Data Race Detection Based on Dependence Analysis

Guanqun Wang
Hitachi Automotive Systems, Ltd.
guanqun.wang.ju@hitachi-automotive.co.jp

Masahiro Matsubara
Hitachi Automotive Systems, Ltd.
masahiro.matsubara.td@hitachi-automotive.co.jp

Abstract

Concurrency issues have been a serious problem in the whole software industry. They very often lead to hard-to-reproduce bugs and thus are very difficult to locate and solve. In this paper, we introduce a novel method for detection of one major category of concurrency problems, data race. This method extracts a Variable Dependence Tree from source code and finds data race by analysis on the Variable Dependence Tree. A prototype is built in Java according to the proposed method and tested on production-level source code (C language) with more than 400k LLOC. As a result, the prototype tool successfully detected all known data races including one relying on redundant structure of software and proved the effectiveness and feasibility of our proposed method.

1. Introduction

With the rapid growing complexity of software systems, concurrency issue has been an prevailing problem in software industry, which has drawn a lot of attentions. Since concurrency issues can easily lead to hard-to-reproduce bugs which take a long time to locate and fix, they are one of the most urgent problems to solve for cost control. According to a survey of Microsoft in 2007[1], over 60% of 684 software engineers had to deal with concurrency issues on a monthly basis. Concurrency bugs usually takes several days to detect and debug, and thus most of 684 engineers would welcome additional help on solving concurrency issues. Another survey also showed that about 73% of the examined non-deadlock concurrency bugs were not fixed by simply adding or changing locks, and many of the fixes were not correct at the first try[2], which implies the difficulties and costs software engineers face during solving non-deadlock concurrency problems.

In this paper, we will focus on data race, one main category of non-deadlock concurrency problem. Data race only happens in some certain program states, which makes its localization and fix very expensive in industrial practice because there are too many states in a real product's program to check one by one. To solve this problem, we propose a novel static method for data race detection based on dependence analysis. While a lot of existing static methods suffer from false positives, the proposed method theoretically has few false positives. By focusing on the dependence among data accesses instead of data accesses only, the proposed method can more effectively exclude program states (each corresponds to a combination of data accesses) that are irrelevant to data races so that the false positive rate can be kept in a manageable range. Regarding the target of detection, we choose source code, the relatively downstream output of software development. The main reason is that considering the fact that data races can be introduced into the program in both design and implementation, we want the proposed method to cover different phases of software development as much as possible.

In the rest part of this paper, Section 2 gives a clear definition of data race. Section 3 introduces some related work on data race detection. Section 4 describes the proposed method for data race detection. Section 5 shows a prototype tool we built based on the method introduced in Section 4. Section 6 describes the how the proposed method is evaluated with production-level source code. Section 7 discusses about the proposed method based on the evaluation. Section 8 gives a conclusion.

2. Data Race

Data race has been an important research topic since 1990s[3]. One widely used definition is:

definition 1. *When multiple threads*[1] *that concurrently run without any synchronization access the same location of memory, and at least one access is write, data race occurs.*

In this definition, we can see data race is a bug due to unsynchronized threads concurrently accessing the same memory. Such data access will lead to undefined behavior, and the actual ordering of accesses on the shared memory is unknown because it is not specified by source code. This unknown ordering may finally lead to unintended output. However, there is still a risk of bug when synchronized threads concurrently accessing the same memory. An improper synchronization can also lead to unexpected execution orderings of threads, which makes the final program output unintended. In fact, according to one characteristic study of concurrency bugs, around one third of the examined non-deadlock concurrency bugs are caused by violation to programmers' order intentions, which may not be easily expressed via synchronization primitives like locks and transactional memories[2]. Because both improper synchronization and unsynchronization can lead to an unintended thread execution ordering which finally leads to an unintended program output, we think it is reasonable to treat improper synchronizations the same as non-synchronization in data race detection. Therefore, "without any synchronization" is actually interpreted as "without any proper synchronization" in this paper.

One typical case of data race is shown in Figure 1. Thread A and thread B execute concurrently. One calculation in thread B depends on two readings of variable Var, which is updated in thread A. If the synchronization of thread A and B is improper or does not exist at all, it is possible for thread A to update Var right between two read accesses of thread B, which may finally lead to an incorrect program output.

There is another concept called race condition. Although race condition is used interchangeably with data race in some researches, it is more often considered as a more general concept. In this paper, we adopt the more general definition as follows,

[1]Thread, task and interrupt service routine (ISR) are used interchangeably to represent the general unit of concurrency. It is true that different operation systems or programming languages may have different names for the unit of concurrency, but we are only interested in the general concurrency issues in this paper.

