Enabling Tracing Of Long-Running Multithreaded Programs Via Dynamic Execution Reduction

Sriraman Tallam
Chen Tian
Dept. of Computer Science
University of Arizona
{tmsriram,tianchen}
@cs.arizona.edu

Xiangyu Zhang
Dept. of Computer Science
Purdue University
xyzhang@cs.purdue.edu

Rajiv Gupta
Dept. of Computer Science
University of Arizona
gupta@cs.arizona.edu

ABSTRACT

Debugging long running multithreaded programs is a very challenging problem when using tracing-based analyses. Since such programs are non-deterministic, reproducing the bug is non-trivial and generating and inspecting traces for long running programs can be prohibitively expensive. We propose a framework in which, to overcome the problem of bug reproducibility, a lightweight logging technique is used to log the events during the original execution. When a bug is encountered, it is reproduced using the generated log and during the replay, a fine-grained tracing technique is employed to collect control-flow/dependence traces that are then used to locate the root cause of the bug. In this paper, we address the key challenges resulting due to tracing, that is, the prohibitively high expense of collecting traces and the significant burden on the user who must examine the large amount of trace information to locate the bug in a long-running multithreaded program. These challenges are addressed through *execution reduction* that realizes a combination of logging and tracing such that traces collected contain only the execution information from those regions of threads that are relevant to the fault. This approach is highly effective because we observe that for long running multithreaded programs, many threads that execute are irrelevant to the fault. Hence, these threads need not be replayed and traced when trying to reproduce the bug. We develop a novel lightweight scheme that identifies such threads by observing all the interthread data dependences and removes their execution footprint in the replay run. In addition, we identify regions of thread executions that need not be replayed or, if they must be replayed, we determine if they need not be traced. Following execution reduction, the replayed execution takes lesser time to run and it produces a much smaller trace than the original execution. Thus, the cost of collecting traces and the effort of examining the traces to locate the fault are greatly reduced.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*

General Terms

Algorithms, Measurement, Reliability

Keywords

debugging, checkpointing, event logging, replay, control flow and dependence tracing

1. INTRODUCTION

Debugging long running, multithreaded applications is a challenging task due to two reasons. First, they are generally non-deterministic, that is, two executions on the same input could behave differently depending on the order in which the threads were scheduled. While the bug may manifest itself in one execution, it need not do so in another execution. Second, it is expensive to collect and store dynamic traces which may be needed to locate a bug. To address the first problem, when designing a debugging framework for multithreaded programs, it is important to make use of a checkpointing/logging infrastructure that can precisely log the important events so that when a bug manifests itself, it can be reproduced during the replay of the current execution. However, solving the second problem by providing a fine-grained tracing mechanism that is practical for long-running programs is a far more challenging task. Support for fine-grained tracing is needed so that the fault can be later analyzed and the root cause of the bug can be detected. It is very important that the traces collected are small because long traces create two problems. First collection and storage of long traces is very expensive in terms of execution time and space needs. Second, the greater the amount of trace information, the greater is the effort required on part of the user to inspect them and locate the root cause of the bug. In the remainder of this section we discuss these techniques (checkpointing/logging and tracing) in more detail and propose a framework that can effectively address the above issues by integrating logging and tracing. This framework is practical due to our novel idea for generating small traces through *execution reduction.*

Checkpointing/logging/replaying is an attractive technique, the merit of which is its capability to replay from the

intermediate points of the execution at which checkpoints are created. It was invented to facilitate debugging parallel and distributed programs [16, 26]. It quickly gained popularity in general application debugging [19, 20]. A lot of research has been carried out on how to reduce its cost [25, 15] and improve its usability [22]. However, most of the existing checkpointing techniques focus on how to faithfully replay an execution. They do not discuss what to do with the replayed execution and simply suggest that the replayed execution can be debugged with general debuggers such as gdb. However, these debuggers are usually less powerful than tracing based tools.

Tracing techniques must be supported in an effective debugging framework as tracing allows us to closely inspect the program and detect the root cause of the bug. Heavy-duty dynamic analyses can be performed on the traces efficiently once they have been gathered. As a result, software errors become much more recognizable if appropriate traces are gathered. For example, dynamic slicing, proposed by Korel and Laski [11], is a tracing based technique to help programmers in the process of debugging. The dynamic slice of a value computed at a program point in the execution trace includes all those executed statements which were directly or indirectly involved in the computation of the value. Prior work [9, 30] has demonstrated that dynamic slicing is quite effective in automatically isolating the cause effect dependence chain from the root cause to the failure point. Unfortunately, tracing based techniques do not scale for long executions even though state-of-the-art techniques can achieve a space efficiency of 0.1-4 bits per instruction [29, 8]. A simple task such as starting Mozilla and browsing a html page creates traces in the order of giga bytes.

We collected a set of multithreaded benchmark programs shown in Table 1. Table 2 shows the sizes of control flow traces and dependence traces for sample runs of these multithreaded programs. It is clear from the data that the sizes of the traces produced is large and the runtime overhead of their collection is substantial. Also, even if the traces are collected, it can be a huge burden on the programmer to inspect these traces and get to the root cause of the bug.

Table 1: Benchmarks and the bugs used in the Experiments.

Program	Description	LOC	Description of bugs used
mysqld	Database (ver. 3.23.56) (ver. 3.23.56) (ver. 3.23.48)	508 K	a) Mem. bug (mysql-1), reported in [2] b) Atomicity bug (mysql-2), reported in [3] c) Mem. bug (mysql-3), reported in [4]
prozilla	Download Accelerator (ver.1.3.5.1)	16 K	a) Mem. bug (prozilla-1), reported in [5] b) Mem. bug (prozilla-2), reported in [6]
proxyc	small C proxy	219	a) SIGPIPE bug (proxyc-1), Found using Change Log
axel	Download Accelerator (ver. 1.0a)	3 K	a) Mem. bug (axel-1), reported in [7]
pftp	Port File Transfer	8 K	NONE
balsa	Email client	100 K	NONE
evolution	email, address book	438 K	NONE

Table 2: Trace sizes of multithreaded programs for small runs and their collection time in seconds.

