be required while performing the dataflow analysis. For the last function  $f_n$  in  $con$ , we need the summary tuples for each location in the function where a lock pointer is accessed. Since the number of such locations are typically few, the sizes of the resulting summaries are small.

**The Algorithm.** Given a pointer  $p$  and location  $l$  in function  $f$ , we perform a backward traversal on the CFG of the given program starting at  $l$  and track the complete update sequences as tuples of the form  $(m, A)$ , where  $m$  is a program location and  $A$  is a set of lock pointers  $q$  such that there is a complete update sequence from  $q$  to  $p$  starting from  $m$  and ending at  $l$ . The algorithm maintains a set  $W$  of tuples that are yet to be processed and a set  $P$  of tuples already processed. Initially  $W$  contains the tuple  $(l, \{p\})$  (line 2). Note that before processing a tuple  $(m, A)$  from  $W$ , since our goal is to compute must-aliases we need to make sure that each successor  $m'$  of  $m$  from which there exists a path in the CFG leading to  $l$  has already been processed during the backward traversal, viz.,  $W$  currently has no tuples of the form  $(m', D)$ . Such a tuple is called *ready* (line 4). If the statement at  $m$  is of the form  $r = t$ , where  $t \in A$ , then in processing  $(m, A)$ , let  $A'$  be the set that we get from  $A$  by replacing  $t$  with  $r$  else  $A' = A$  (lines 5-7).

---

**Algorithm 2 Summary Computation for Lock Pointer Analysis**

---

```
1: Input: Lock Pointer:  $p$ , Control Location  $l$ , Function  $f$ .
2: Initialize  $W$  to  $(l, \{p\})$  and  $P$  to  $\emptyset$ .
3: repeat
4:   Remove a ready tuple  $tup = (m, A)$  from  $W$ . Set  $A' = A$ .
5:   if lock pointer  $t \in A$  and the statement at location  $m$  is of the form  $r = t$  then
6:      $A' = (A \setminus \{t\}) \cup \{r\}$ 
7:   end if
8:   if  $m$  is the entry location of function  $f$  add  $(p, A)$  to the summary then
9:     return
10:  else if  $m$  is the call return site of a function call for  $g$  then
11:    if the summary tuples have already been computed for all lock pointers in  $A'$  for the
    exit location  $exit_g$  of  $g$  then
12:       $B = \bigcup_{t \in A'} Sum(g, exit_g, t)$ , where  $Sum(g, exit_g, t)$  is the summary of pointer  $t$ 
    with respect to  $exit_g$  if  $t$  is written to in  $g$  else it is  $t$ 
13:      Let  $NewTuples = \{(m', B)\}$ , where  $m'$  is the call site of  $g$  corresponding to  $m$ 
14:    else
15:      Add  $(exit_g, A')$  to  $W$ 
16:    end if
17:  else
18:     $NewTuples = \bigcup_{m' \in Pred} \{(m', A')\}$ , where  $Pred$  is the set of predecessors of  $m$ 
19:  end if
20:  for each tuple  $(l, B) \in NewTuples$  that has not already been processed do
21:    if there exists a tuple of the form  $(l, C)$  in  $W$  then
22:      replace  $(l, C)$  by  $(l, C \cap B)$ 
23:    else
24:      Add  $(l, B)$  to  $W$ 
25:    end if
26:  end for
27: until  $W$  is empty
```

---

In order to propagate the pointers in  $A'$  backwards, there are two cases to consider. First, assume that  $m$  is a return site of a function  $g$  that was called from within  $f$ . Then we have to propagate the effect of executing  $g$  backwards on each pointer in  $A'$ . Towards that end, we first check whether the summary tuples for  $g$  have already been computed for each of the pointers in  $A'$  for the exit location  $exit_g$  of  $g$ . If they have, then we form the new tuple  $(m', B)$ , where  $m'$  is the call site of  $g$  corresponding to the return site  $m$  and  $B = \bigcup_{r \in A'} Sum(g, r, exit_g)$  (lines 11-13). If, on the other hand, the summary tuples have not been computed, we introduce the new tuple  $(exit_g, A')$  in the worklist (line 15). For the second case, we assume that,  $m$  is not a function call return site, we consider the set  $Pred$  of all the predecessor locations of  $m$  in  $f$  (line 18). For each  $m' \in Pred$ , we form the tuple  $(m', A)$ . If tuple  $(m', A)$  has already been processed no action is required. Else, if there already exists a tuple of the form  $(m', C)$  in  $W$ , then we have discovered a new backward path to location  $m'$ . Since we are computing must aliases, viz., intersection of aliases discovered along all backwards CFG paths, we

replace the tuple  $(m', C)$  with the tuple  $(m', A \cap C)$  (line 22). If there exists no such tuple, then we simply add the new tuple  $(m', A)$  to  $W$ .

