Dynamic Slicing of Multithreaded Programs for Race Detection

Sriraman Tallam
Google Inc.
tmsriram@google.com

Chen Tian, Rajiv Gupta
University of California at Riverside
{*tianc, gupta*}*@cs.ucr.edu*

Abstract

Prior work has shown that computing dynamic slices of erroneous program values can greatly assist in locating the root cause of erroneous behavior by identifying faulty statements in sequential programs. These dynamic slices represent backward transitive closure over exercised read-after-write data dependences and control dependences. However, for a multithreaded program executing on a processor, data races represent an additional source of errors which are not captured by dynamic slices. We present an extended form of dynamic slice for multithreaded programs which can assist in locating faults, including those caused by data races. We demonstrate the effectiveness of our approach via case studies and also describe an efficient algorithm for computing dynamic slices.

1. Introduction

Given a program run, the dynamic dependence graph (DDG) captures the dynamically exercised Read-After-Write (RAW) and Control dependences – each node in the graph represents an execution instance of a statement while the edges represent the dependences. Let $s < t >$ denote an execution instance t of statement s. The dynamic slice of the value computed by $s < t >$ during a program run is the subgraph of DDG that is reachable via a backward traversal of DDG starting at the node corresponding to $s < t >$. This traditional definition of dynamic slicing has been found to be useful in locating bugs in single-threaded applications.

In this paper we consider fault location in multithreaded programs executing on a uniprocessor. In prior work we have shown that a straightforward extension of dynamic slicing for multithreaded programs can detect certain types of faults [28, 24]. However, presence of *data races* is not captured by the above dynamic slices. Hence, backward dynamic slices by themselves are inadequate for debugging programs when one source of errors can be the presence of data races. One approach could be to use a specialized data race detector for detecting data races and use dynamic slicing for other types of faults. Specialized race detectors are based upon *happens-before* algorithms [6, 21, 15], *lock-set* algorithms [23, 12, 9, 29], or a combination of the two [7, 19, 27]. If a fault is caused by a data race error, these race detectors can be effective despite their false alarms or extra hardware requirements. However, when the programmer observes anomalous behavior, he or she does not know what kind of fault is present and would thus have to make use of multiple tools to debug the program. A more desirable approach that we present in this paper develops an extended dynamic slicing algorithm that can be used to debug data races as well as other types of faults. The key contributions of this paper are as follows.

- *Extended Dynamic Slicing*. First, in section 2, we develop an extended dynamic slicing algorithm that, in addition to considering Read-After-Write (RAW) and Control dependences, also includes selected Write-After-Write (WAW) and Write-After-Read (WAR) dependences in the dynamic slice to capture data races in the dynamic slice. We demonstrate the effectiveness of this approach through case studies based upon real bugs.
- *Efficient Dynamic Slicing*. Second, in section 3, we present an efficient approach for constructing the dynamic dependence graph needed for computing the dynamic slices. A series of optimizations are developed which ensure that capturing only a small subset of Write-After-Write (WAW) and Write-After-Read (WAR) dependences is sufficient for computing the extended dynamic slices.

It is worth noting that there have been prior approaches to compute the dynamic slices for concurrent programs and distributed programs [26]. One of the early works [5] presents a graph based approach to computing the slice. In addition to data and control dependences, three new forms of dependences are defined and used: *selection, synchronization and communication dependences* that can occur in concurrent and distributed programs. Duesterwald et al. [8] proposed another algorithm to compute *executable* slices of distributed programs. In both of the above works the dependences consider are still inter- and intra-process control and

RAW data dependences. Thus, they are inadequate for data race detection as they do not consider WAW and WAR dependences. Moreover, the slices computed by these methods are *imprecise*. In our work we use highly optimized algorithms that compute precise slices with efficiency.

Although other works [11, 10, 20] presented the algorithms that can compute dynamic slices precisely, they are not general enough for detecting errors in shared memory concurrent programs. Specifically, the method presented in [11] can only be used in concurrent Ada programs with rendezvous communications. And the work proposed in [10, 20] are only applicable for distributed programs. In contrast, the dynamic slicing algorithm proposed in this paper is more general and applicable to multithreaded programs where the different threads communicate through shared memory.

2. Multithreaded Dynamic Slicing

Next we explain why the presence of a data race is not captured by a backward dynamic slice and how we can extend the notion of dynamic slicing to address this problem. A data race arises when a variable is being concurrently accessed by two threads with at least one of the access being an update. The order in which these accesses occur may vary depending upon the order in which the threads are scheduled. When the value of variable is read at its use, the value obtained may be the updated one or the one prior to the update. The DDG will simply contain a RAW edge that indicates the source of the value. However, to understand the presence of a bug we need additional information, i.e. the DDG should not only identify the thread from which value was obtained but also the thread from which value could have been obtained under a different schedule. Next we show how by incorporating additional dependence edges, Write-After-Write (WAW) and Write-After-Read (WAR), we can achieve this goal.

Consider a situation in which thread T_1 writes to variable X at statement D_1 and later reads the value of X at statement U_1. In addition, thread T_2 concurrently writes to variable X at statement D_2. However, due to presence of a data race the value read at U_1 may come from D_1 or D_2. Let us first consider the execution timing illustrated by Fig. 1(a) where, when T_1 reads the value of X at U_1, it receives the value from D_2 which executes after D_1 has executed. If we compute the dynamic slice of U_1, it includes D_2 but not D_1 and thus the presence of the data race between D_1 and D_2 is not captured by the dynamic slice. However, if we extend the DDG to also capture the **WAW** dependence from D_1 to D_2 and extend the dynamic slice to include this backward WAW dependence, then we will be able to capture both D_1 and D_2 to reveal the presence of a data race whose alternate outcome would have caused U_1 to read the value of X defined by D_1 instead of the value of X defined at D_2.