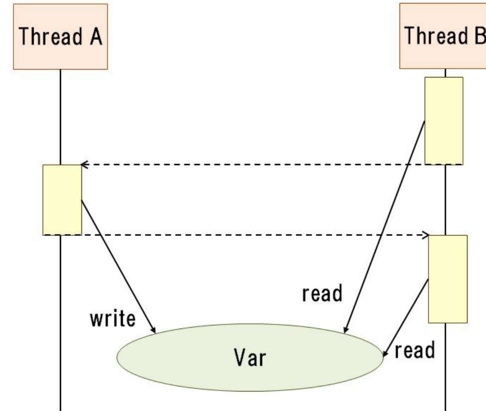

Figure 1. Data race on single variable

definition 2. *When the correctness of a program's output is affected by the potential nondeterminism in timing or ordering of threads' execution, race condition occurs.*

One interesting case is shown in Figure 2. Unlike the case in Figure 1, this race problem happens on two variables instead of one. More specifically, the two variables, $Var1$ and $Var2$ are actually a redundant variable pair, which means $Var1$ and $Var2$ stands for exactly the same thing, and they both exist due to the redundant structure of software which is applied in a lot of critical systems for safety concerns. Thread B reads $Var1$ and $Var2$, then compares their value to check if there is a malfunction in the system. Thread A runs concurrently with Thread B and updates both $Var1$ and $Var2$. If Thread A updates the values of $Var1$ and $Var2$ when Thread B already read $Var1$ but have not read $Var2$ yet, the comparison result of $Var1$ and $Var2$ may be wrong and finally lead to a system failure. This kind of race problem actually happen during development and is sometimes even more difficult to find and fix.

According to definition 1 and definition 2, the race-related problem shown in Figure 2 is actually a race condition instead of data race because Thread A and B are not accessing the same location of memory. However, this problem has almost the same mechanism as data race except there are two variables involved. Therefore, we think it is reasonable to improve definition 1 by changing "the same location of memory" to "memory storing the same information", which will make definition 1 generalize similar problems better. In the rest of this paper, we will use definition 3 instead of definition 1 for data race and consider the problem shown in Figure 2 as a data race.

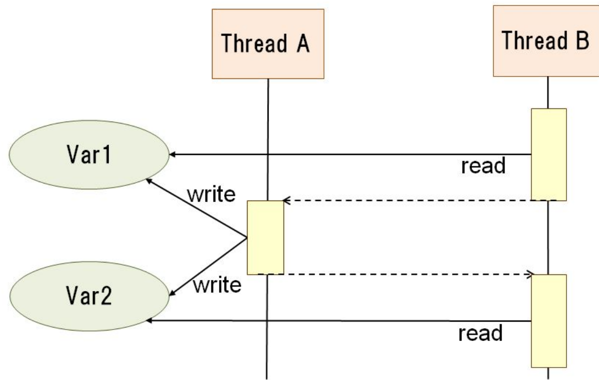

Figure 2. Data race on a redundant variable pair

definition 3. *When multiple threads that concurrently run without any synchronization access memory storing the same information, and at least one access is write, data race occurs.*

Without introducing definition 3, it is also possible to consider a race on a redundant variable pair as a combination of two data races, each of which includes exactly one write access and one read access of one variable, but we chose not to do so due to reduce false positives (details can be found in the 5th paragraph of Section 4.2).

3. Related Work

There are many researches done to solve the data race. The simplest method is Mutual Exclusion Locking discipline, which requires a lock be held on every access to a shared variable. Appropriate use of mutual exclusion locks can eliminate atomicity violations of any data, therefore prevent data races. Another simple method is to use buffer for shared variables, which allows a program to has a copy of certain variable at certain time and can use it at any time necessary. However, when we have a extremely large concurrent program, both methods lead to a very high maintenance cost. In addition, for programs of systems with limited resources (e.g. automotive control systems), it is not practical to use mutual exclusion locks or buffer for all variable shared between threads.

Type System can be used to prevent data race. The basic idea is to make a programming language that cannot realize data race. A lot of early researches on such type systems use locking discipline while some other ideas for the type system construction have also emerged. Recent work can be found in [4][5]. The main problem in type-based solution is the high cost of switching to a new language and the expression limitations in the current race-free languages.

Dynamic Analysis analyzes the runtime behavior of program to find potential data race. Most dynamic race detection tools do lockset-based analysis[6] or happens-before analysis[7] or both[8]. Current dynamic methods for data race detection usually suffer from two main problems. One is that such methods usually require instrumentation in the source code, which can dramatically increase the development cost sometimes. The other problem is that dynamic method can only find data race in the execution paths taken in the detection because the whole analysis is based on runtime behaviors. This makes dynamic detection unpractical for critical systems that run for a long time. For some programs with much longer runtime than the testing time (e.g. one automotive control program can run for thousands of hours in millions of cars), it is much more likely that a data race appearing in actual execution was missed in dynamic testing.