#### 4 Leveraging Acquisition Histories for Warning Reduction

We present two new race warning reduction techniques that are based on tracking lock acquisition patterns and are provably more accurate than the existing lockset-based warning reduction techniques [8]. Our new reduction technique proceeds in two stages. In the first stage, we make use of the notion of consistency of lock acquisition histories which governs whether program locations in two different threads are simultaneously reachable. This allows us to discard, in a sound fashion, those warnings wherein lock acquisition histories are inconsistent even though disjoint locks are held at the corresponding program locations. Lockset based techniques could not remove such warnings. In the second stage, we use yet another warning reduction technique complementary to the first one which is based on defining an acquisition history based weaker than relation on the remaining warnings that is more refined than the lockset based weaker than relation defined in [8]. Our strategy improves upon [8] in two ways (i) by leveraging inconsistency of acquisition histories to remove warnings in a sound fashion, and (ii) by providing a provably more refined weaker than relation.

The lockset based *weaker than relation* technique of [8] defines an *access event* as a 4-tuple of the form  $(v, T, L, a, c)$ , where  $v$  is a shared variable accessed at control location  $c$  of thread  $T$  with lockset  $L$  and  $a$  denotes the type of accesses, i.e., whether it is a read or a write. Let  $e_1, e_2$  and  $e_3$  be access events such that  $e_2$  and  $e_3$  occur along same local computation path of a thread. Then if the occurrence of a race between  $e_1$  and  $e_2$  implies the occurrence of a race between  $e_1$  and  $e_3$ , we need not generate a warning for the pair  $(e_1, e_2)$ . In this case, the event  $e_3$  is said to be *weaker than*  $e_2$ , denoted by  $e_3 \sqsubseteq e_2$ . The relation  $\sqsubseteq$  is hard to determine precisely without exploring the state space of the program which is inefficient. Instead, it is typically over-approximated via static analysis. A lockset based approximation,  $\sqsubseteq_l$ , given in [8] is defined below.

**Definition 3. (Lockset Based Weaker Than [8])** For access event  $p = (v, T, L_1, a_1, c_1)$  occurring before access event  $q = (v, T, L_2, a_2, c_2)$  along a common local computation  $x$  of thread  $T$ ,  $p \sqsubseteq_l q$  iff  $L_1 \subseteq L_2$  and either  $a_1 = a_2$  or  $a_1$  is a write operation.

**Acquisition History based Warning Reduction.** We motivate our technique with the help of a simple concurrent program  $\mathcal{CP}$  comprised of the two threads  $T_1$  and  $T_2$  shown in figure 2 that access shared variable  $x$ . Let  $e_1, e_2, e_3$  and  $e_4$  denote accesses to  $x$  at locations 6a, 6b, 9b and 2b, respectively. Note that the locksets at control locations 6b and 9b are  $L_2 = \{lk2\}$  and  $L_3 = \{lk2\}$ , respectively. Since  $L_2 \subseteq L_3$ ,  $e_2 \sqsubseteq_l e_3$  and so the lockset based reduction technique would drop  $(e_1, e_3)$  in favor of  $(e_1, e_2)$ .