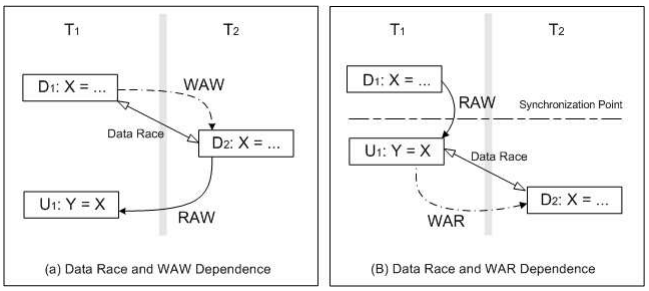

Figure 1. Data race and WAW, WAR dependences.

Fig. 1(b) shows another execution timing where U_1 is executed by T_1 before D_2 is executed by T_2. The backward dynamic slice in this situation includes D_1 but not D_2. However, if we extend the DDG to capture **WAR** dependence from U_1 to D_2 and extend the dynamic slice to include this forward WAR dependence, then we will be able to capture both D_1 and D_2 to reveal the presence of a data race whose alternate outcome would have caused U_1 to read the value of X defined by D_2 instead of the value of X defined at D_1.

In summary, we must extend the DDG to include inter-thread **WAW** and **WAR** data dependences and then construct the dynamic slice to include certain forward WAR and backward WAW dependences to capture data races. In the remainder of this section we present the precise form of DDG and algorithm for computing dynamic slices. We also then apply our approach to a few real bugs and demonstrate that our approach is highly effective.

2.1. Extended Dynamic Slicing

Let S represent the set of executed program statements for an multithreaded program execution. Let $s\langle t, T\rangle$ denote the unique execution instance of a statement s ($s \in S$) at time t by thread T. Note that since the multithreaded programs are being run on a uniprocessor, there is a strict time order of the various instructions executed by the different threads and hence each instruction can be uniquely timestamped.

Further, $s\langle t, T\rangle$ is said to be dependent on $s'\langle t', T'\rangle$, denoted by $s'\langle t', T'\rangle \rightarrow s\langle t, T\rangle$ (also called *dependence edge*), if there is a $dependence$ between the execution instance of statement s at timestamp t by thread T and the execution instance of statement s' at timestamp t' by thread T'. The $dependence$ can be one of four types: control, Read-After-Write, Write-After-Write, or Write-After-Read. For convenience, we use the abbreviations *CTRL-Dep*, *RAW-Dep*, *WAW-Dep*, and *WAR-Dep* to respectively denote sets of these four kinds of exercised dependences. Then, a DDG for a multithreaded program execution is defined as follows.

The DDG of a program's execution is a directed graph (N, E) where N is the set of nodes in the graph and E is the set of edges where,

$$N = \{s\langle t, T\rangle | s(s \in S) \text{ executed at time } t \text{ by thread } T\ \}$$
$$E = \{e | e = (s\langle t_1, T_1\rangle \rightarrow s'\langle t_2, T_2\rangle)$$
$$\in \{\textit{CTRL-Dep} \cup \textit{RAW-Dep} \cup \textit{WAW-Dep} \cup \textit{WAR-Dep}\}\} \ .$$

From this definition, we can see that each dependence edge falls into one of the four different types of dependences. However, note that we only consider WAW and WAR edges that are inter-thread edges; hence, $T_1 \neq T_2$ if a dependence edge e is in WAW or WAR set.

Function *Slice_Orig*($DDG(N, E)$, $s\langle t, T\rangle$)
1: $NS(s\langle t, T\rangle) = \{s\}$;
2: $ES(s\langle t, T\rangle) = \{\}$;
3: **for all** s' such that there exists an edge $s'\langle t', T'\rangle \rightarrow s\langle t, T\rangle \in$ *CTRL-Dep* $\cup$ *RAW-Dep* **do**
4: $(NS(s'\langle t', T'\rangle), ES(s'\langle t', T'\rangle))$ = *Slice_Orig*($DDG(N, E)$, $s'\langle t', T'\rangle$);
5: $NS(s\langle t, T\rangle) = NS(s\langle t, T\rangle) \cup NS(s'\langle t', T'\rangle)$;
6: $e = s'\langle t', T'\rangle \rightarrow s\langle t, T\rangle$;
7: $ES(s\langle t, T\rangle) = ES(s\langle t, T\rangle) \cup \{e\} \cup ES(s'\langle t', T'\rangle)$;
8: **end for**
9: **return** $(NS(s\langle t, T\rangle), ES(s\langle t, T\rangle))$;