Static Analysis is used for race detection as well. Static method is usually very flexible, therefore the performance varies from one to the other. Among all static methods, some recent flow-sensitive methods showed potentials for industrial use because they have a combination of soundness and completeness. RacerX[9] is one good example. It intentionally sacrificed some soundness for better completeness (fewer false positives) by applying several heuristics in the detection. When elimination of false negative is not required, false positives can be controlled to a low level with this method. Inamori and Yamada proposed a static method for detection of data race caused by improper use of interrupts in automotive systems [10]. It searches for a data access pattern which ensures there is no false negative, and runs several checks to reduce false positives. In the target data access pattern, two read accesses with low priority and one write access with high priority happen on the same variable concurrently. Since this method uses model checking to eliminate false positives in the end, it suffers from the state explosion problem, which makes this method not feasible for large programs.

Model Checking is a method that can find all data race including those hard-to-reproduce ones. Unlike dynamic methods, model checking is able to explore all theoretically possible execution paths by analyze a model of the program instead of the program itself. However, model checking suffers from the state explosion problem. When the size of target program is very large, there are too many possible execution paths to check, which takes a lot of time and memory. In our previous work, we tried to apply a program slic-

ing before model checking to solve the state explosion problem[11]. For some complex programs, the state explosion still appears after slicing, so we also tried a divide and conquer strategy (divided the problem into small parts, do model checking for each and the interfaces between them). This strategy was proved effective, but another question comes up as how to determine which part to check first. When we know there is a bug, it will be possible to find it earlier if we can check the part that is most likely to contain bugs at first. However, this requires some techniques of bug prediction or vulnerability analysis.

4. Detection Method

In this section, we propose a novel method for data race detection based on dependence analysis, which statically analyze source code and point out data race in one part of the code that the user is most interested in. The proposed method is consisted of two main steps. The first step is extraction of variable dependence tree, and the second one is race search.

4.1. Extraction of Variable Dependence Tree

The first step of our proposed method, extraction of Variable Dependence Tree, is a technique originally used for program slicing, which was introduced in our previous work[11]. Program slicing is a popular technique used for locating bugs in large programs. This technique allows the user to focus on the code of interest, which can be the code that is most likely to have bugs, or code related with some observed malfunction. Specifically, it means to take a slice of the program containing all statements that may affect a specific variable at a specific point in the program. It was originally proposed by Weiser [12] and have been quite a hot topic in researches since then[13].

In our proposed method, instead of taking a slice of program related to one variable the user is interested in, we take a slice of variable dependence related to one specific variable the user is interested in and represent it in a tree called Variable Dependence Tree (VDT). Although in [11], VDT is ultimately used for program slicing, the actual program slicing is not necessary for data race detection in this paper because the detection is purely an analysis on VDT. In the rest part of this subsection, we will briefly introduce the concept of VDT and several other concepts it is built on.

The study on dependence representation of program is not a new research field. In 1987, Program Dependence Graph (PDG) was first introduced by Ferrante, Ottenstein and Warren to make explicit both the data and control dependences for each operation in a program. Specifically, PDG is a directed graph. In this graph, nodes are statements and predicates; edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends[14]. Based on PDG, Horwitz, Reps and Binkley proposed System Dependence Graph (SDG) for interprocedural slicing[15]. SDG extends PDG to incorporate sets of procedures rather than just monolithic programs and can be considered as a set of PDGs.

Variable Dependence Graph (VDG)[11] is a concept we developed based on concurrent version of SDG. VDG is a graph derived from SDG in which each node no longer represents a statement. A node in VDG represents a variable or a constant in a specific statement with a specific execution path or a specific control statement (e.g. if statement, while statement) with a specific execution path. For example, the same variable appearing in two different statements is represented by two nodes (one for each statement). The same variable in the same statement that has two possible execution paths is also represented by two nodes (one for each execution path). The label of each node is the name of the variable or the control statement this node represents. Besides the name shown in label, each node also carries trace information of the variable or control statement it represents. An edge in VDG represents data dependence or control dependence between two nodes. The method to extract a VDG from a program is discussed in [11].

Variable Dependence Tree (VDT) is a VDG pruned into a tree. There are two types of VDT, Goal VDT and Start VDT. We will only use Goal VDT in this paper. A goal VDT is a directed tree $T(V, E)$ rooted at node s, where V is the set of nodes, and E is the set of edges. The set of children of node v is denoted by $C(v)$. $\forall v \in V$ depends on $\forall c \in C(v)$. The edge between node v and its child $c \in C(v)$ points from c to v.

