Avoiding Program Failures Through Safe Execution Perturbations

Sriraman Tallam
Google Inc.
tmsriram@google.com

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

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Abstract

We present an online framework to capture and recover from program failures and prevent them from occurring in the future through safe execution perturbations. The perturbations are safe as they respect the semantics of the program. We use a checkpointing/logging mechanism to capture a program execution to an event log. If the execution results in a failure, the framework automatically searches for perturbation of the execution by altering the event log and replaying the execution using the altered log to avoid the failure. If found, the perturbation is recorded as a dynamic patch, which is later applied by all future executions of this application to prevent the failure from occurring again. Our experiments show that the proposed framework is very effective in avoiding concurrency faults, heap memory overflow faults, and malicious requests. The entailed overhead for normal execution is very low (2-18%).

1. Introduction

A large number of failures that occur in today's software, including those causing system crashes or producing wrong visible outputs, are due to the execution environment. In a study by Chandra and Chen [9], 56% of failures in Apache server are dependent on the environment. These failures are often fixable by safe execution perturbations such as changing thread scheduling, padding memory allocations, and dropping user requests. These perturbations are safe as they do not change the correct behavior of a software.

We present an online framework to capture and recover from failures caused by execution environment and prevent them from occurring in the future. As these failures could be non-deterministic, we use a checkpointing/logging mechanism to capture the execution in an event log. If the execution results in a failure, the framework tries to fix the faulty program execution through safe perturbation, which is performed by altering the event log and replaying the execution using the altered log. The perturbation, if found, is recorded and later referred to and applied by future executions to prevent the failure or similar failures caused by the same fault from occurring again. The framework has the potential of improving system availability without waiting for the official patches from the developer.

Previously, *checkpointing/logging/replaying* has been used to deterministically replay shared memory programs [12, 22] and also in recovery [14]. Most of these techniques rollback execition to a previous checkpoint and replay without any perturbation. While such schemes could avoid some non-deterministic bugs, it cannot recover from deterministic ones. Also, they cannot avoid similar failures from happening in the future. In our framework, we use the checkpointing/logging scheme for two reasons. First, we use it to capture the faulty execution to allow deterministic replay, even in the case of non-deterministic failures. Second, we try to avoid failures through safely perturbing executions by changing the replay log. *Rx* [20] is a system designed to help applications recover from failures due to the environment, by removing the "allergen" that caused the bug to manifest. When a failure occurs, the Rx system rolls back the application to a recent checkpoint and executes it under a modified environment. Repeated environment modifications and re-executions are done until the failure is avoided or a time threshold is passed. If the failure is avoided, the execution is resumed. However, since Rx re-executes the program from a checkpoint with changes applied and *without following any log file*, the perturbed execution might deviate from the original execution and incur wrong decisions on whether the failure has been successfully avoided. To apply changes, various aspects of the system environment, such as the OS scheduler, need to be modified. Furthermore, lacking a *script* that dictates how the system should behave in certain situations entails repeated searches for the same fix for repeated failures. In contrast, our work relies on logging. The perturbed execution is harnessed by a largely unaltered log file. The framework resides completely in the user space without the need of changing the OS because perturbations are conducted through changing the log file instead of the environment. Finally, the fix of a failure is easily applied to future executions through the log file.

In our system, each application that runs goes through three main phases: *Logging Phase*; *Failure Avoidance Phase*; and *Prevention-Logging Phase*. Figure 1 shows the

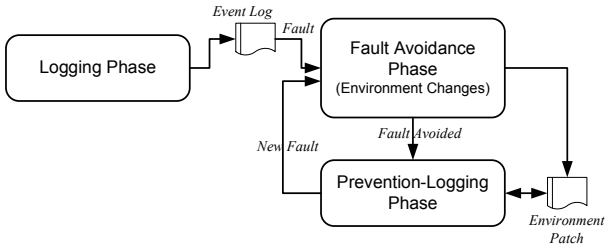

Figure 1. The various phases that an application goes through in our system.