Function *Slice_Race*($DDG(N, E)$, $s\langle t, T\rangle$)
1: $NS(s\langle t, T\rangle) = \{\}$;
2: $ES(s\langle t, T\rangle) = \{\}$;
3: **for all** v in the node set of $SLICE_{orig}(s\langle t, T\rangle)$ that has RAW edges **do**
4: **for all** u such that there exists an edge $u\langle t_u, T_u\rangle \rightarrow v\langle t_v, T_v\rangle \in$ *WAW-Dep* **do**
5: $NS(s\langle t, T\rangle) = NS(s\langle t, T\rangle) \cup \{u\}$;
6: $e_{race} = u\langle t_u, T_u\rangle \rightarrow v\langle t_v, T_v\rangle$;
7: $ES(s\langle t, T\rangle) = ES(s\langle t, T\rangle) \cup \{e_{race}\}$;
8: **end for**
9: **for all** w such that there exists an edge $v\langle t_v, T_v\rangle \rightarrow w\langle t_w, T_w\rangle \in$ *WAR-Dep* **do**
10: $NS(s\langle t, T\rangle) = NS(s\langle t, T\rangle) \cup \{w\}$;
11: $e_{race} = v\langle t_v, T_v\rangle \rightarrow w\langle t_w, T_w\rangle$;
12: $ES(s\langle t, T\rangle) = ES(s\langle t, T\rangle) \cup \{e_{race}\}$;
13: **end for**
14: **end for**
15: **return** $(NS(s\langle t, T\rangle), ES(s\langle t, T\rangle))$;

$SLICE_{orig} = Slice\_Orig(DDG(N, E), s\langle t, T\rangle)$
$SLICE_{race} = Slice\_Race(DDG(N, E), s\langle t, T\rangle)$
$SLICE(s\langle t, T\rangle) = SLICE_{orig} \cup SLICE_{race}$

Figure 2. Dynamic slice computation.

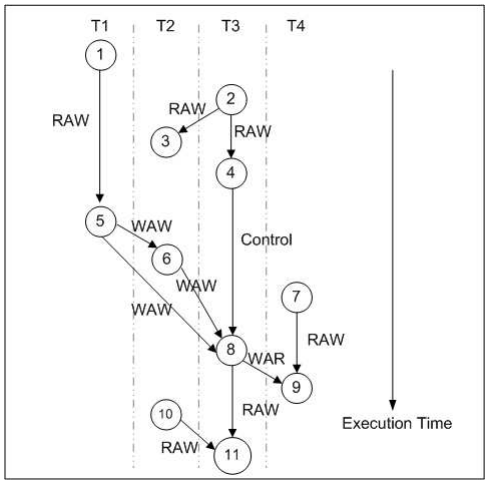

Figure 3. A DDG example.

Given DDG (N, E), the backward dynamic slice of a statement s executed at time t by thread T, denoted as $SLICE(s\langle t, T\rangle)$, is a subgraph of DDG, $\{NS(s\langle t, T\rangle), ES(s\langle t, T\rangle)\}$. It is a union of two parts. The first part is denoted by $SLICE_{ORIG}(s\langle t, T\rangle)$. Here, the nodes represent statements that are directly or indirectly linked to the statement instance $s\langle t, T\rangle$ through RAW or control dependences edges. The edges represent the traversed dependences. Statement s itself is also one of the nodes. $SLICE_{ORIG}(s\langle t, T\rangle)$ can be computed by the recursive function $Slice\_Orig()$ shown in Figure 2. The second part is denoted by $SLICE_{race}(s\langle t, T\rangle)$. Consider every node n in $SLICE_{ORIG}(s\langle t, T\rangle)$ that is involved with one or more RAW dependences edges. Then the nodes in $SLICE_{race}(s\langle t, T\rangle)$ represent those statements directly linked to n via a WAW or WAR dependence edge. The edges in $SLICE_{race}(s\langle t, T\rangle)$ represent these WAW and WAR dependences. Function $Slice\_Race()$ in Figure 2 shows how to compute $SLICE_{race}(s\langle t, T\rangle)$. In this function, line 4-8 search for the WAW dependences and line 9-13 for WAR dependences in the DDG. Note that for every node v, there can be multiple WAW and WAR dependences.

Figure 3 shows a simple DDG example where 11 statement instances are executed by 4 threads. Now let us assume that node 11 is faulty and we need to compute its slice. As discussed above, we first compute $SLICE_{orig}(11)$. Following the algorithm, we can finally get its node set {2,4,8,10,11} and the edge set which contains the edges among these nodes. $SLICE_{race}(11)$ can also be easily computed by traversing the nodes in $SLICE_{orig}(11)$. this produces the node set {5,6,9} and the edge set $\{5 \rightarrow 8, 6 \rightarrow 8, 8 \rightarrow 9\}$. The final slice of node 11 can be obtained by taking the union of $SLICE_{orig}(11)$ and $SLICE_{race}(11)$. Since the WAW and WAR dependences are incorporated,

it can be used to understand data race errors in addition to other types of errors that can be found using traditional dynamic slices.

From this example, it is clear that the closure is taken over all nodes that had RAW or control edges, but not over nodes that had WAW and WAR edges. This is because the value at the fault point could have been affected by only statements along RAW and control dependent chains. WAW and WAR edges point at places where races could have occurred but the value at the point where the fault is observed could not have been computed directly or indirectly by statement execution instances along WAW and WAR chains.

2.2. Case Studies

We now use some examples to illustrate how a data race error is found using multithreaded slicing.

Mysql-I. `mysql` [4] is a multithreaded application which is one of the world's most popular open source databases. It is known for its consistent fast performance, ease of use, and high reliability. It is used in more than 10 million installations and runs on more than 20 platforms.

The program `mysql` ver. 4.0.12 has an atomicity violation bug [1] which is as follows. A thread that tries to close and open a new log file atomically in order to flush the previous log gets interrupted just after closing the old log by another thread that does an insert operation into a database. The second thread, hence, does not find any open log files and does not record the insert operation. These logs are used to restore databases and incorrect logs can result in inconsistency.