Since the transformation from VDG to VDT is already introduced in [11], we will only briefly describe the basic idea about it in this paper. To prune a VDG into a VDT, we search for nodes depending on the same node in VDG. The set of nodes depending on an arbitrary node x is denoted by D. For all edges pointing from x to $d \in D$ except the leftmost one, remove the edge, and add a new leaf node x'_i as the child of $d \in D$. The new leaf node x'_i means that the dependence from x has already appeared elsewhere in this tree and therefore can be ignored, where the subscript i is just an index to prevent redundant node labels. A simple example is shown in Figure 3 and Figure 4. Fig-

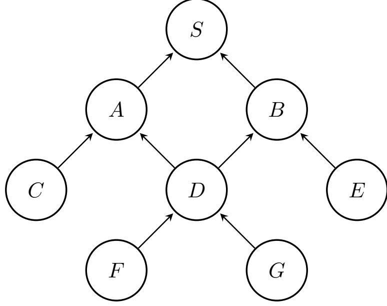

Figure 3. VDG extracted from variable S

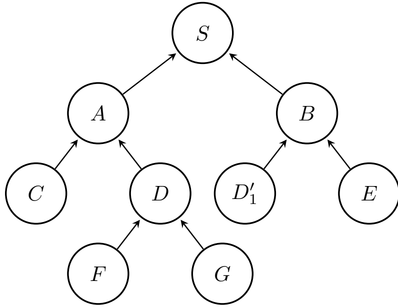

Figure 4. VDT transformed from VDG in Figure 3

ure 3 shows a VDG extracted from variable S, where all nodes represents variables. As we can see, node A and node B both depends on node D. To prune this VDG into VDT, we keep the leftmost edge (A, D) and remove edge (B, D). Then we add a new leaf node D'_1 as a child of variable B. In this way, the VDG in Figure 3 is pruned into the VDT in Figure 4. When the resources (e.g. time, memory) is limited, a partial extraction of VDT is also possible, which is introduced in [11].

By adopting VDT, we are able to eliminate all artificial orderings in the program and make potential parallelism explicit. Since variables are represented as nodes, it is more efficient to use VDT to detect data race happening on specific variable(s). In the real-life application, it is usually efficient and effective to extract a VDT from a certain error flag or output variable the user is interested in. If the user does not have information about such a variable and just wants to reach a full coverage of the dependence in the program, which is not recommended because of the efficiency concerns, the user can extract a VDT from each final output of the program.

4.2. Race Search

In the algorithm of race search, a concept called interference dependence is used. The early definition and

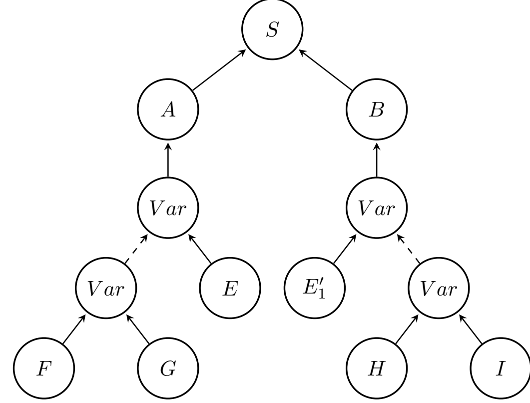

Figure 5. The pattern of data race candidate

application of interference dependence can be found in Krinke's work[16]. According to Krinke, interference dependence occurs when a variable is defined in one thread and used in a parallel executing thread. Specifically, in a VDT, if an edge connects two nodes with different thread information, this edge represents an interference dependence. In [17], Krinke also showed how to analyze interference dependence efficiently.

In our proposed method, we find data race candidates based on dependence analysis. Specifically, if multiple threads affects the memory storing the same information, this situation is considered as a data race candidate. Then data races are extracted by filtering these candidates through priority and reality checks. After adopting the concept of VDT and interference dependence, the situation considered as data race candidate can be interpreted to "In a VDT, ① there are multiple interference dependences, ② variable(s) storing the same information appear on the descendant side of each interference dependences." Figure 5 shows an example of such situation in VDT. There are two interference dependences (drawn as dotted arrows), which means the value of S is affected by multiple threads. Variable Var appears on the descendant side (bottom side in the tree) of each interference dependence, which means that value of S depends on two readings of Var while these two readings are obtained in a thread different from where S is defined. When the thread where S is defined and the thread(s) where two readings of Var are done execute concurrently, the value of S may be different for a different execution ordering, thus an unintended value becomes possible, which is exactly the data access problem we showed in Figure 1. The details on priority and reality checks will be introduced later in this section.