However, control locations 6a and 6b are not simultaneously reachable whereas 6a and 9b are, even though disjoint locksets are held at the two locations in either case. The key reason is that simultaneous reachability of two control locations in separate threads depends not merely on the locks held at these locations but also on the patterns in which they were acquired in the individual threads. Indeed, in order for  $T_2$  to reach 6b it needs to execute the statements at locations 3b and 5b, viz., acquire and release lock  $lk1$ . Note, however, that once  $T_1$  acquires  $lk1$  at location 2a it does not release it until after it has exited 6a. Thus in order for the two threads to simultaneously reach

	1b: lock(lk2);
1a: a = 1;	2b: x = 0;
2a: lock(lk1);	3b: lock(lk1);
3a: lock(lk2);	4b: b = 2;
4a: y = 1;	5b: unlock(lk1);
5a: unlock(lk2);	6b: x = 2;
6a: x = 3;	7b: unlock(lk2);
7a: unlock(lk1);	8b: lock(lk2);
	9b: x = 1;

**Fig. 2.** Threads  $T_1$  and  $T_2$  with shared variable  $x$

6a and 6b,  $T_2$  must first acquire and release  $lk1$ , viz., must already have executed 5b before  $T_1$  executes 2a. However, in that case  $T_2$  holds lock  $lk2$  (via execution of 1b) which it cannot release, thus preventing  $T_2$  from executing 3a and transiting further. This proves our claim. The simultaneous reachability of 6a and 9b, on the other hand, is easy to check. Thus the  $\sqsubseteq_l$ -based reduction of [8] drops a warning corresponding to a real data race in favor of a bogus one. In general, when testing for reachability of control states  $c$  and  $c'$  of two different threads we need to test whether there exist paths  $x$  and  $y$  in the individual threads leading to states  $c$  and  $c'$  holding lock sets  $L$  and  $L'$  which can be acquired in a compatible fashion so as to prevent the scenario above. Compatibility can be captured using the notion of acquisition histories.

**Definition 4 (Acquisition History) [9]** *Let  $x$  be a global computation of a concurrent program  $\mathcal{CP}$  leading to global configuration  $c$ . Then for thread  $T_i$  and lock  $l$  of  $\mathcal{CP}$  such that  $l \in \text{Lock-Set}(T_i, c)$ , we define  $AH(T_i, l, x)$  to be the set of locks that were acquired (and possibly released) by  $T_i$  after the last acquisition of  $l$  by  $T_i$  along  $x$ .*

If  $L$  is the set of locks, each acquisition history  $AH$  is a map  $L \rightarrow 2^L$  associating which each lock a lockset – the acquisition history of that lock. We say that  $AH_1$  and  $AH_2$  are *consistent* iff there do not exist locks  $l_1$  and  $l_2$ ,  $l_1 \in AH_2(l_2)$  and  $l_2 \in AH_1(l_1)$ . Then the above discussion can be formalized as follows.

**Theorem 5 (Decomposition Result) [9]** *Let  $\mathcal{CP}$  be a concurrent program comprised of the two threads  $T_1$  and  $T_2$ . Then for control states  $a_1$  and  $b_2$  of  $T_1$  and  $T_2$ , respectively,  $a_1$  and  $b_2$  are simultaneously reachable only if there are local computations  $x$  and  $y$  of threads  $T_1$  and  $T_2$  leading to control states  $a_1$  and  $b_2$ , respectively, such that (i) locksets at  $T_1$  and  $T_2$  at  $a_1$  and  $b_2$ , respectively, are disjoint, and (ii) the acquisition histories  $AH_1$  and  $AH_2$  at  $a_1$  and  $b_2$ , respectively, are consistent. If the threads communicate solely via nested locks then the above conditions are also sufficient.*

These acquisition histories can be tracked via static analysis much like locksets. To leverage the Decomposition result, we therefore define an **ah**-augmented access event as a tuple of the form  $(v, T, L, AH, a, c)$ , where  $(v, T, L, a, c)$  is an access event and  $AH$  is the current acquisition history. Our warning reduction proceeds in two stages.

**Stage I.** Since consistency of acquisition histories is a necessary condition for simultaneous reachability, we drop all warnings  $(e_1, e_2)$ , where  $e_i = (v, T, L_i, AH_i, a_i)$  and  $AH_1$  and  $AH_2$  are inconsistent. In our example,  $(e_1, e_3)$  will be dropped at this stage.