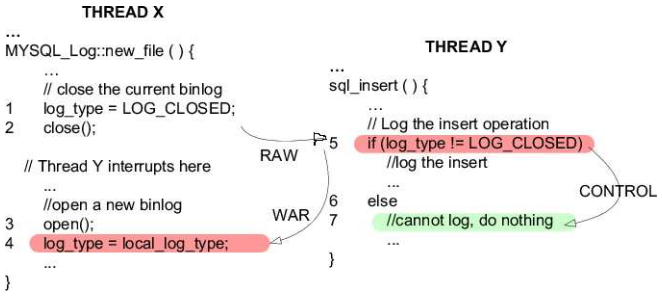

Figure 4. Data race error in `mysql` - I.

Figure 4 shows the code executed by the two threads that lead to the fault. Thread X closes the *binlog* (log that stores all database operations) at line 2 but before it can reopen it in lines 3 and 4, Thread Y interrupts and performs an insert operation. It tries to log the operation and checks for an open log at line 5. But, since it does not find any open log it executes the *else* part of the branch and the insert operation does not get logged. Now, once the dynamic slice is constructed and traversed from the fault point, which is line 7, the last instance of line 5 is in the slice due to the control dependence. Then, going further, it is found that the condition at line 5 obtains its value from line 1 by the RAW dependence. Notice that this value is wrong as it obtains a value of *LOG_CLOSED*. Further, the WAR dependence between lines 5 and 4 indicate the possibility of a race. Further inspection shows that this is indeed the root cause as Thread Y raced past Thread X at this point as the operations in Thread X were not locked. Less than 5 static program statements had to be inspected to find the root cause of the error in this case.

Mysql-II. According to the bug report [2], `mysql` ver. 3.23.56 has an atomicity violation error which is as follows. For some table 't' in the database, when one thread does a row delete from it and another thread does an insert into it in quick succession, though the operations take place in the order they are called, they are logged in the `mysql` *binlog* as done in the reverse order. The `mysql` *binlog* does not reflect the true sequence of operations on the same table and hence it is inconsistent with the state of the table as shown.

—— Log File ——
SET TIMESTAMP=1151980120*;*
insert into b values (1);
SET TIMESTAMP=1151980107*;*
delete from b;
—— End of Log File ——

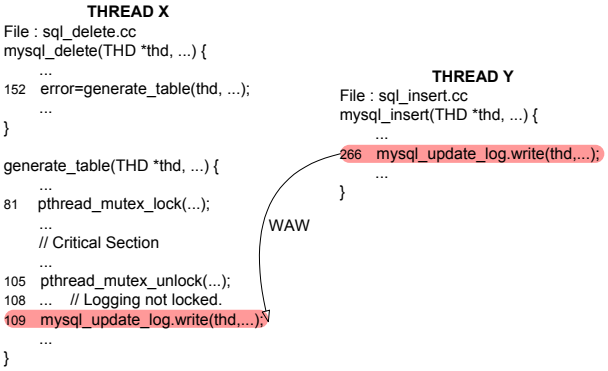

Figure 5. Data race error in `mysql`-II.

Notice that although the delete operation is done first it gets logged after the insert operation. The reason is that line 109 in Figure 5 which performs the write to the *binlog* is not inside the critical section. So, the thread corresponding to the insert operation gets scheduled before this point and the write to the *binlog* happens earlier at line 266. Now, once the dynamic slice is constructed and inspected from the fault point at line 109, the WAW

dependence immediately reveals the race. Notice that this WAW dependence is through a shared file and not shared memory. Again, here less than 5 static statements had to be inspected to identify the root cause.

Apache-I. Apache is an open-source multithreaded program that provides HTTP services on modern operation systems including Unix and Windows. Due to its security, efficiency and extensibility, it has become one of the most popular HTTP servers. But in Apache ver.2.0.48, data races exist in the function *ap_buffered_log_writer* which is implemented to log every accesses to the server. In particular, updating the log buffer in this function is not protected by the lock. Consequently, the server log will be corrupted if two threads execute this function in an interleaved fashion [14].

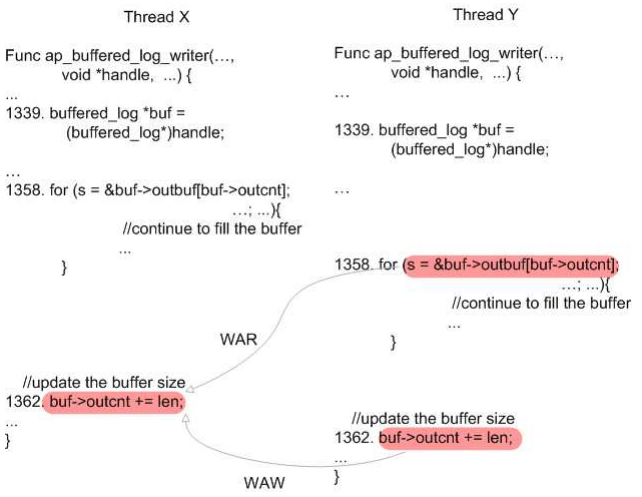

Figure 6. Data race error in apache**.**

Figure 6 shows a possible execution order. As we can see, thread X first filled the buffer from the position *buf→outbuf[buf→outcnt]*. Before it updated the size of buffer (line 1362), thread Y read the old value of *buf→outcnt*, and therefore updated the same locations that were modified by thread X. Clearly, there is a race between line 1362 executed by X and line 1358 by Y. When examining the slice of some fault point in Y where the log corruption is observed, we can discover this race easily as these two statements form a WAR dependence in this execution. Also, line 13 itself is not an atomic operation. In this execution, the race between its two instances, one in X and one in Y, can also be revealed because of the WAW dependence.