It is important to notice that the variables storing the same information on the descendant side of interfer-

```
1  int x1,x2,a,b,c;
2  int DI0;
3  int errflg;
4  void main(void){
5      x1=a+1;
6      x2=b+1;
7      if (x1!=x2){
8          errflg=1;
9      }
10 }
11 void calc(void){
12     c=DI0;
13     a=c*2;
14     b=c+c;
15 }
```

Figure 6. Sample code for data race due to indirect influences of data access

ence dependence (Var in the case of Figure 5) are not necessarily the same variable, they can also be a redundant variable pair (two variables storing the same information because of redundant structure), which corresponds to the case in Figure 2. Of course, in order to recognize redundant variable pairs, the user has to collect additional information about redundant variables before data race detection. Fortunately, such information can usually be extracted from design documents.

One interesting fact is that even when the user cannot recognize redundant variable pairs, the data race on redundant variable pairs can still be partially detected. For the case shown in Figure 2, it is possible that $Var1$ and $Var2$ finally depend on the same variable in VDT. For instance, this variable can be some raw sensor data, and $Var1$ and $Var2$ are calculated based on it by different calculation methods. We did not confirm if such case actually exist in evaluation, but it is theoretically possible. Figure 6 shows a sample code of this case. Main() and calc() are two functions running concurrently. a and b are redundant variables used to calculate $x1$ and $x2$ respectively in main(). If $x1$ and $x2$ are not equal, an error flag is triggered. In calc(), a and b are calculated based on variable c (raw sensor data) with different methods. Indeed, data race happens on the redundant variable pair a and b. However, the dependence on the same variable (variable c in the sample code) makes such data race no different from those on single variable. Thus it can be successfully detected even when we do not recognize a and b as a redundant variable pair. Nevertheless, the information on redundant variable pairs is still helpful because it makes the data race detectable at a higher level in VDT.

When locating data race candidates as introduced

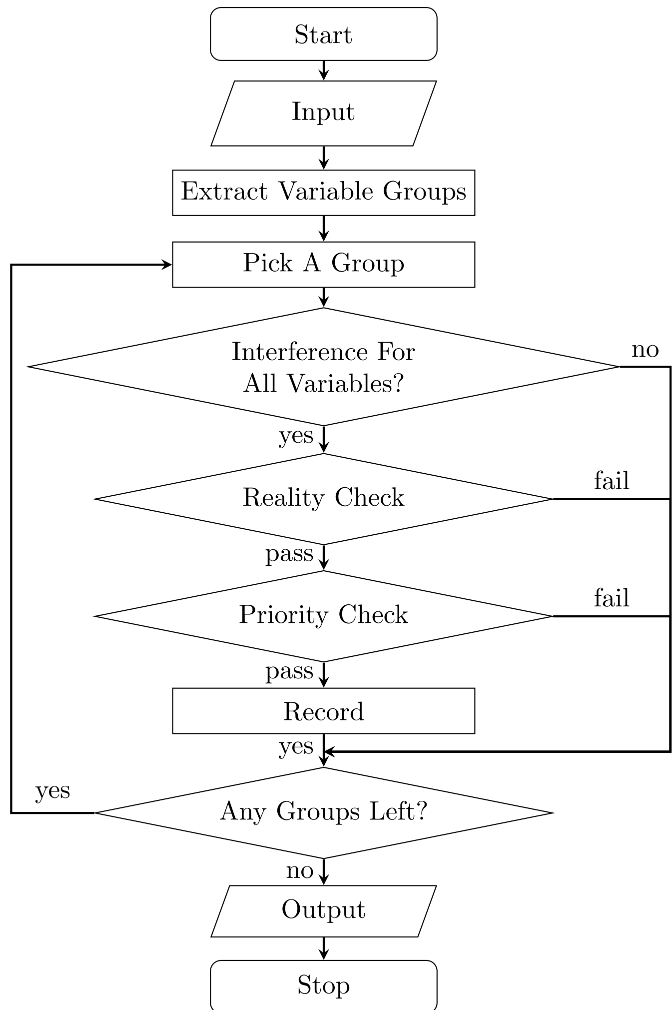

Figure 7. Flowchart of race risk search

above, a kind of data race is intentionally excluded in detection. No matter in the typical definition of data race (definition 1) or the extended definition (definition 3) used in this paper, data race happens when only two concurrent accesses of the same variable exist, one of which writes and the other reads. In fact, data race on a redundant variable pair can also be considered as a combination of two such "1 read & 1 write" data races. However, all "1 read & 1 write" data races are excluded in detection for a lower false positive rate. In a large concurrent program such as an engine control program, it's normal to see many such "1 read & 1 write" data races, but they rarely cause real problems in practice since they are easy to find and fix by conventional methods.