various phases that an application has to go through in our system. The *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, i.e., the *event log*. The set of logged events can be used to exactly replay the execution when necessary, like when a faulty execution is encountered. Once a failure is detected at any point during the execution or at the end, the application is taken to the *Failure Avoidance Phase*. In this phase, the application is analyzed to correct the failure. If the application was a long-running program like a server, then the clients experience non-availability of the application until the failure is corrected and the application is moved out of this phase. In this phase, the framework inspects the event log of the faulty execution and searches for corrective perturbations, which results in altering the execution environment while ensuring safety. For example, to avoid atomicity violation errors, the system changes the order in which threads are scheduled in the new execution by shuffling the threads in the event log. If the failure does disappear after a certain number of tries, the corresponding perturbation is recorded in a *Environmental Patch* (EP), which is indeed in the form of a log file. Such EP files are enforced at runtime by the framework in the *Prevention-Logging Phase* to avoid future occurrences of the same failure or similar failures of the same fault. Note that a fault might result in multiple failures (e.g., a atomicity violation fault may lead to wrong output at different places). In fact, normal programs execution always happens in the logging phase, and also the prevention-logging phase if the EP file is not empty.

We consider three different types of faults: *atomicity violation* faults, avoided by changing the scheduling decisions; *heap buffer overflow* faults, avoided by padding memory requests; and *malformed user request* faults, avoided by dropping the request. These faults were chosen as they are the most common types of environmental faults [20, 9]. We have applied our system to a set of bugs that belong to one of the three types and found that our system can avoid failures caused by these faults in all the cases and the overhead is very low. The key contributions of this paper are as follows:

- We present a scheme that can capture and avoid environment bugs by using a checkpointing/logging system. The presence of logging enables the technique to completely reside in the user space and have less complexity. It also harnesses the perturbed execution and focuses the changes on the faulty region.
- Our scheme also prevents the captured failure and some failures of the same fault from occurring again by supplying EPs, without requiring a developer to debug the program. Our system can handle three different types of faults, namely, atomicity violation, heap buffer overflow, and malformed user request.
- We have tested our scheme on a set of real bugs. The results show it to be effective and efficient.

2. Motivating Example

MySql ver. 4.0.12 has an atomicity violation bug [4] 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 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.

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 2 shows the events recorded in the original event log. T_1, T_2, T_3, and T_4 refer to four unique threads that are created during the execution. 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 2 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 issues a "flush log" command that closes the old mysql log and opens a new mysql log file.
- Thread T_4 is created by T_1 to handle another user. This user does an insertion operation into table 'b'.

Figure 3 shows the code that is executed by Threads T_3 and T_4. T_3 is interrupted at the point just after it closes the *binlog*, corresponding to TEI 7 in the event log. At TEI 8, thread T_4 performs the insert operation but does not find any log open and hence does not record it. At TEI 9, a new