**Stage II.** On the remaining warnings, we impose a new acquisition history based weaker than relation  $\sqsubseteq_a$ . Towards that end, given two acquisition histories  $AH_1$  and  $AH_2$ , we say that  $AH_1 \sqsubseteq AH_2$  iff for each lock  $l$ ,  $AH_1(l) \subseteq AH_2(l)$ . An immediate but important consequence is the following

**Proposition 1.** *Given acquisition history tuples  $AH$ ,  $AH_1$  and  $AH_2$ , such that  $AH_1 \sqsubseteq AH_2$ ,  $AH$  is consistent with  $AH_2$  implies that  $AH$  is consistent with  $AH_1$ .*

**Definition (Acquisition History based Weaker Than).** *For access event  $e_1 = (v, T, L_1, AH_1, a_1, c_1)$  occurring before  $e_2 = (v, T, L_2, AH_2, a_2, c_2)$  along a common computation of thread  $T$ ,  $e_1 \sqsubseteq_a e_2$  iff  $L_1 \subseteq L_2$ ,  $AH_1 \sqsubseteq AH_2$  and either  $a_1 = a_2$  or  $a_1$  is a write operation.*

In our example, the acquisition histories for events  $e_1, e_3$  and  $e_4$  are  $AH_1 = \{(lk1, \{lk2\})\}$ ,  $AH_3 = \{(lk2, \emptyset)\}$  and  $AH_4 = \{(lk2, \emptyset)\}$ , respectively. Clearly,  $AH_4 \sqsubseteq AH_3$ , and so  $e_4 \sqsubseteq_a e_3$ . Thus we drop  $(e_1, e_3)$  and retain  $(e_1, e_4)$ . The intuition behind this is that any local computation of  $T_2$  leading to accesses  $e_3$  has to pass through the access  $e_4$ . Moreover, since  $AH_3 \sqsubseteq AH_4$ , it follows that if  $AH_1$  and  $AH_3$  are consistent then so are  $AH_1$  and  $AH_4$ . Thus, since  $T_1$  and  $T_2$  communicate only via nested locks, by the decomposition result, if there is a computation realizing the data race corresponding to the warning  $(e_1, e_3)$ , then there also exists one realizing  $(e_1, e_4)$ . Thus we may drop  $(e_1, e_3)$  in favor of  $(e_1, e_4)$ .

**Acquisition History-based Covers.** Note that in general there might be multiple paths leading to an access event  $e_k$ , in which case before dropping a pair  $(e_i, e_k)$ , we need to make sure that along each path in the program leading to  $e_k$  there exists an access event  $e_j \sqsubseteq_a e_k$ . This can be accomplished by using the notion of a *cover* for an access event. Given an access event  $e$ , a *Cover* for  $e$  is a set of access events  $c$  such that  $c \sqsubseteq_a e$ . Such a cover can be easily determined via a backwards dataflow analysis from the program location corresponding to  $e$ .

**Making Reduction Sound via Ranking.** Finally, we note that if the thread synchronization is not merely lock based, a reduction strategy based on either  $\sqsubseteq_a$  or  $\sqsubseteq_l$  is not sound. In [8], a manual inspection routine is proposed in order to identify real warnings that may have been dropped which may not be practical. We propose using ranking in order to ensure soundness. Towards that end, we do not drop any warning based on  $\sqsubseteq_a$  but merely ranks them lower. Then whether a warning lower in the order is inspected is contingent on the fact that the warning higher in the order turns out to be a bogus one.

## 5 Experimental Results

The experimental results for our suite of 10 Linux device drivers downloaded from `kernel.org` are tabulated below. The results clearly demonstrate (i) the effectiveness of our shared variable discovery routine, (ii) the scalability and efficiency (*Time* column) of our new summary based pointer analysis, and (iii) the effectiveness and hence the importance of leveraging warning reduction techniques. The `BDDBDDB` and *Time* columns refer to the times taken (not including the time taken for building the CFG - typically less than a minute) when using, respectively, the `BDDBDDB` engine or our new summary based technique for must-alias analysis. The improvement in efficiency is clearly evident. The `BDDBDDB` engine was run only on the first three

drivers. The columns labeled *War* and *Aft. Red.* refer, respectively, to the total number of warnings generated originally and after applying reduction based on the  $\sqsubseteq_a$  relation. Even after applying these reductions, there could still be a lot of warnings generated as Linux drivers usually have a large number of small functions resulting a large number of contexts. Thus the same program location may generate many warnings that result from essentially the same data race but different contexts. The column *Aft. Con.* refers to the number of warnings left after generating only one warning for each program location and abstracting out the contexts.