The algorithm of potential race search is shown in the flow chart in Figure 7. The input block takes a VDT, information on thread execution priority as well

as redundant variable, and reality information as input. Reality information refers to the information about if some threads can run concurrently in reality. This information can include synchronization information, hardware limitations, limitations of the runtime environment and so on. Then "Extract Variable Groups" process traverses the VDT to extract nodes representing access to the variable that store the same information and divides them into groups by variable (e.g. Var in Figure 5). After extracting variable groups, "Pick A Group" process randomly picks one group for further analysis. Decision block "Interference For All Variables?" checks the paths from picked nodes to a closest common ancestor. If there are interference dependences in multiple paths, a data race candidate is found and the algorithm goes to decision block "Reality Check". If not, the algorithm jumps to "Any Groups Left?" block.

"Reality Check" and "Priority Check" block reduce false positives by checking if the threads involved in data race candidates can actually run concurrently without synchronizations during runtime. "Reality Check" checks concurrent execution limitations that are not determined in the program under test (e.g. external synchronizations, limitations of hardware platform or runtime environment). In a program with optimized resource consumption and complexity, it is normal that only some threads can run in a specific scenario and some threads can never do even if it is not explicitly set in the program. For example, a thread for situation when car speed is equal to 0 and a thread for situation when car speed is higher than 100mph can never run concurrently. "Priority Check" checks the execution priority of threads involved in the data race candidates. Given the OS or hardware platform where the program runs, some threads are not allowed to run concurrently due to execution priorities. For example, in a lot microcontrollers, threads with the same priority are executed sequentially instead concurrently, therefore there can never be data race between them. It should be noticed that rules in both checks are flexible and should be adjusted based on the program under test or the goal of testing.

When both checks pass, the algorithm will go to "Record" block. Otherwise, the algorithm jumps to "Any Groups Left?" block. "Record" block records candidates that passed the checks as potential data races. Decision block "Any Groups Left" checks if there is any extracted variable group left. if there is any, the algorithm goes back to "Pick A Group" block to pick up a new group and search for potential data race. If not, the algorithm ends.

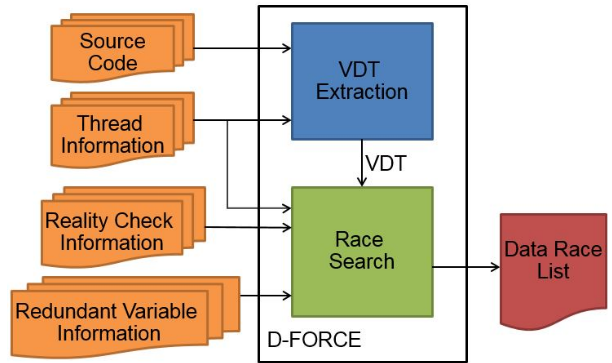

Figure 8. Overview of prototype tool

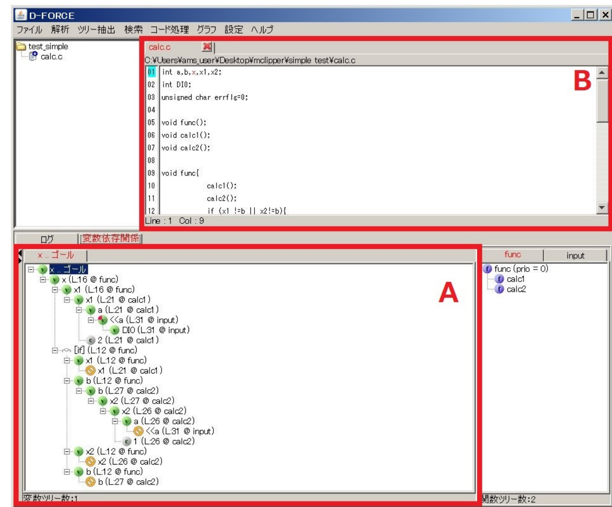

Figure 9. Screen capture of D-FORCE

5. Implementation

We built a prototype tool called D-FORCE in Java for the proposed detection method introduced in Section 4. As shown in Figure 8, it takes inputs including source code, redundant variable information, thread information (priority and entry point of each thread) and reality check information (which threads may run concurrently) to generate data race list. For an efficient implementation, we assumed the target programs are only written in C language without loss of generality. One screen capture of the D-FORCE is shown in Figure 9. Since it is also used in our other researches, only area A and B are used for data race detection. Area A shows the extracted VDT, and area B is a source code reference window, which shows the corresponding source code of selected node in VDT so that the user gets a visual confirmation of actual source code.