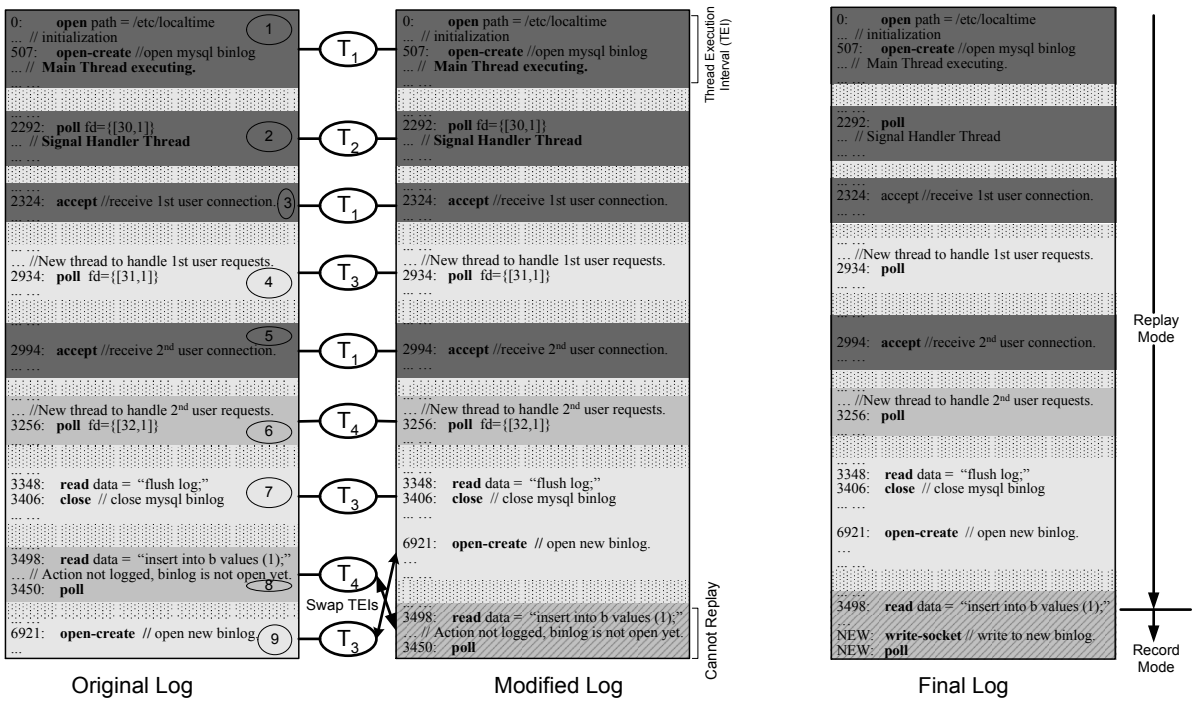

Figure 2. Motivation - MySql Atomicity Violation Error. The figure shows the original log corresponding to the failure and the modified log where the failure is avoided by switching the thread schedules. The final log where the failure (or similar failures of the same fault) has been avoided is also shown. The threads (T_1,...,T_4) are shown and the TEIs(1,...,9) are marked.

binlog is opened but the insert operation is not found to be recorded in any of the logs. Hence, a failure is discovered and the program is taken to the next phase of the framework. The execution could have proceeded much further before this failure is actually detected, like when an administrator runs sanity checks. However, since the event log is present, the execution can be reproduced and the exact point at which the failure occurred could be tracked. Notice that this bug is non-deterministic as the scheduling decisions could be different in another execution instance. Also notice that the log captures the failure successfully.

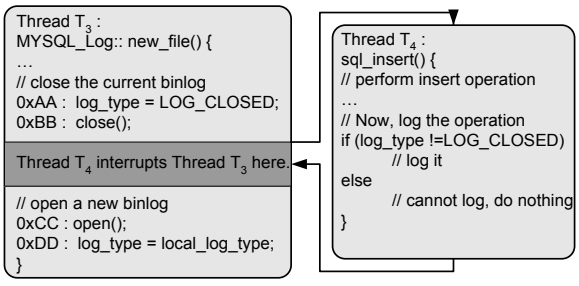

Figure 3. Source code of the mysql atomicity violation fault.

Failure Avoidance Phase. In this phase, the system tries to avoid the failure by safely perturbing the original execution. The faulty execution is first replayed. An oracle is provided by the user to test if the failure occurs by testing whether an insert operation is present in a TEI. TEI 8 was identified as being faulty since the insert operation is done at this point but is not recorded in the *binlog*. The system tries to perturb the original execution by canceling some of the scheduling actions. We note that TEI 8 corresponding to thread T_4 interrupted the execution of thread T_3. Therefore, canceling the scheduling at the boundary of TEIs 8 and 9 results in swapping TEIs 8 and 9 in the event log. Doing so, the interruption disappears. The modified event log is shown in Figure 2. Now, replaying the execution with the modified log results in the insert operation being present in the new *binlog*.

While replaying from the event log, executing the "insert operation" corresponding to TEI 9 in the original log causes the execution path to change. This is clear from Figure 3, where, originally, the condition in the *if* statement evaluated to *false*, it now evaluates to *true*. Hence, the events necessary to replay this is not present in the log of the original execution. So, when this happens, we switch the execution mode from replay to normal execution (and recording). The final log shows the set of captured events for the correct ex-

ecution. Notice that there is an entry now in the final event log corresponding to logging the insert operation.

Now that the failure has been avoided, let us show how the failure or similar failures of the same fault are prevented permanently in the future. From the log, the system identifies that thread T_3 was interrupted by T_4 just when it was performing the *close()* event of the old bin log, whose *PC* value is $0xBB$ as shown in Figure 3. A safe perturbation is to avoid scheduling at this code location which has the effect of enlarging the atomic region of the code. Hence, an entry of the form "$< 0xBB >$: Don't Schedule out" is added to the *Environment Patch* (EP) file. We will show how this helps in avoiding future failures. Now, the server is ready to move to the next phase and start servicing requests normally as the failure has been avoided.

In Rx [20], logging is absent and changes are enforced by changing the OS, which is more complex. A mechanism to enforce the fix in the future is also lacking.