Driver	KLOC	# <i>ShVars</i>	#War	#Aft.Red.	#Aft.Con.	BDDDBDD	Time(secs)
hugetlb	1.2	5	4	1	1	15min	3.2
ipoib_multicast	26.1	10	33228	6	6	>1hr	7
plip_test	13.7	17	94	51	51	>30min	5
sock	0.9	6	32	21	13	-	2
ctrace_comb	1.4	19	985	218	58	-	6.7
autofs_expire	8.3	7	20	8	3	-	6
ptrace	15.4	3	9	1	1	-	15
tty_io	17.8	1	6	3	3	-	4
raid	17.2	6	23	21	13	-	1.5
pci_gart	0.6	1	3	1	1	-	1

## 6 Conclusion and Related Work

Data race detection being a problem of fundamental interest has been the subject of extensive research. Many techniques have been leveraged in order to attack the problem including dynamic run-time detection, static analysis and model checking.

Early work on dynamic data race detection includes the Eraser data race detector [15] which is based on computing locksets. There has been much work that improves upon the basic Eraser methodology. One such approach is [8] which leverages the use of static analysis to reduce the number of data race warnings that need to be validated via a run-time analysis. Other run-time detection tools based on Lamport's happened before model restrict the number of interleavings that need be explored [6, 11]. The advantage of run-time techniques is the absence of false warnings. On the other hand, the disadvantages are the extra cost incurred in instrumenting the code and poor coverage both of which become worse as the size of code increases especially in the context of concurrent programs. Additionally, run time detection techniques presume that the give code can be executed which may not be an option for applications like device drivers.

Model Checking[3], which is an efficient exploration of the state space of the given program, is another powerful technique that has been employed in the verification of concurrent programs [1, 4]. However, the state space explosion has made it hard to verify concurrent programs beyond 10K lines of code and is thus not, with the current state-of-the-art, an option for debugging large-scale real-life code.

Recently, there has been a spurt of activity in applying static analysis techniques for data race detection [5, 10, 17, 2, 13, 12, 7, 14, 8]. An advantage of such techniques is that they can be made to scale to large programs. The key disadvantage is that since static analysis works on heavily abstracted versions of the original program, they are not refined enough and can produce a large number of false warnings.

A credible approach is to strengthen static analysis to make it more refined with the goal of reducing the number of bogus warnings. The key steps to an accurate race detection procedure are (i) accurate shared variable discovery, (ii) scalable context sensitive pointer analysis to determine must locksets, and (iii) effective warning reduction. In this paper, we have proposed a new shared variable detection analysis that can be used to enhance existing correlation based techniques [7, 14]. Secondly, we have proposed a new scalable context sensitive must alias analysis which is critical in ensuring both scalability and accuracy of our race detection analysis. Prior context-sensitive alias analysis techniques have been shown to be more successful for Java [13, 12, 18] than C, whereas other techniques [7] simply do not use any pointer analysis at all which limits their accuracy. Finally, we have proposed a new two stage acquisition history based warning reduction technique which is provably more accurate than the existing lockset based analysis given in [8]. Experimental results on a suite of Linux drivers demonstrate the efficacy, viz., both the accuracy and scalability, of our new techniques.

## References

1. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *ASE*, 2000.
2. M. Burrows and K. Leino. Finding stale-value errors in concurrent programs. In *Compaq Systems Research Center SRC-TR-2002-004*, 2002.
3. E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, pages 52–71, 1981.
4. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE*, 2000.
5. D. Detlefs, K.R.M. Leino, G. Nelson, and J. Saxe. Extended static checking. In *TR SRC-159 Compaq SRC*, 1998.
6. A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.
7. D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.
8. Choi J, K. Lee, A. Loginov, R.O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.
9. V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *17th International Conference on Computer Aided Verification (CAV)*, 2005.
10. R. Leino, G. Nelson, and J. Saxe. Esc/java users’ manual. In *Technical Note 2000-002, Compaq Systems Research Center*, 2001.
11. J. Mellor-Crummey. One-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 Supercomputer Debugging Workshop*, 1991.
12. D. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.
13. D. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.
14. P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programming. In *ACM TCS*, volume 15(4), 1997.
16. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
17. N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.
18. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.