Program	Num of Threads	Exec. Time (secs.)		Control Trace	Dep. Trace
		Original	Traced		
mysql	10	13	2886	6 GB	21 GB
evolution	10	11	179	87 MB	390 MB
balsa	7	17	1787	92 MB	209 MB
pftp	3	10	903	543 MB	482 MB
proxyc	9	10	880	1360 MB	456 MB
axel	3	8	184	313 MB	456 MB
prozilla	5	8	2640	2 GB	6 GB

From the above discussion we can see that an integration of checkpointing and tracing within a single infrastructure would be very useful. It will allow us to replay the fault and collect traces during replay, to debug the program and find the root cause of the bug, and will also be efficient if the tracing overhead can be minimized. In this paper, we propose one such framework which can achieve the above objective. Our approach essentially consists of two main steps. First, we log the events of the original run and checkpoint at regular intervals in order to be able to faithfully replay the execution from any checkpoint or the start of the program. In case the execution fails, like due to a segmentation fault, during the replay we turn on our tracing infrastructure that can generate fine grained traces which allows us to look closely at the execution and find the root cause of the bug. To make this framework practical, we minimize the tracing overhead by reducing the execution footprint. We propose the novel idea of *execution reduction* that reduces the length of the execution for replay and tracing by exploiting the observation that most threads that get executed are not directly relevant to the fault and need not be replayed or traced.

Many multithreaded and long-running applications such as server programs are event driven and, usually, a thread is spawned to service a new request from a client. Most of the threads execute independently of the other and a fault that occurs in one thread, which can even lead to a crash, is not influenced by a majority of the other threads. We exploit this observation during replay as follows. We only replay those threads that cause the fault to occur and prevent the remaining threads from executing as they are irrelevant to the fault. The result is that the replayed execution is exactly what is necessary to reproduce the bug and hence, the trace that it generates is shorter and also exactly captures the program behavior that led to the fault. To find the threads that are relevant to the fault, we have developed a technique that can detect the various dependences between the executing threads very efficiently in time and space. Using the dependence information between threads, we can obtain the set of those threads that contributed to the fault and those that were irrelevant to the fault. Now, we trace the execution by only replaying the threads relevant to the fault. The resulting trace sizes of the shortened executions are 3 to 6 orders of magnitude smaller than the original trace size and they are generated with an overhead smaller than the original traced execution by factors ranging from 4 to 5618.

Our algorithm essentially consists of three phases: *Logging phase*; *Execution Reduction Phase*; and *Replay Phase*. Logging Phase corresponds to the original program run during which the checkpointing and logging infrastructure is turned on. This phase produces the record of all the events, that is, the *event log*. In the case a bug is encountered, during the

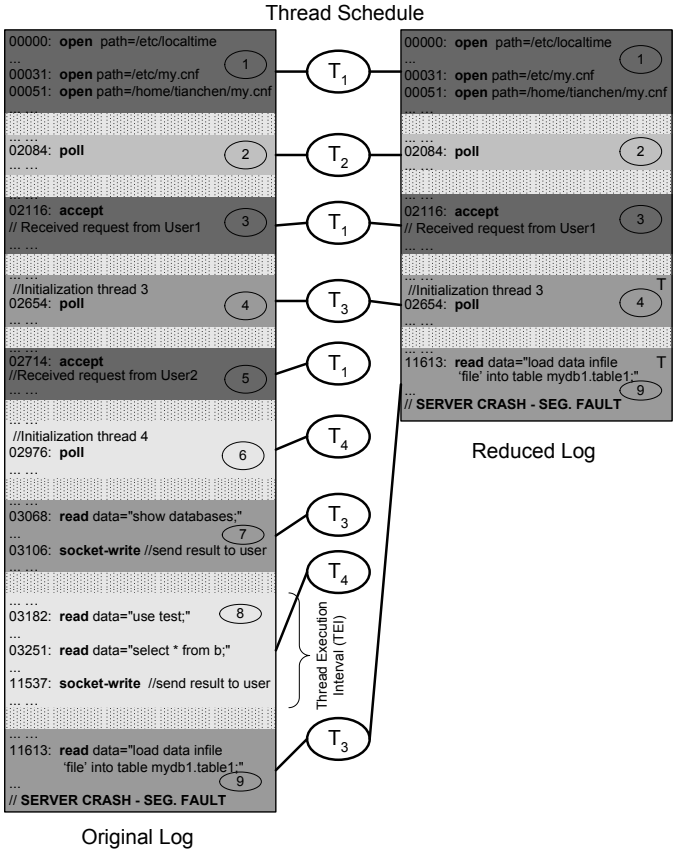

Figure 1: Motivation - MySql (Seg. Fault) Memory error, the different threads ($T_1, \ldots, T_4$) are marked and the thread execution intervals ($1, \ldots, 9$) are numbered. The reduced log shows the intervals that are replayed. The Symbol 'T' in the reduced log shows the only intervals that are traced.

Replay Phase the event log can be used to replay the execution from a checkpoint or the start of the program. During the Replay Phase tracing is turned on to collect control-flow and/or dependence traces that are then used during debugging for fault location. Since the cost of the Replay Phase can be very high due to tracing, to reduce the cost of this phase it is preceded by the Execution Reduction Phase. In this phase first the dynamic dependences between the various threads are discovered. Then using this information, the subset of threads and their execution intervals that must be replayed in order to reproduce the fault are identified. Only the identified execution intervals of the subset of threads are retained for the Replay Phase and the event log is pruned to enable replay of the reduced execution. The key contributions of this paper are as follows.

- We propose a framework to debug long-running, multithreaded programs by combining two powerful techniques, checkpointing/logging and tracing. Logging is turned on during the original program run and, in case a bug is encountered, tracing is turned on during replay. Thus, tracing is used when needed.
- To limit the trace collection overhead and trace size, we propose a novel *execution reduction* technique that replays only relevant parts of the execution by automatically removing irrelevant threads and thread execution intervals.
- To discover and eliminate irrelevant threads, and thread execution intervals, we propose a dynamic technique that finds dynamic shared memory dependences between threads accurately and efficiently.

The rest of the paper is organized as follows. Section 2 gives an overview of our approach using an example. Section 3 describes the three phases of our debugging framework in detail. Section 4 discusses the implementation aspects of our framework. Section 5 presents the results of our experiments. Related work is presented in section 6 and conclusions are presented in section 7.

2. A MOTIVATING EXAMPLE

In this section, we motivate our approach using as example, a memory bug in the *MySql* database server. *MySql* [1] 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. Next we describe a known error in a *MySql* version and we show how to apply our approach of replay to trace on the faulty execution.

According to [2], *MySql* version 3.23.56, has a memory error which is as follows. When a thread tries to load data into a table without explicitly connecting to the database, the field that stores the database name is accessed without checking for the *NULL* value. This causes the server to crash at this point. Let us consider an execution in which, after processing a set of queries, the above fault is exercised.

Logging Phase. To begin with, the server is run with light weight logging enabled, that is, the events are logged and checkpoints are performed at fixed intervals. Figure 1 shows the events recorded in the event log. T_1, T_2 T_3, and T_4 refer to four unique threads created. The event log also shows the points where a thread is descheduled and another thread is scheduled. We refer to a region in the log corresponding to the maximal set of consecutive events from the same thread as a *thread execution interval* (TEI). The log in Figure 1 shows 9 thread execution intervals.