Prevention-Logging Phase. In this phase, the application runs normally with logging turned on just like in the logging phase. When each event is being logged, control is transferred from the application to the logging system. At this point, the EP file is checked to see if the PC of the currently executing event corresponds to a faulty region. For instance, if some thread T_i is executing the piece of code corresponding to PC $0xBB$, the logging system detects that it is a potentially faulty region by looking up the environment patch and also sees that no scheduling must happen at this point. Hence, the priority of the executing thread is raised before the application gains further control. Now, this thread continues executing past this event without being scheduled out and the failure is prevented from happening. The priority of the thread is reset after a predetermined number of events have been executed. Since the logging infrastructure is active, any new failures can still be captured in the log. When a new failure occurs, the application moves back to the failure avoidance phase.

3. Faults and the Corresponding Perturbations

We identify three types of faults which are described in detail in the section. We discuss how we discover the point at which the environment changes must be made to avoid failures. We also show how we use the information we have gathered to prevent future occurrences of the fault in each case. Note that in order to use the methods, classification of observed failures must be carried out. Our system first gathers information about the application – is it multithreaded, should it satisfy certain invariants, and how does it terminate when it is successful. Now, if the execution results in a failure, the system uses the information gathered to classify the failure and apply the appropriate avoidance technique. If a program is multithreaded, all three techniques will be tried one after another till one technique succeeds. Specifically, if an invariant violation is observed, rescheduling and dropping user requests will be tried by our system in that order. In the case of single threaded programs, the rescheduling does not apply and hence the system will try to avoid the failure by dropping requests. In the case of abnormal termination, the system will try to avoid the failure by using buffer overflow techniques first and if necessary follow that by dropping user requests. In this work we consider the scenario in which applications, single or multithreaded, are executed on a uniprocessor.

3.1. Handling Synchronization Faults

To avoid failures due to synchronization errors among threads, like the example in Figure 2, threads that are involved in the failure are first detected. A synchronization error occurs because the execution of a thread, which we call *interuptee*, is interrupted by another thread, which we call *interrupter*, while atomicity is expected. For instance, in Figure 2, thread T_3 is the interuptee and T_4 the interrupter. After the interuptee thread is descheduled, the thread that is executed next is the interrupter.

Now let us discuss the conditions which must be satisfied for a thread to be considered interrupted. Event boundaries at which thread scheduling decisions take place could be synchronous or asynchronous. A thread is interrupted if and only if it is scheduled out after it executed a synchronous event. For example, if a thread after performing a file read event (synchronous) is scheduled out, it is considered interrupted, whereas, a thread which was scheduled out when executing a polling event (asynchronous) is not considered interrupted. The latter case is because the asynchronous event can block the thread for an arbitrarily long interval of time, depending on when the polling is successful, and hence a different thread must be scheduled if execution of the application must proceed. Note that all interruptions do not lead to synchronization errors but we use this as the basis to decide if a synchronization error could have taken place. In Figure 2, thread T_4 did interrupt thread T_3 at TEI 8 as the event at which T_3 was scheduled out corresponded to closing a file, which is synchronous. Also at TEI 5 in Figure 2, thread T_1 did not interrupt T_3 as polling is an asynchronous event.

In the failure avoidance phase, once the TEI where the failure occurred is identified, the system starts searching for a safe perturbation by eliminating interruptions. The strategy is to search backward from the failure residence TEI and let a interruptee execute further by combining it with the next TEI from the same thread. The process is iterative until the synchronization error is avoided or all combinations are exhausted. Again, in the example in Figure 2, the failure occured in TEI 8, which correspond to thread T_4. Extending TEI 7 by combining it with TEI 9 avoids the failure. Note that if we change the scheduler in such a way that all threads execute without interruptions until blocked by an

asynchronous event, the error would go away. Doing so is however not desirable in terms of performance as threads stuck in expensive I/O operations are not scheduled out.

3.2. Handling Heap Buffer Overflow Faults

Upon the occurrence of system crashes that are potentially caused by a heap buffer overflow fault, the execution enters the failure avoidance phase, in which the heap buffer that has been overflowed is detected. In order to do so, a dynamic memory checker is used, which is built upon the *valgrind* instrumentation engine. A *hash table* is used to maintain for each virtual memory address the EIP of the instruction that performed a memory allocation. The program is instrumented at each heap memory allocation instruction that allocates a range of addresses to update the corresponding entries in the hash table with its EIP. The program is then replayed from the event log. During execution, for every load and store to a heap address, the memory checker checks if the accessed address has a corresponding hash table entry. If an entry is not present, an unallocated address is touched. Now, neighboring addresses are searched to see if they have an entry in the hash. If so, this, very likely, corresponds to the EIP of the instruction that allocated the heap buffer which has overflowed.