A CLR Test Case. *Common Language Runtime* (CLR) is a Microsoft implementation of the Common Language Infrastructure (CLI) standard. In its regression test suite, a data race exists in a test case used to test if an attribute A or B is true for an object [27].

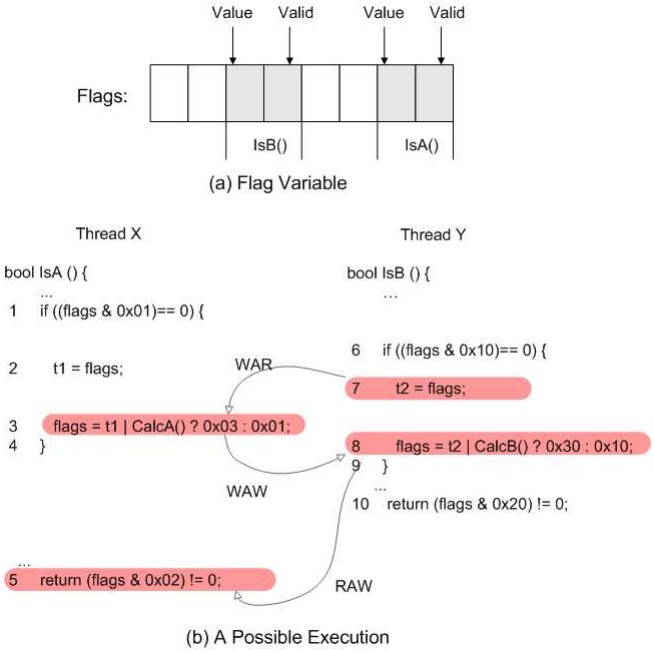

Figure 7. A data race error in a CLR test case.

As shown in Figure 7 (a), the information about whether or not two attributes are true is represented in the one-byte variable $flag$. This one byte consists of one pair of bits, $value$ and $valid$, for each attribute. Function $IsA()$ and $IsB()$ will actually check these two attributes and update the associated pair respectively. Although they access different bits within the byte, the operations are not atomic. Figure 7 (b) shows a possible execution of two threads. As we can see, line 3 which defined the variable *flags* is executed by thread X after line 7 where the variable *flags* is read by thread Y, and before line 8 where this variable is redefined by thread Y. Since there is not any lock protection for *flags*, the WAR and WAW dependences actually form two races. Hence, the return value of thread X in this example will be incorrect. To get the root cause of this error, we only need to inspect no more than four static statements which are in the slice of the statement at line 5.

3. Efficient DDG Construction

As discussed in previous section, to use the dynamic slice for locating race errors in multithreaded programs, we need to trace the program execution to build DDG. The cost of constructing the DDG including Control, RAW, WAR, and WAW dependences can be very high. In this section we present techniques to greatly reduce this cost. First we present optimizations that allow us to capture only

a subset of WAW and WAR dependences without compromising data race detection. Second we show how a highly efficient technique for capturing RAW dependences that we presented in [24] can be extended to also capture WAW and WAR dependences. Finally we present experimental data that evaluates these techniques.

3.1. Transitivity Optimization

This section describes how to apply a transitive optimization that is similar to Netzer's [18] optimization for replay of shared-memory parallel programs. To capture a subset of WAW and WAR dependences (also called *relevant WAW and WAR dependences*) without losing the information for analyzing data races . The key observation is that to find a race, it is enough to capture the WAW and WAR dependences only between two consecutive shared memory accesses.

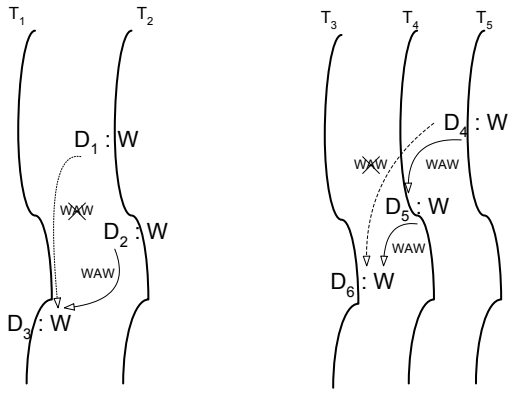

Figure 8. Relevant WAWs to be captured.

Figure 8 shows this using an example. Here, on the left two threads T_1 and T_2 are shown. Now, let there be a WAW dependence between D_1 and D_3 and also between D_2 and D_3. However, only the dependence between D_2 and D_3, which are consecutive accesses, needs to be captured. Because if the WAW between D_1 and D_3 was actually incorrect during the execution due to a race then, obviously, the WAW between D_2 and D_3 becomes incorrect too. In Figure 8, on the right a similar scenario is shown with 3 threads. Here, only the WAW dependences between D_4 and D_5, and D_5 and D_6 are captured. If the WAW dependence between D_4 and D_6 turned out to be a race then at least one of the two captured dependences is also a race error. It is worth noting that in this case, the WAW dependence between D_4 and D_6 needs to be restored in the DDG when we compute the slice of a statement instance whose RAW chain contains D_6. Because the data race error may actually happen between D_4 and D_6, and hence, D_4 needs to be included in the slice.