The queries and the activities of the corresponding threads are as follows:

- Thread T_1 is the startup thread that handles new connections and creates threads to service requests.
- Thread T_2 is created by T_1 to handle signals.
- Thread T_3 is created by T_1 to handle a user. This user first looks at all the databases and then crashes the server by issuing the "load" command.
- Thread T_4 is created by T_1 to handle another user. This user does a "select" operation on table 'b' in database 'test'.

The server crashes at TEI 9 and the program is taken to the next phase of our framework. Figure 2 shows the root cause for this bug. Notice how the database field(thd→db) is accessed without checking for an invalid value. The fix is to place a check before the access and report an error if the value is invalid, instead of crashing.

```
sql/mysql_load.cc:
int mysql_load (THD *thd,...)
{
        ...
150     if( ...
151     ... +strlen(thd->db) + 3 <
152      FN_REFLEN)
    ...
}
```

Figure 2: Source Code - MySql Memory Error, root cause

Execution Reduction Phase. Once it is known that a fault has occurred (e.g., program crash has occurred), we would like to replay the fault and collect the trace during replay to assist the user in debugging. However, first the execution reduction phase is used to determine the subset of computation that needs to be replayed and traced.

The first step in this phase is to identify and remove the threads from the event log that did not contribute to the bug. This reduces the execution time of the program and keeps the resulting traces small. Alternatively, if we try to generate the trace for the entire execution by replaying it with the tracing turned on, then the resulting trace sizes are as high as 16 GB for 15 seconds of execution. It is easy to see that if the programs had run for a long time before the fault occurred then the trace sizes would become unmanageable.

Lets consider the threads in our example. We notice that thread T_4 is irrelevant to the fault. Intuitively, it can be seen that the bug would have still occurred even if the second user did not exist. The queries corresponding to the execution of thread T_4 are completely independent of the queries corresponding to T_3. Hence, we first remove all the events from the replay log corresponding to T_4. In the reduced log in Figure 1, notice that the TEIs 5, 6, and 8 do not exist. These correspond to the creation and execution of thread T_4 in the original log.

Now that we have removed irrelevant threads we proceed to the next step in this phase which identifies irrelevant TEIs. All the remaining threads are relevant to the bug but not all of their TEIs are relevant. In Figure 1, TEI 7 corresponding to T_3, which is the execution corresponding to a "show databases" query from the user, is irrelevant to the fault. Hence, this is also removed from the replay log. The reduced log now shows only the relevant TEIs that caused the fault. Replaying the program using the reduced log generates the fault much faster.

Replay Phase. In this final phase we replay the program using the reduced log with the tracing infrastructure turned on. Since the execution contains only the necessary TEIs, the traces produced are much smaller. We further optimize the size of the traces in this phase by exploiting the following observation. *We observe that even though all the TEIs in the reduced log have to be replayed to produce the fault, not all of them have to be traced.*

For instance, thread T_1 is present only to create the faulty T_3, and thread T_2 to handle signals. The code that is executed in T_1 and T_2 does not contain the root cause. Hence, only thread T_3 needs to be traced, that is, TEIs 4 and 9 in the reduced log. The original trace is reduced in size by 99.99% and this reduced trace captures the root cause as desired. The user then inspects the generated trace and discovers the root cause of the bug.

3. AUTOMATED EXECUTION REDUCTION

In this section we identify the types of dynamic dependences that must be found to automatically perform execution reduction as well as efficient dynamic algorithms used to identify these dynamic dependences. Before we present the above details we briefly describe the types of events that are recorded in the replay log.

When the application is being executed in the real world environment, we execute it with a lightweight checkpointing/logging mechanism turned on. Non-deterministic events are recorded as and when they happen. In addition, the program is checkpointed at regular intervals. The logging is to ensure that the program can be replayed to reproduce a fault if one occurs. Let us discuss some of the events that must be recorded in the log in order to correctly replay the execution of the multithreaded program. The *thread scheduling* events of a multithreaded program are the most important events that need to be captured since they can vary from one execution to another. The replay log captures all the events where a thread was descheduled and a different thread was scheduled. It should be noted that the order in which the different threads access shared memory, which must be preserved to successfully replay a multithreaded program, need not be captured explicitly. By recording the scheduling information, we ensure that when the program is replayed as per the schedule, the order of shared memory accesses by the different threads does not change. This holds only when the user level threads are executing in a uniprocessor environment. Notice that the order in which a single thread accesses memory is preserved in the control flow of the execution and hence, need not be explicitly recorded. To summarize, by preserving the thread scheduling of the original execution we guarantee that the order of memory accesses in the replayed program is exactly the same as the original program. Some of the other important events that need to be captured are external events like signals, interrupts, and IO reads and writes. Reads and writes to files need to be logged along with the file offsets and the size of the read or write in order to undo these operations and restore the original file contents when commencing replay.

When a fault is encountered, we carry out execution reduction before carrying out replay for the purpose of collecting traces to aid in debugging. Execution reduction is critical because even if the program is replayed from the most recent checkpoint, the trace that is generated can be very long and its generation can take a long time. Execution reduction is based upon two types of information: identification of irrelevant threads; and identification of irrelevant thread execution intervals. The algorithms for identifying irrelevant threads and irrelevant thread execution intervals are described in the next two subsections.

3.1 Discovering Irrelevant Threads

Lets consider the problem of identifying threads that are irrelevant to the fault, that is, the execution of these threads does not influence the execution of the threads that resulted in the fault. To achieve this goal, we need to know the interactions or dependences between the different threads. Thread interactions can take place via *events*, *files*, or *shared memory*. Examples of event interactions are actions such as thread creation and join involving two or more threads. File interactions occur when threads communicate by reading

and writing from files. The most frequent type of thread interactions are through the use of shared memory regions.

To detect whether a thread is relevant or irrelevant to the fault, we need to obtain the information about the three kinds of dependences between the threads. To replay a thread T_i, we also replay another thread T_j if and only if thread T_i depends on thread T_j. Once we have the dependence information, we construct a *Thread Dependence Graph* (TDG). We then identify the set of relevant threads, $REL(T_i)$, for any thread T_i by traversing the TDG. To replay the execution of thread T_i exactly, it is necessary and sufficient to only replay those threads that are in the set $REL(T_i)$. Given this information, we can detect and eliminate all the threads that are irrelevant to the faulty thread by pruning the replay log. We will then only have those threads that contributed to the fault resulting in reduction in the execution. The definitions of TDG and Relevant Threads are given below.

Definition 1.(*Thread Dependence Graph (TDG)*) The Thread Dependence Graph of a multithreaded program execution, $TDG(N, E)$, consists of a set of nodes N and a set of directed edges E where each node $n_i \in N$ corresponds to a unique thread T_i that was created in the current run and each edge $(m_i \rightarrow n_j) \in E$ indicates that there is a dependence path from thread T_i to thread T_j, that is, T_j is dependent on thread T_i. Also, each edge is annotated with one or more of the symbols in the set $\{File, Event, SharedMem\}$ to indicate the type of dependence(s).