Figure 10 shows a sample source code. If we extract

a VDT from variable x, it will be visualized in area A as shown in Figure 11. This is a standard result of the first step of our proposed method, extraction of VDT. Each node has an icon followed by a label. The icon shows data type. If the node represents a variable, a green round icon with a “v” inside is shown, just like the one before label “x (L:16@func)”. If the node represents a constant, a gray icon with a “c” inside is shown, just like the one before label “2 (L:21@calc1)”. If the node represents a control statement, a diamond shape icon is shown, just like the one before label “[if] (L:12@func)”. If a node represents anything that already appeared somewhere else in the VDT (such as the D'_1 in Figure 4), the icon is a yellow stop sign, just like the one before “x1 (L21@calc1)”. Label gives more details about the node. For example, label “x (L:16@func)” means the node represents variable x on line 16 of function $func$. A label “[if] (L:12@func)” means the node represents the if statement on line 12 of function $func$. When a variable node is at the write side of an interference dependence, the variable icon is marked with a red arrow and a “≪” mark is added to the beginning of the label.

The result of potential race risk search on the VDT is also shown in Figure 4. At the root node, the “(risk=1)” shows that there is one potential data race detected. A “=ic1=” mark is added at the beginning of the label “x (L:16@func)”, which indicates that this variable is the common ancestor related with the first detected data race. “=rd1=” and “=wo1=” indicates the corresponding variable are one side of the interference dependence related with the first detected data race. When such a result is available, D-FORCE also automatically generates a file with all formation about locations of interference dependences and the common ancestor, based on which the user can understand how the potential data race can happen and trace back to the exact problematic statements in the source code.

6. Evaluation

The feasibility of the proposed method in industrial use is confirmed in our evaluation. We ran D-FORCE on obsolete version code of two different powertrain ECUs (Electrical Control Unit) developed in Hitachi Automotive Systems. 100% (4 out of 4) data races actually encountered during development were successfully detected while the number of false positives was maintained in a manageable range.

Both sets of source code are written in C and have more than 400k LLOC (Logical Lines of Code). D-FORCE was executed in Windows 7 running on a PC with a Intel Core i7 870 CPU (2.93GHz) and 16GB

```
1  int a,b,c,x,x1,x2;
2  int DI0;
3  unsigned char errflg=0;
4
5  void calc1(void);
6  void calc2(void);
7  void input(void);
8
9  void func(){
10         calc1();
11         calc2();
12         if (x1 !=c || x2!=c){
13                 errflg=1;
14         }
15         else {
16                 x=x1;
17         }
18 }
19
20 void calc1(){
21         x1=a*2;
22         b=x1;
23 }
24
25 void calc2(){
26         x2=a<<1;
27         b=x2;
28 }
29
30 void input(){
31         a=DI0;
32 }
```

Figure 10. Sample source code

```
x .. ゴール(risk = 1)
  =ic1=x (L:16 @ func)
    x1 (L:16 @ func)
      x1 (L:21 @ calc1)
        =rd1=a (L:21 @ calc1)
          =wo1=<<a (L:31 @ input)
            DI0 (L:31 @ input)
        2 (L:21 @ calc1)
    [if] (L:12 @ func)
      x1 (L:12 @ func)
        x1 (L:21 @ calc1)
      b (L:12 @ func)
        b (L:27 @ calc2)
          x2 (L:27 @ calc2)
            x2 (L:26 @ calc2)
              =rd1=a (L:26 @ calc2)
                =wo1=<<a (L:31 @ input)
              1 (L:26 @ calc2)
      x2 (L:12 @ func)
        x2 (L:26 @ calc2)
      b (L:12 @ func)
        b (L:27 @ calc2)
```

Figure 11. One example of VDT visualization in D-FORCE

RAM. The whole analysis finished on both sets of source code within 15 minutes. Redundant variable information and thread information were obtained manually by reviewing design documents. Since we noticed that some false positives appeared in initialization threads, we manually excluded all initialization threads during data race detection, which can essentially be considered as a kind of reality check.

Among 4 known data races, 3 of them have the same data access pattern, which is exactly the typical case shown in Figure 1. The other one is a data race on a redundant variable pair as shown in Figure 2. All 4 data races were confirmed as bugs leading to system failure in previous development. According to our observation, false positives still exist, so a more accurate measurement on completeness is necessary in the future work, after which we can decide the direction for further improvements of the proposed method.

7. Discussion

In the proposed method, we did not use synchronization information to find data race candidates because we wanted to detect data race due to both "no synchronization" and "wrong synchronization". Obviously, synchronization is partially used in the priority check to reduce false positives, but there are still some synchronization informations that can be helpful but are currently not used. For instance, information on interrupt disables are used in the work of Inamori and Yamada[10] to reduce false positives while it is not used in our method at all. Such underutilization of synchronization information has pointed out one possible direction for us to further reduce false positives.