The argument for WAR is similar, i.e., if a write access W_1 is WAR dependent on a read access R_1 then this dependence is captured only if the write access is the immediate next write after the read access, i.e., W_1 is the first write access following the read.

3.2. Happens-before Relationship

Even with Nezter optimization, potentially, the number of WAW and WAR dependences can still be very large in a multithreaded program. However, not all these dependences correspond to data races. In order to restrict the dependence set to those that can be potential races, the happens-before algorithm from [13] is used. Happens-Before relationship is designed on the assumption that if shared-memory accesses are guarded appropriately using synchronizations then these cannot lead to data races. The happens-before relation provides a partial temporal order of the memory accesses dynamically based on thread synchronizations and order of execution. Now, two memory accesses from different threads that form a WAW or WAR dependence is considered for capture only if a temporal ordering, a happens-before relation, cannot be found between them. However, if there is a temporal ordering based on the happens-before relation then this dependence is not captured as this dependence is not considered as a data race.

The implementation of the happens-before algorithm is done according to the procedure described in the paper by Narayanasamy et al. [16]. A sequencer (S_k), which is nothing but a global timestamp, is associated at that point of a thread's execution where a synchronization operation is executed by the thread. All the different sequencers have a strict time ordering. Any memory access M of any thread falls between two sequencers S_M and S'_M. For instance, in Figure 9 the write to memory location $0xAF$ falls between the interval formed by sequencers S_1 and S_2. Now two memory accesses i and j belonging to different threads that resulted in a WAW or a WAR dependence is not considered as a race if their sequencer intervals do not overlap, i.e., $S'_i < S_j$ or $S'_j < S_i$. Otherwise, this dependence is a race and is captured.

Figure 9 illustrates this where two threads are shown to be executing with sequencers associated at points where the threads executed synchronization operations. Now, the WAW dependence between D_1 and D_2 is not captured because D_1 strictly happens-before D_2 according to the sequencer intervals encompassing them; this dependence is not considered as a race. However, in the case of D_2 and D_3, where there is a WAW dependence, this is viewed as a race because the sequencer intervals which overlap do not reveal any happens-before relationship between D_3 and D_2. Hence, this dependence is considered as a potential race and must be captured.

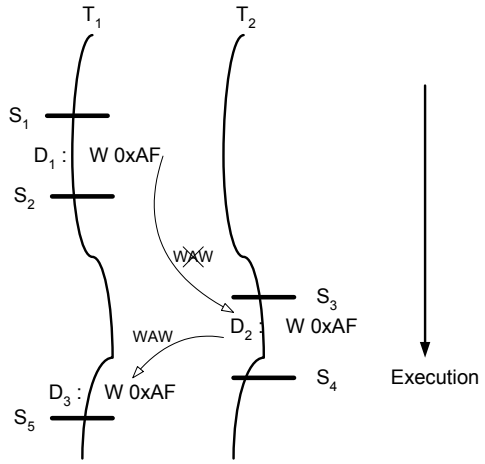

Figure 9. Sequencers in two executing threads to illustrate happens-before relation.

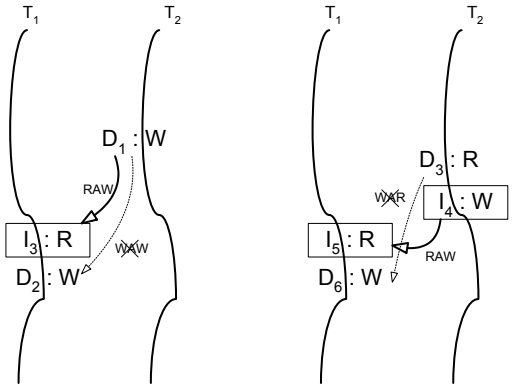

Figure 10. Converting WAW and WAR into RAW dependences.

3.3. Capturing Dependences

We recently presented an execution reduction technique to efficiently collect the relevant subset of dynamic dependences in multithreaded programs [24]. The key idea is to trace the part of the whole execution that is relevant to the fault. In this framework, the original execution is first logged by a logging/replay tool. Due to the scheduling, the logged execution can be viewed as a sequence of *execution intervals* of different threads. When a fault is observed, the execution is replayed and traced to find the dependences among the threads and their execution intervals. The tracing mechanism is optimized to track shared memory dependences, and is therefore *light-weight*. The dependence information is preserved in *thread dependence graph* (TDG). Based on this dependence graph a small set of execution intervals from subset of threads that are relevant to the fault are identified and replayed again with tracing turned on so that the DDG can be constructed.

This technique is very effective in reducing the overhead of DDG construction. However, this work only captured RAW and control dependences when building the TDG. Next we describe how to extend this framework to capture WAW and WAR dependences. The key idea is to convert the capturing of WAW and WAR dependences into equivalent RAW dependences. As shown in Figure 10, every static write instruction in the program is instrumented with a read instruction to the same address immediately before it. Similarly, every static read instruction in the program is instrumented with a write to the same address, writing the same value as is read, just after it. Now, this does not affect the correctness of the program. In Figure 10, on the left the write at D_2 is preceded by the instrumented read, I_3. Now, the WAW dependence between D_1 and D_2 is inferred by the RAW dependence between I_3 and D_1. There is a RAW dependence between I_3 and D_1 because D_1 and D_2 have to be consecutive memory accesses according to the previous section. Hence, the read at I_3 will get its value from D_1. On the right, I_4 and I_5 are the instrumented instructions for the read and write at D_3 and D_6 respectively. The WAR dependence between D_3 and D_6 is inferred by the RAW dependence between I_4 and I_5. The instrumented instructions are clearly marked in order to differentiate between the original RAW dependences and the synthetic RAW dependences. Notice that the ability to do this conversion is possible because all WAW and WAR dependences that need to be captured are between two memory accesses that are consecutive. Also, it should be noted that this instrumentation needs to be done only for those reads and writes that can potentially access shared memory. Once this conversion is done, the *ER* system can be used to capture the RAW dependences, both original and induced. The WAW and WAR dependences can later be recovered from the RAW dependences by looking at RAW dependences between instructions which involved an instrumented load or store or both.