Definition 2.(Relevant Threads) The set of relevant threads corresponding to a thread T_i, $REL(T_i)$, is defined as $REL(T_i) = \{T_j | T_j \in \Gamma$ and $\exists$ a dependence path from T_j to $T_i\}$ where Γ is the set of all threads in the current run.

Next we discuss each of the three types of interthread dependences in detail. In particular, we discuss how they are identified.

Event Dependences. The replay log contains explicit information on all the thread events (e.g., thread creation and termination, synchronization events such as join, etc.) and hence the log can be analyzed to obtain all event dependences between threads. Hence, by inspecting the log and looking at the records corresponding to these events, the various event dependences between the different threads can be detected. For example, in the replay log in Figure 1, the events in TEI 1 corresponding to thread T_1 indicate that a new thread T_2 was created and scheduled to run in TEI 2. Hence, we infer that thread T_2 is dependent on thread T_1 by the parent-child relationship. Notice that this means T_1 has to be replayed in order to replay T_2 whereas the reverse is not true. *In summary, the replay log captures all the event dependences between threads and a simple scan of this log is enough to discover all of them.*

File Dependences. We now discuss how to discover dependences between threads due to file operations. A file data dependence exists from thread T_i to thread T_j if thread T_j reads from an offset in any file F that was written to by thread T_i. This implies that for successful execution of thread T_j, thread T_i must also be replayed. Dependences between threads due to files can be directly obtained from the replay log. The replay log records information on the files that were read or written by every thread, the offsets from which the reads and writes took place and the size of the operation. This is done primarily to restore the contents of the files while commencing replay. *Hence, by scanning the replay log all file dependences between threads can be retrieved.*

Shared Memory Dependences. Let us now discuss how to discover the most common interactions between threads that result in shared memory dependences. There exists a shared memory dependence from thread T_i to thread T_j if T_j reads a value from any memory address 'a' that was written by T_i. We say that T_j is dependent on T_i since T_i generates the value and has to be replayed for successfully replaying T_j.

Shared Memory dependences between threads cannot be simply obtained from the replay log. Recall that to make the logging scheme lightweight, explicit capturing of information unnecessary for replay must be avoided. By capturing the thread schedule in the log, the need of capturing the shared memory dependences does not arise. Therefore, to obtain shared memory dependences, we must replay the program and track these dependences as they occur using a mechanism for detecting shared memory dependences. This mechanism must track shared memory dependences between threads and output thread ordered pairs (T_i, T_j) that are involved in at least one shared memory dependence. Note that to construct the TDG, we do not output every occurrence of a dependence between a pair of threads.

It is desirable that the technique that detects shared memory dependences has low overhead. Even though this phase is carried out in the debugging stage of the program, unreasonable delays is not desirable to the user who is debugging the code. One approach that does not involve runtime overhead could be based upon static analysis [23, 21]. This approach has the disadvantage of producing a conservative TDG using which fewer threads may be identified as being irrelevant. Another issue is that dynamic opportunities for eliminating irrelevant threads will be lost. During a given execution, a potentially shared memory region may however, be accessed by just one thread. The static approach cannot take advantage of such opportunities. However, a well designed dynamic approach can take advantage of this information to identify more irrelevant threads.

Let us first consider a *naive* dynamic strategy for detecting shared memory dependences. A *hash table* can be used to maintain, for each address 'a', the *thread id* of the thread that performed the *most recent write* to 'a'. To detect an interthread dependency, when a load operation is performed on address 'a' by thread T_j, we retrieve the thread id T_i that wrote to it last from the *hash table* and then form an interthread dependence if T_i and T_j are different. Although this scheme is straightforward, it is inefficient in both space and time. It is space inefficient because the size of the hash table is as large as the memory footprint of the original program. For large applications, we could potentially run out of memory. It is time inefficient because every load and store that executes must access the hash table. Every store must write the thread id to the corresponding hash entry

```
do_for_every_load_and_store(ThreadId currThread, Address a){
/* RegionMap, array of 2^16 entries, each entry has 2 fields
    isSharedMem bit, firstThread field - initialized to 0 */
// prevThread,prevRegion,prevSharedMem - prev. load / store
currRegion=a >> 16; // higher order 16 bits
Stage I :
    if(prevThread=currThread && prevRegion=currRegion
        && prevSharedMem=False)
        return;
    prevThread=currThread; prevRegion=currRegion;
Stage II :
    entry = RegionMap[currRegion];// Lookup Region table
    if(entry→isSharedMem=False)
        if(entry→firstThread=0)
            prevSharedMem=False;
            entry→firstThread=currThread; return;
        if(currThread = entry→firstThread)
            prevSharedMem=False; return;
        else
            prevSharedMem=True;
            /* stores update shared memory bit
            loads check for dependence with first thread */
            if(load instruction)
                return;
Stage III :
    if(store instruction)
        write_threadid_into_hash_entry(a);
    else  //load instruction
        threadId = read_threadid_from_hash_entry(a);
        if(threadId is valid)
            Track_Dependence(currThread, threadId);
        else
            Track_Dependence(currThread, entry→firstThread);
    return;
}
```

Figure 3: Pseudo-code for detecting shared memory dependences. The code shows the processing that is done for every memory load and store instruction. The 3 stages are clearly marked.

and every load must read it from the hash table. Next we present a scheme that greatly improves the efficiency of the above naive interthread dependence detection scheme. This scheme is efficient in both space and time. This scheme is based upon two optimizations that achieve elimination of majority of the expensive hash table lookups.

The first optimization introduces a new look-up table, called *RegionMap*, such that accesses to this new table are less expensive than accesses to the *hash table*. Often times, the dependence is resolved by accessing the *RegionMap* and hence the need for accessing the *hash table* is eliminated resulting in savings in time. In addition, we will see that the size of the hash table is greatly reduced.

Let us discuss the *RegionMap* in greater detail. We divide the 32-bit virtual memory address into two parts: the higher order 16-bits act as a *region specifier*; and the lower order 16-bits are used as the *offset* address within the region. The region itself can be either a *shared memory region* or a *non-shared memory region*. The *RegionMap* is indexed by the 16-bit region specifier and it contains a bit for every region, called *isSharedMem*, that indicates if the region has been *dynamically observed* to behave as a shared memory region or not. All region bits are initially set to *False* implying that all regions are non-shared memory to start with. The region bit for a region is set to *True* if more than one thread accesses the region. The region table also contains a field, called *firstThread* that stores the identifier of the first thread that wrote to it. This is initially set to an invalid value and is initialized by the *first thread* that writes to it.