Dependence analysis in the proposed method allows data race detection to focus on interference in dependence instead of accesses on certain variables. This enables detection of data race resulted from indirect influences of data access. The code in Figure 6 is also an example for this feature. When variable c is updated in one thread, a concurrently running thread does not necessarily have to read exactly variable c multiple times to cause a data race. In the thread where c is updated, variable a and b are defined based on c. A concurrently running thread can also cause a data race by reading a and b and using their values together for some other calculation (a and b might be calculated from different c values). In this case, there are at most two accesses on variables shared among threads (a and b), so it is very hard to detect by conventional methods without generating many positives. In our proposed method, this is not a problem due to dependence analysis.

```
1 int a;
2 void calc(int *pt);
3 void main(void);
4     a=0;
5     calc(&a);
6 }
7 void calc(int *pt){
8     *pt=3   ;
9 }
```

Figure 12. Sample code for data access through pointer

While a lot of data race detection methods have trouble analyzing program with pointers, dependence analysis in our proposed method helps to find out which piece of memory is accessed when pointers are involved in the data access. One example is the sample code in Figure 12. Main() and calc() are two function that run concurrently. They actually access the same variable a, but the write access in calc() is done through a pointer passed to it. Since pointers are also included the dependence analysis, it is very easy to figure out the two threads are accessing the same variable a in our proposed method.

It is also important to notice that our proposed method is naturally compatible with programs containing parallel threads even if we only mentioned concurrent threads so far. For the proposed method, the only difference brought by parallel threads is that the data accesses can not only be concurrent, but also simultaneous (physically at the same time). This difference cannot affect how we identify data race candidates or the correctness of checks applied to filter them, so the proposed method can also detect data races in program with parallel threads as long as we have information about which threads run in parallel.

When a data race happens, the program's output may still be correct after certain processing in the program. Such situations are relatively rare, but still possible. The data race that does not affect the correctness of a program's output is referred to as "benign data race"[18]. Our proposed method cannot completely distinguish benign data race, which seems like a problem. However, recent researches showed that it is impossible to guarantee that a data race never affects the correctness of a program as long as the compiler or hardware can be changed[19]. Therefore, we chose to include "benign data race" in the detection results.

8. Conclusion

In this paper, we proposed a novel method for data race detection based on dependence analysis. A pro-

totype tool is build in Java to realize the proposed method. By running the prototype tool on production-level code of two real industrial products, the proposed method proved to be feasible for data race detection in industrial practice. However, more empirical validation is still necessary in the future. Besides the programs we used for evaluation in this paper, the proposed method should also be tested on more programs for a more accurate measurement of soundness and completeness, after which we may be able to make necessary improvements (e.g. further reducing false positives by a better utilization of synchronization information such as interrupt disable and mutex).

References

[1] P. Godefroid and N. Nagappan, "Concurrency at microsoft: An exploratory survey," in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM Sigplan Notices*, vol. 43-3, pp. 329–339, ACM, 2008.

[3] R. H. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[4] E. Castegren and T. Wrigstad, "Reference capabilities for concurrency control," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 56, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[5] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and reference immutability for safe parallelism," in *ACM SIGPLAN Notices*, vol. 47, pp. 21–40, ACM, 2012.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[7] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *ACM SIGPLAN Notices*, vol. 26, pp. 85–96, ACM, 1991.

[8] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *ACM SIGPLAN Notices*, vol. 38, pp. 167–178, ACM, 2003.

[9] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 237–252, ACM, 2003.

[10] Y. Inamori and N. Yamada, "Static interrupt-race detection method for c program in vehicle," *IPSJ Transactions on Consumer Devices&Systems (CDS)*, vol. 3, no. 3, pp. 76–84, 2013.

[11] M. Matsubara, K. Sakurai, F. Narisawa, M. Enshoiwa, Y. Yamane, and H. Yamanaka, "Model checking with program slicing based on variable dependence graph," in *First International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2012)*, p. 88, 2012.

[12] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, 1981.

[13] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[15] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[16] J. Krinke, "Static slicing of threaded programs," *ACM Sigplan Notices*, vol. 33, no. 7, pp. 35–42, 1998.

[17] J. Krinke, "Context-sensitive slicing of concurrent programs," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 178–187, 2003.

[18] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *ACM SIGPLAN Notices*, vol. 42, pp. 22–31, ACM, 2007.

[19] H.-J. Boehm, "How to miscompile programs with" benign" data races.," in *HotPar*, 2011.