3.4. Experimental Evaluation

System Implementation. Our system has been implemented on top of the *Execution Reduction (ER)* system [24] which uses *checkpointing* and *logging* to reduce the execution foot-print of the faulty execution by eliminating the portion of the execution that is not relevant to the fault. The ER system uses *jockey* [22] to perform checkpointing and logging for replay. During execution, even before the application can execute, jockey takes control and scans the application binary for system call instructions. It then redirects these calls to a jockey handler and lets the application execute. During system calls, jockey logs events, scheduling decisions, creates checkpoints, etc. Scheduling of the user-level threads can be controlled by jockey because it uses its

own thread libraries and any current thread is descheduled only at a system call boundary. Checkpointing is achieved by retrieving the layout of the application's virtual space and dumping all virtual memory segments that belong to the application. The output of the *ER* system is the reduced execution which exists in the form of a jockey event log. The reduced execution can then be replayed using jockey by presenting the event log and the application binary and will result in the fault. Notice that execution reduction already reduces the size of the dynamic slice to be inspected since the execution foot-print has been reduced.

Next, to capture the various dependences, the ER system performs dynamic instrumentation. Note that although the ER system only captures the RAW data dependences along with the control dependences, this is enough since we convert all WAW and WAR dependences into RAW dependences, as mentioned in Section 3.3. To perform dynamic instrumentation, the *valgrind* [17] system is used which can handle x86 binaries. The reduced execution is replayed with jockey but within the valgrind system. The valgrind system calls an instrumentation function just before a basic block is to be executed for the first time. The instrumentation transforms the basic block and rewrites the code cache with the instrumented basic block so that future calls to execute this basic block does not have to go through the instrumentation process. The code cache of a basic block can be invalidated which will cause the instrumentation function to be called when this basic block executes again. Here, we could either modify or turn off the instrumentation. Hence, the instrumented code can be dynamically manipulated. Now, the various dependences can be collected by instrumenting the loads and stores in every basic block accordingly and replaying the execution.

Experimental Setup. The multithreaded benchmarks we used for our experiments are shown in Table 1. For all these benchmarks, we studied the effectiveness of our optimizations on the dynamic slices of their executions. The programs we considered include the ones with bugs that were discussed in the case studies presented earlier. In addition, we also consider parallel programs from the *SPLASH-2* suite [25] – these programs have no faults; thus, they are used only in the evaluation of the effectiveness of our optimizations.

For the programs with bugs we created the following execution scenario for evaluation. First, to illustrate the effectiveness of *execution reduction phase* (ER) [24] in reducing the execution footprint of long running programs, we took the buggy program and created a reasonably long running execution at the end of which the bug triggers the failure. For example, in *mysql*, we create a number of clients and issue queries to the different databases and tables we created. Some of the query operations we have used were among the common ones like *select*, *join*, *insert*, *delete*, *order by*, etc. At the end we perform the query operations that causes the bug to occur. Then, we studied the effectiveness of our optimizations in reducing the dynamic slices of the reduced executions. The executions create between 5 and 10 threads and execute for a couple of seconds before the fault is triggered.

Table 1. Benchmarks and the bugs used in the experiments.

Program	Description	LOC	Description of bugs used
mysqld	Database (ver. 4.0.12) (ver. 3.23.56) (ver. 4.0.16)	508 K	a) Data race bug (mysql-1), reported in [1] b) Data race bug (mysql-2), reported in [2] c) Data race bug (mysql-3), reported in [3]
apache	(ver. 2.0.48)	191 K	Data race bug (apache-1) reported in [14]
CLR testcase	A test case for CLR	896	Data race bug (CLR-TestCase) reported in [27]
splash-2 suite	parallel applications [25]	on average 6 K	NONE

For SPLASH-2 [25] programs, we could not find any reported harmful data races. Hence, we did not conduct experiments for execution reduction on these programs but we could study the effectiveness of our proposed optimizations on reducing the dynamic slices by creating small successful program runs. The various SPLASH-2 program runs created 4 threads and executed for a few seconds.

Now, we present the results of the various experiments conducted to evaluate the efficiency of the dynamic slicing described in this section.

ER Phase. Table 2 shows the number of threads, thread execution intervals (*TEI*) and the number of instructions in the original and reduced executions of the buggy programs. Execution reduction [24] was able to reduce the execution foot print (instruction count) to between 15 % and 85 %. Notice that this would mean that the size of the dynamic slices of the faulty execution is also reduced.