Let us now see how an access to the hash table may be avoided by first accessing the *RegionMap*. For a load operation, we look up the *RegionMap* first to see if we are accessing a shared memory region or not. If the region is currently indicated to be non-shared memory, we do not need to do anything further as this load does not involve an interthread dependence. However, if it is a shared memory region we obtain the *threadId* of the store operation involved in the dependence by looking up the *hash table* and check if it is an interthread dependence. For a store operation, if the region is shared memory, we update the hash entry corresponding to the memory address with *threadId*. If the region is not shared memory, we check if the thread performing this store could potentially make it a shared memory region, that is, we compare the current thread's id with *firstThread* id to see if they are different. If they are different, a shared memory region has been detected and the region bit, *isSharedMem*, for this region is set to *True* in the *RegionMap*. The hash entry for the 32-bit address is also updated. However, if the region is still not shared then we have to do nothing further.

From the above operation of the *RegionMap*, and the *hash table*, we have achieved the following. *The size of the hash memory now at most equals the combined sizes of only the shared memory regions and not the total virtual space used.* Hence, this is a huge saving and for the many programs we looked at, the amount of shared memory that is used is much less than the actual memory used. Also, *for loads and stores that do not access shared memory regions, the expensive hash table lookup operation is avoided.*

The second optimization is designed to further reduce the runtime overhead by reducing the *RegionMap* lookups and replacing them with cheaper operations. In this sense this optimization is analogous to the first optimization which reduced the runtime overhead by replacing some of the expensive hash table lookups by cheaper *RegionMap* lookups. This second optimization exploits the *locality in the regions accessed* by most loads and stores. In particular, locality here refers to the characteristic that consecutive executions of the same static load (store) often involve the same region. When this is the case, handling of one region access by a load (store) makes the handling of the next access to the same region by the same load (store) redundant.

Finally, we obtain a three stage algorithm for handling each load and store such that first stage is the cheapest and the last stage (hash table lookup) is the most expensive. While in general a load or store may have to go through all three stages, very often this is not the case and hence the runtime overhead of the three stage scheme is greatly reduced when compared to the runtime overhead of a single stage scheme involving hash table lookup. Next we put all of the ideas together into a three stage algorithm described below (pseudocode is given in Figure 3).

Stage I - Check region of previous memory operation. In this stage, for the load or store that is being processed, if the previous memory operation (load or store) was from the same thread, it accessed the same region, and was found to be non-shared, then we can guarantee that this region will continue to remain non-shared. (The variables *prevThread*, *prevRegion* and *prevSharedMem* contain this information about the most recent load/store operation.) Hence, we do not require a *RegionMap* lookup and we are done processing this memory operation. Due to significant locality of regions, over half of the loads/stores did not proceed beyond this stage.

Stage II - Check RegionMap table and update isSharedMem bit. In this stage, we need to access the *RegionMap* as the locality check in Stage I failed. The *RegionMap* tells us if the region accessed is shared memory or not. For a load or store, if this region is not shared memory we do not need to consult the hash table. However, a check needs to be done to see if this thread's access could potentially make it shared memory and flip the *isSharedMem* bit accordingly.

Stage III - Access the hash table. In this stage, we have determined that the region is shared memory by looking up the region table and therefore we perform the expensive hash accesses. For a load operation we access the hash table to retrieve the thread that wrote to this 32-bit address last and check if it is an interthread dependence. The function *Track_Dependence* does this check. However, if this address was last written to by a thread when this region was not detected to be shared memory, then the contents of the hash memory would be invalid as the thread that wrote to it last did not create the hash entry. In this case, we access the *firstThread* field of this region. We now have the dependence. For a store operation, we update the hash entry corresponding to the 32-bit address.

Table 3: Cost of shared memory dependence tracking for some multithreaded programs.

Program	Staged Tracking		Time	Memory Used	
	Stage I %Ld+St	Stage II %Ld+St	Staged/ Naive	Naive	Staged
mysql	52 %	10 %	55 %	3.6 MB	0.8 MB
evolution	16 %	10 %	64 %	8.4 MB	5.9 MB
balsa	67 %	15 %	73 %	8.1 MB	1.8 MB
proftp	50 %	0 %	50 %	3.3 MB	3.3 MB
proxyC	72 %	16 %	56 %	6.6 MB	1.3 MB
axel	50 %	18 %	12 %	0.3 MB	0.1 MB
prozilla	56 %	19 %	41 %	1.2 MB	0.3 MB
Average	52 %	13 %	58 %	4.5 MB	1.9 MB

We have conducted experiments on some multithreaded long running programs and measured the percentage of loads and stores that terminated at each stage. Table 3 shows the data. It also shows the space overhead of this approach. From this data we can see that on average 52% of all loads and stores terminate at Stage I, that is, they do not require a *RegionMap* or a *hash table* access. Additional 13% terminate in Stage II. Thus, finally, on an average, only 35% of all loads and stores performed hash accesses as they reached Stage III. The runtime overhead of the staged approach is 58% of the naive tracking scheme. Also, on an average, the total memory used by the Staged Tracking approach is only 42% of the memory used by the naive approach.

Note that the region size, which is 16 bits now, can be varied to be coarser or finer. By making it finer, we could determine shared memory space much more accurately but we would lose on the locality optimization. Notice that a region size of 32 bits is basically equivalent to the naive approach. Making the region size coarser could give more opportunities for locality but more regions would become shared memory and hence the locality benefits might not be useful. Hence, the region size is a trade-off between how finely we can detect shared memory and how much locality we could get. We found a 16 bit sized region to work well with our benchmarks.

Eliminating Irrelevant threads. At this point we have the complete thread dependence graph with all dependences detected and annotated. We find the set of threads that are relevant to replaying the fault. The rest of the threads are irrelevant. We prune the replay log to remove all the records corresponding to the irrelevant threads. Now, the reduced replay log has only information on relevant threads and the execution has already been shortened.

3.2 Discovering Dependences Across TEIs

Now that we have eliminated the threads that are irrelevant to the fault, we now discuss how to eliminate irrelevant thread execution intervals (TEIs) from the relevant threads. For this step, we need to detect the interactions between the various TEIs. Notice that we already have the information on the dependences between TEIs that correspond to different threads. Now, we need to find the event, file, and memory dependences between TEIs belonging to the same thread. Event and file dependences across TEIs of the same thread are found using the original replay log. To find memory dependences, we replay the program again, but using the reduced replay log, and use the naive approach described in the last section as the execution has been shortened already.

Now, just as we detected irrelevant threads by using the TDG, analogously, we construct a dependence graph for TEIs and remove all irrelevant TEIs. We prune the reduced replay log further to remove all records corresponding to the irrelevant TEIs. We now have a highly reduced log that contains only relevant TEIs. This completes the second phase of our framework.

3.3 Selective Tracing of Reduced Execution

The reduced replay log contains only those thread execution intervals that need to be replayed. *However, not all TEIs have to be traced.* For instance, a thread's execution trace, that merely created the faulty thread which has a memory error, is not useful as the invalid memory access could not have come from this thread. We identify all such TEIs. During replay, we turn on tracing when a TEI needs to be traced and turn it off otherwise. The overhead of toggling tracing is low as it is done at the granularity of TEIs. At the end of this stage, we get a trace of the faulty execution that is short and contains the root cause of the bug.

4. THE EXECUTION REDUCTION SYSTEM

In this section, we describe the implementation of the ER system that incorporates checkpointing/logging, dependency detection, and (selective) tracing. This system was used to analyze several bugs in long-running multithreaded programs.

Figure 4 shows the system. The system consists of a logging component whose main role is to log the events of the original execution and also create checkpoints at regular intervals. Our system's key component is the dynamic instrumentation engine. It is involved in many steps of the debugging process. It uses the information in the replay log created by the logging infrastructure to replay the multithreaded program exactly. Also, while replaying the program, it can dynamically instrument the binary to detect dependences and collect traces. The information it generates is used to shorten the replay logs by pruning irrelevant

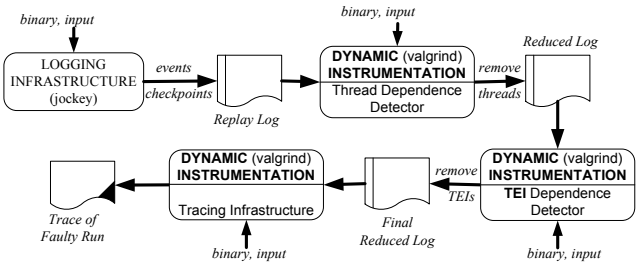

Figure 4: Implementation of our system showing each step of the framework.

threads and TEIs. Since the instrumentation is dynamic, tracing can be turned on and off at run-time. Let us discuss the tools used to perform logging and dynamic instrumentation.

Logging/Checkpointing Infrastructure. We have used the *jockey* user level library [22] to perform checkpointing and logging for replay. *jockey* is a very powerful system that works on most multithreaded programs and is also very easy to use. 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. To summarize, *jockey* related work is performed only during a system call and otherwise, the application executes as though it was unaware of *jockey*. Since *jockey* works only for uni-processor systems, we cannot log the execution of multithreaded programs that run on multiprocessors. However, our approach is general and by using a logging mechanism for multiprocessors [13, 27], our execution reduction techniques can be applied to programs that execute on multiprocessors.

Dynamic Instrumentation Engine. To perform dynamic instrumentation, we have used the *valgrind* [24] system which can handle x86 binaries. The binary is executed with *valgrind* which 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.

The dynamic instrumentation engine forms the core of our framework. Its first job is to parse the log generated by *jockey* and replay the program. For multithreaded programs, the scheduler decisions are the most important events that have to be replayed. We replay the schedules as follows. We know that scheduling decisions in the original program are made only at system call sites by *jockey*'s thread library. When logging the schedule, we also log the number of system calls that the thread executed since it was scheduled and before it was descheduled. In *valgrind*, for a thread that is currently executing, we use the number of system calls executed to decide when to deschedule the thread. When we reach that system call (*valgrind* has event handlers that are called before and after a system call and we use this to count system calls), we force the scheduler to deschedule this thread and switch to the appropriate thread. Hence, we can guarantee that the threads will be scheduled according to the replay log. As we have mentioned before, preserving the schedules will also guarantee that the shared memory dependences of the original execution will be preserved. File events can be replayed exactly if we restore the contents of the modified files. There are some system calls for which *jockey* saves the contents of the original run. For example, in a server program, if a client makes a connection request, the contents of the *socket-read* system call are saved in the *jockey* log. During replay, when the system call *socket-read* is to be executed, *jockey* will return the saved contents instead of executing the system call. We do exactly the same in *valgrind* when we replay the program. When the *socket-read* system call is reached, we do not perform the system call but return the contents saved in the *jockey* log. For the programs we considered, the events that we handled were enough to replay the execution.

Given that we can successfully replay the program in *valgrind*, we now use it to detect shared memory dependences, TEI dependences and, finally, obtain traces. In the thread execution reduction phase, we replay the program and instrument the loads and stores in every basic block according to the algorithm described in Figure 3. Once we have obtained the shared memory dependences, we can now find the irrelevant threads of this faulty execution. Note that the file and event dependences are already available in the replay log. We then use this information to prune the replay log. Similarly, we find dependences across TEIs and further prune the log. Once we have the final reduced log, we use *valgrind* to trace the shortened execution and output the trace. Here, since we do not trace all replayed TEIs, we use *valgrind*'s ability to selectively switch on or off the tracing for a particular TEI.

5. EXPERIMENTS

The multithreaded benchmark programs used in our experiments were already described in Table 1. For all these programs, the trace sizes and the cost of detecting shared memory dependences have been already shown in Tables 2 and 3 respectively. Now, for studying the effectiveness of our entire ER system, we performed experiments with only the buggy versions of these programs.

For each bug, we have created the following execution scenario. We take the buggy program and then create 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 we created. Some of the query operations we have used were among the common ones like *select*, *join*, *insert*, *delete*, *orderby*, etc. At the end we perform the query operations that causes the bug to occur. We have limited the length of the execution to be around 10 seconds for *mysql* and *proxyc*. For prozilla, the length of the execution is around 5 to 7

Table 4: Trace sizes produced by the original and shortened runs (M - million, B - billion).

Program	Number of Basic Blocks					Number of Data Dependences				
	Orig.	R_Thread	R_TEI	SR_TEI	Orig./SR_TEI	Orig.	R_Thread	R_TEI	SR_TEI	Orig./SR_TEI
mysql-1	976 M	19349	16695	1964	490000	1.5 B	30375	25391	3175	470000
mysql-2	733 M	1.1 M	29809	29809	24500	1.1 B	1.27 M	49263	49263	22000
mysql-3	857 M	122 M	24834	9511	90100	1.3 B	188 M	40869	17929	73000
prozilla-1	536 M	106749	81179	81179	6600	720 M	135466	123918	123918	5800
prozilla-2	764 M	764 M	764 M	1.6 M	478	1 B	1 B	1 B	2.6 M	380
proxyc-1	200 M	23736	23736	23736	8400	56 M	6513	6513	6513	8600
axel-1	55.4 M	7734	7734	1622	34000	53.9 M	5119	5119	1156	46600

Table 5: Overhead of logging and the running time in seconds of the original execution and the reduced execution with and without tracing.