Post-ER phase. After *ER* has reduced the execution foot print by elimination of irrelevant portions of the executions, the size of the dynamic slices of the reduced execution were further reduced by applying the various optimizations proposed in the paper. Table 3 shows the number of inter-thread dependences before and after the optimizations. The number under the column *Before* gives the number of dependences before any optimization is applied. The number under the column *Transitivity* is the number of dependences after the transitivity optimization [18] is applied and

Table 2. Effectiveness of ER and the trace size.

Programs	ER Phase						Data dependences after ER				
	Orig. Numbers			Numbers after ER			RAW		Opt. WAW	Opt. WAR	Total
	Thread	TEI	Inst.	Thread	TEI	Inst.	Intra-	Inter-	Inter-	Inter-	
Mysql-I	8	62	89.2M	5	26	23.1M	1.4M	8998	781	448	1.4M
Mysql-II	10	58	7.8M	4	14	3.5M	1.3M	7832	472	236	1.3M
Mysql-III	8	78	86.3M	5	32	13.9M	4.7M	35209	379	131	4.7M
Apache-I	5	96	130.2M	3	30	20.5M	11.1M	15572	207	34	11.1M
CLR-TestCase	6	14	2.4M	3	6	1.2M	81664	452	47	3	82166

column *Final* shows the final number of dependences after the happens-before optimization is also applied. The data shows that the optimizations can reduce the number of dependences that have to be captured by up to 4 orders of magnitude. Table 2 also shows the itemized count of RAW, WAR and WAW dependences of the reduced executions of the buggy programs after applying the optimizations. It also shows the number of intra-thread RAW dependences which have to be captured in the dynamic slice.

Table 3. Capturing inter-thread dependences with the transitivity and happens-before algorithm (M - Million, B - Billion).

Program	Instrs.	Inter-Thread Dependences		
		Before	Transitivity	After H-B (Final)
Fmm	92 M	86 M	10290	4217
Barnes	4.3 B	81.3 M	91185	84825
Water-ns	1.3 B	2 M	356	150
Water-sp	1.1 B	1.6 M	244	156
Radiosity	907 M	2.1 B	167341	153379
Mysql-I	23.1M	5.1M	13828	10227
Mysql-II	3.5M	1.5M	15226	8752
Mysql-III	13.9M	4.7M	50399	35719
Apache-I	20.5M	1.7M	16598	15813
CLR-testcase	1.2M	20772	634	502

Table 4. Time overhead.

Program	Baseline (Valgrind)	Overhead of Tracing	
		plain	optimized
Mysql-I	0.27	2.08	1.75
Mysql-II	0.22	2.17	1.66
Mysql-III	0.68	9.88	7.90
Apache-I	0.73	5.58	4.18
CLR-TestCase	0.10	1.34	0.93

Tracing Overhead. Table 4 shows the overhead of tracing the dependences for constructing the slice. Since *valgrind* was used to trace the programs, we set the baseline to include the overhead imposed by *valgrind*. Hence, *baseline* shows the running time of the reduced executions of the buggy programs under valgrind but with no tracing involved. The data under column labeled *plain* and *optimized* shows the overhead of tracing the dependences with and without the optimizations respectively. The data shows that although applying the optimizations involves a lot of processing, the tracing time decreases as the number of dependences to be collected also significantly decreases. The trade-off between processing and I/O is favorable to optimizations.

4. Conclusion

In this paper, we have shown that dynamic slices of multithreaded programs must also contain inter-thread **WAW** and **WAR** dependences in order to effectively capture bugs due to data races. We have shown how to compute and traverse the slice in the presence of these additional dependences and also proposed optimizations that can reduce the number of dependences to be captured but still retain the capability to detect all races. Experiments have shown that the proposed techniques which aid the *execution reduction framework* [24] reduce the slices of the faulty executions by up to 4 orders of magnitude.

Acknowledgements This work is supported by NSF grants CNS-0810906, CNS-0751961, CCF-0753470, and CNS-0751949 to the University of California, Riverside.

References

[1] Mysql atomicity violation-1 : http://bugs.mysql.com/bug.php?id=791.
[2] Mysql atomicity violation-2 : http://bugs.mysql.com/bug.php?id=169.
[3] Mysql atomicity violation-3 : http://bugs.mysql.com/bug.php?id=2011.
[4] www.mysql.org.

[5] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 223–240, London, UK, 1993. Springer-Verlag.

[6] M. Christiaens and K. D. Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium*, pages 105–116, Berkeley, CA, USA, 2001. USENIX Association.

[7] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, New York, NY, USA, 1991. ACM Press.

[8] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *LCPC : The International Workshop on Languages and Compilers for Parallel Computing*, pages 497–511, 1992.

[9] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[10] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, pages 222–231, Washington, DC, USA, 1995. IEEE Computer Society.

[11] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *ISSTA : Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 66–79, 1994.

[12] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07: Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 54–64, New York, NY, USA, 2007. ACM Press.

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

[14] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, pages 1–5, 2005.

[15] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33, New York, NY, USA, 1991. ACM.

[16] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31, New York, NY, USA, 2007. ACM Press.

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07 : Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[18] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[19] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[20] J. Rilling and H. F. Li. Predicate-based dynamic slicing of message passing programs. In *ICSM'02: Proceeding of the IEEE International Conference on Software Maintenance*, pages 133–142, 2002.

[21] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[22] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG : Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 69–76, 2005.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 27–37, New York, NY, USA, 1997. ACM Press.

[24] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 207–218, New York, NY, USA, 2007. ACM.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[26] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[27] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.

[28] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 81–91, New York, NY, USA, 2006. ACM Press.

[29] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA'07: Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 121–132, 2007.