Bug Program	Logging Overhead			Replay without Tracing			Replay with Tracing				
	Orig-1	Logged	Logged/ Orig-1	Orig-2	R_Thread-2	R_TEI-2	Orig-3	Orig-3/ Orig-2	R_Thread-3	R_TEI-3	SR_TEI-3
mysql-1	14.8	16.8	1.1	16.4	0.1	0.08	3736	227.8	0.8	0.7	0.67
mysql-2	12.3	14.0	1.1	12.6	1.1	0.1	2806	222.6	4.0	0.9	0.9
mysql-3	13.9	15.8	1.1	15.4	2.3	0.09	3270	212.3	468	0.9	0.9
prozilla-1	4.8	13.4	2.8	12.4	0.08	0.05	2664	214.8	0.6	0.5	0.5
prozilla-2	7.2	18.7	2.6	16.5	16.5	16.5	2364	143.3	2364	2364	560
proxyc-1	11.0	19.8	1.8	16.6	0.07	0.07	960	57.8	0.3	0.3	0.3
axel-1	0.15	0.16	1.1	0.14	0.02	0.02	3.2	22.8	0.3	0.3	0.26

seconds. For *axel*, the bug that we consider happens during the initialization phase. Hence, we could not make this program as long as other programs. Note that even though checkpointing is supported in our system, given the lengths of executions, multiple checkpoints were not created. Now, let us discuss the different experiments we have conducted.

Space Overhead. Table 4 shows the size of the basic block (control flow) and dependence traces in terms of the number of basic blocks and dependences for the various executions we have considered. We have measured the basic block and dependence trace sizes for four different executions of the same program. First, we measured the trace sizes of the original run shown under the heading *Orig* in Table 4. Then, we measured the trace sizes of the programs by replaying only the *relevant threads* which is shown under the heading R_Thread. The data under the heading R_TEI corresponds to the trace sizes by replaying only the *relevant thread execution intervals* in the program. For R_Thread (R_TEI), the basic block traces are smaller than the original by factors ranging from 1 (1) to 50442 (58460) and the dependence traces are smaller by factors ranging from 1 (1) to 49300 (59000).

We also performed an additional experiment of measuring the trace sizes by using *selective tracing* (SR_TEI), that is, we replay all the relevant TEIs but do not necessarily trace all of them. Selective tracing of TEIs is performed as follows. For programs with a memory bug, that causes a Segmentation Fault, the bug manifests itself from the root cause to the crash point through a series of memory dependences in the program. Hence, if the faulty interval TEI_i is not memory dependent on another interval TEI_j, then the trace of TEI_j does not contain any useful information about the crash. However, TEI_j may still have to be replayed since TEI_i may be dependent on it due to event dependences. Since we already have all dependences between the various TEIs, we use this information to decide which TEIs to trace. Then, we use our dynamic tracing infrastructure to selectively turn on tracing for the appropriate TEIs. With selective tracing, the reduced basic block traces are smaller than the original by a factor of 478 to 490000. The corresponding reduction factors for dependence traces range from 380 to 470000. Note that this huge reduction in trace sizes comes from both execution reduction and selective tracing. For *prozilla-2*, selective tracing is the only single contributing factor.

Time Overhead. Table 5 gives the data on the runtime performance for the various executions we have considered. First, we measured the *logging overhead* on the original execution. In Table 5, under the heading of Logging Overhead, we give the execution times of the program run without logging (*Orig*-1) and with logging (*Logged*). The ratio of the two in column *Logged*/*Orig*-1 shows that the program execution slows down by a factor ranging from 1.1 to 2.8. The logging overhead is small for *mysql* and *axel* and slightly higher for *prozilla*, *proxyc*, and *proftp*. The reason for the slightly increased overhead for some programs is because their long-running execution involves downloading large files from a website. Jockey makes a separate copy of the contents of the downloaded file and this increases the overhead. However, the overhead is still reasonable and is acceptable to have logging turned on during normal execution.

We then measured the execution times of the programs during replay from corresponding logs both without and with tracing. In each of these two cases we made three measurements: the execution time to replay the *entire execution* (*Orig*-2/3); the execution time to replay the execution of only the *relevant threads* (R_Thread-2/3); and the execution time to replay the execution of only the *relevant*

Table 6: Replay Log Sizes of original and shortened runs, (M - million).

Program	Number of events in replay log			
	Orig.	R_Thread	R_TEI	Orig./ R_TEI
mysql-1	4801	281	236	20.3
mysql-2	3749	489	365	10.3
mysql-3	5453	902	332	16.4
prozilla-1	7.2 M	621	73	98000
prozilla-2	10.8 M	10.8 M	10.8 M	1
proxyc-1	32.8 M	798	798	41000
axel-1	1695	954	954	1.8

thread execution intervals (R_TEI-2/3). In case of replay with tracing, we made an additional measurement that takes advantage of selective tracing (SR_TEI-3).

Let us consider the performance of replaying the original and reduced executions *without tracing* turned on. Excluding *prozilla-2*, while the original execution time $Orig$-2 that includes all threads ranges from 0.14 to 16.6 seconds, the execution time R_Thread-2 which excludes irrelevant threads ranges from 0.02 to only 2.3 seconds. Then, if irrelevant TEIs are removed, the execution time is further reduced to R_TEI-2 which ranges from 0.02 to only 0.1 seconds. With the exception of *prozilla-2*, all the buggy programs have a significant reduction in their execution times.

Next we consider the performance of the various executions *with tracing* turned on. The overhead of tracing given by column $Orig$-3/$Orig$-2 is as high as 228 which is the factor by which the execution slows down. For *mysql* our data shows that tracing can cause a significant slowdown in performance which cannot be tolerated even during debugging. However, after execution reduction this overhead is greatly reduced. Excluding *prozilla-2*, while the original execution time $Orig$-3 that includes all threads ranges from 3.2 to 3736 seconds, the execution time R_Thread-3 which excludes irrelevant threads ranges from 0.3 to 468 seconds. Then, if irrelevant TEIs are removed, the execution time R_TEI-3 is further reduced and it ranges from 0.3 to only 0.9 seconds (excluding *prozilla-2*). With selective tracing the execution time SR_TEI-3 for *prozilla-2* is greatly reduced, that is, from 2364 to 560 seconds. Thus, the combination of removing irrelevant threads, removing irrelevant TEIs, and performing selective tracing proves effective for all programs.

Table 6 gives the number of events in the original and reduced replay logs for the original and shortened executions. The final reduced log is smaller than the original by factors ranging from 1 to 41000 which translates into smaller execution times as already observed and hence smaller trace sizes.

6. RELATED WORK

There have been many works that have explored the technologies of checkpointing/logging for replay [25, 15, 22] and lower level tracing [29, 8] to collect dynamic information useful in debugging. However, these technologies have been explored separately. In this paper, we show benefits of integrating them into a single framework.

The prior work that is closest to ours is the *Execution Fast Forwarding* (EFF) [30] system. The EFF system also performs a form of Execution Reduction by integrating checkpointing with fine-grained tracing. It is based on the idea that often a fault is triggered by a certain input. By filtering the inputs to find the fault triggering input, the fault can be reproduced. Tracing can then be applied to the smaller program run corresponding to the triggering input. However, the *Execution Reduction* (ER) system described in this paper is much more general than the EFF system. In particular, the advantages of ER over EFF include the following:

- The ER system is designed to handle multithreaded applications while the EFF system was designed for single threaded applications. One of the key contributions of the ER system is the dynamic algorithm that we provide for efficiently identifying the interthread dependences. The EFF system does not address this issue as it does not consider multithreaded applications.
- In the EFF system, the execution reduction is achieved by exploiting information collected using static analysis. The disadvantage of using static analysis is that it is conservative and hence dynamic opportunities for achieving execution reduction cannot be exploited. In particular, if static dependences do not manifest themselves at runtime, this information can be exploited for execution reduction in the ER system but not in the EFF system.
- The ER system does not necessarily trace the entire execution that is replayed. In contrast, the EFF system traces the entire execution that is replayed. Therefore, in the ER system, the reduction in tracing is not limited by the amount of execution reduction achieved.
- Finally, input filtering that is used as the basis of execution reduction in EFF is the special case of execution reduction achieved by the ER system. ER system can handle a variety of situations, including those where input filtering is applicable.

In summary, the attractive features of the ER system are that it is more general and more effective than the EFF system.

There has been some recent work that propose designs of specialized hardware to limit the overhead of checkpointing/logging [13, 27]. These systems aim to minimize the overhead incurred while logging an execution for replay. This work is orthogonal to ours as these systems could be used to improve the performance of our logging phase. However, the advantage of the proposed ER system is that it relies entirely on software techniques and is therefore applicable to systems being used today. There has also been work on how to record shared memory dependences efficiently to replay multithreaded programs when run on a multiprocessor [28, 14]. When run on a multiprocessor, recording the schedules alone is not enough because multiple threads can execute simultaneously. However, only the shared memory dependences (RAW, WAR and WAW) of concurrently executing threads need to be recorded. In our work, to find interthread dependences, we need to track all shared memory dependences but only of the type RAW.

7. CONCLUSIONS

In this paper, we described the *execution reduction system* that can effectively combine checkpointing and tracing in

order to debug long-running multithreaded programs. Our system uses dynamic techniques for eliminating the execution of irrelevant threads and irrelevant thread execution intervals from the final replay phase that collects traces. Further, it also eliminates unnecessary tracing during the replaying of relevant threads and thread execution intervals. The combined effect of the above approach is that the tracing overhead and the amount of trace data collected is greatly reduced. Most importantly, to make the above scheme work, we developed a three stage scheme for identifying dynamic shared memory dependences between executing threads that is both space and time efficient. Our experiments demonstrate the effectiveness of the proposed techniques.

8. REFERENCES

[1] www.mysql.org

[2] http://bugs.mysql.com/bug.php?id=110

[3] http://bugs.mysql.com/bug.php?id=169

[4] http://bugs.mysql.com – Change Log

[5] http://www.securityfocus.com/bid/12635

[6] http://prozilla.genesys.ro/?p=news

[7] http://www.securityfocus.com/bid/13059

[8] S. Bhansali, W-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," *Virtual Execution Environments Conference*, Ottawa, Canada, June 2006.

[9] N. Gupta, H. He, X. Zhang, and R.Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263-272, Long Beach, California, Nov. 2005.

[10] T. Gyimothy, A. Beszedes, I. Forgacs, "An efficient relevant slicing method for debugging," *7th European Software Engineering Conference/ 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, 1999.

[11] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, Vol. 29, No. 3, pages 155-163, 1988.

[12] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: a benchmark for evaluating bug detection tools", *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[13] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Shared Memory Dependences for Deterministic Replay Debugging," *Thirty Second International Symposium on Computer Architecture*, Wisconsin, USA, June 2005.

[14] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependences using Strata," *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2006.

[15] R.H.B. Netzer and M.H. Weaver, "Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, USA, pages 313-325, June 1994.

[16] D.Z. Pan and M.A. Linton, "Supporting reverse execution of parallel programs," *ACM workshop on parallel and distributed debugging*, Madison, WI, USA, May 1988.

[17] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies - a safe method to survive software failures", *the 20th ACM Symposium on Operating Systems Principles* Brighton, UK, pages 235-248, Oct. 2005

[18] M.C. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee, "Enhancing Server Availability and Security Through Failure-Oblivious Computing", *the Sixth Symposium on Operating System Design and Implementation* San Francisco, California, pages 303-316, 2004

[19] M. Ronsse, K. De Bosschere, M. Christiaens, J.C. de Kergommeaux, and D. Kranzlmller, "Record/replay for nondeterministic program executions", *Communication of the ACM* 46(9), pages 62-67, 2003

[20] M. Ronsse, K. De Bosschere, and J.C. de Kergommeaux, "Execution replay and debugging", *Fourth Workshop on Automated and Analysis-Driven Debugging*, Munich, Germany, August 2000.

[21] R. Rugina and M.C. Rinard, "Pointer Analysis for Multithreaded Programs," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77-90, Atlanta, May 1999.

[22] Y. Saito, "Jockey: a user-space library for record-replay debugging", *Sixth International Symposium on Automated and Analysis-Driven Debugging*, Monterey, California, September 2005.

[23] A. Salcianu and M.C. Rinard, "Pointer and Escape Analysis for Multithreaded Programs," *8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12-23, Snowbird, Utah June 2001.

[24] J. Seward et al. "Valgrind: A GPL'd system for debugging and profiling x86-linux programs", *http://valgrind.ked.org/*, 2004.

[25] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging", *USENIX Annual Technical Conference*, Boston, MA, USA, June 1994.

[26] L.D. Wittie. "Debugging distributed C programs by real time replay," *ACM workshop on parallel and distributed debugging*, pages 57-67, Madison, WI, USA, May 1988.

[27] M. Xu, R. Bodik, and M. Hill. "A Flight-Data Recorder for enabling Full-System Multiprocessor Deterministic Replay," *Thirtieth International Symposium in Computer Architecture* , San Diego, California, June 2003.

[28] M. Xu, R. Bodik, and M. Hill. "A Regulated Transitive Reduction for Longer Memory Race Recording," *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2006.

[29] X. Zhang and R. Gupta, "Whole Execution Traces," *IEEE/ACM 37th International Symposium on Microarchitecture*, pages 105-116, 2004.

[30] X. Zhang, S. Tallam, and R. Gupta "Dynamic Slicing Long Running Programs through Execution Fast Forwarding," *14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006