Once the EIP of the instruction that allocated the heap buffer, denoted as EIP_{mem}, is obtained. The system enters the failure avoidance phase. In this phase, the program is replayed again but with a safe perturbation, which is to pad the memory returned to the overflowed heap buffer by doubling it. If the failure is avoided, an entry will be added to the EP file with the form : "$< EIP_{mem}$: Double Memory>". Now, the application enters the final phase and all future executions avoid failures caused by this fault permanently as follows. During a memory allocation call, the EIP of the instruction is checked to see if it matches an entry in the environment patch. If so, the memory to be allocated is padded by doubling it. The decision to double the memory to be padded is a heuristic based on the faults we looked at.

3.3. Handling Bad User Request Faults

Faults belonging to this category could be malicious user requests that are intended to expose a bug in a server and not do anything useful otherwise. It could also be a set of user requests that are malformed. These faults usually end up crashing the server by overflowing a stack buffer or even a heap buffer. Our strategy to avoid failures caused by these faults is to ignore such requests. This is the last resort as dropping requests that are not malicious is a form of *denial of request*. However, it is still better than starving all the users by bringing down the server. Before dropping a request, the system checks if the failure can be avoided by padding a overflowed heap buffer or modifying thread schedules. If they all fail, the system starts to drop requests one by one in the backward order. If the failure can be avoided, the EIP of the instruction where this request was accepted, EIP_{read}, along with the user request, req, are recorded in the environment patch file as : "$< EIP_{read}, req$: Drop Request>". In the prevention-logging phase, when the same request turns up at this EIP, it is not serviced and the failure is averted.

4. System Description

In this section, we describe the implementation of our system that incorporates checkpointing/logging and a dynamic instrumentation capability. We have used this system to avoid repeated occurrences of environmental faults in different applications.

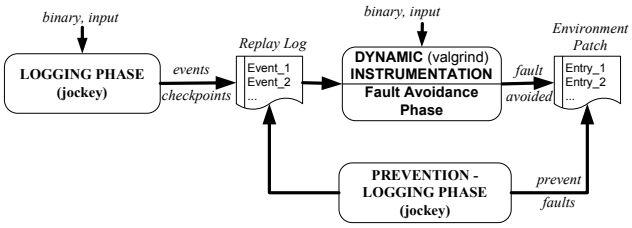

Figure 4. Implementation of our system showing each step of the framework

Logging and Checkpointing Infrastructure. We have used the *jockey* user level library [23] to perform checkpointing and logging for replay. Jockey works on most single and multiple threaded programs. We have modified jockey to pass the *EIP* of each system call and *malloc* instruction in the application as an argument to the jockey system call and *malloc* handler, respectively. This information will be used by jockey in the prevention-logging phase to determine if an entry in the EP is applicable.

Dynamic Instrumentation Engine. The dynamic instrumentation tool is used in the failure avoidance phase. It is used to instrument the application and control its execution so that the system can detect when a failure happens. For synchronization faults, the user has to provide a small oracle function to decide if the output is correct – such as a particular entry needs to present in the *binlog*. This component is built upon the *valgrind* [18] system. It is capable of replaying a program execution from the log file generated by jockey. It also detects illegal heap memory accesses.

5. Case Studies

5.1. Atomicity Violation Fault in mysql

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 below.

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

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 hence, this inconsistency occurs. Figure 5 shows the event

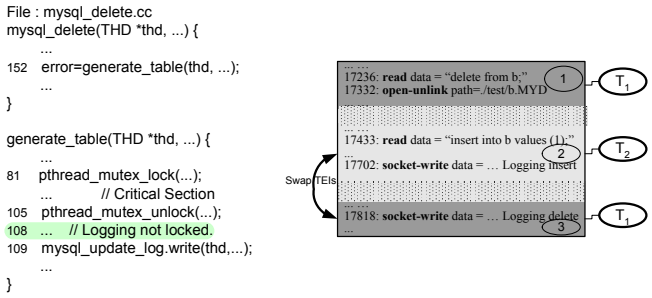

Figure 5. Mysql atomicity violation fault.

log corresponding to the faulty execution. When we replay the program using the log, we can easily detect that TEIs 1 and 2 are directly involved in the failure as the delete and insert operations take place during these points. We can also detect that TEI 2 interfered with the execution of thread T_1. We now extend the execution of thread T_1 by swapping TEIs 2 and 3 and the failure is avoided. We also note the PC at line 108, which is $0x81023AC$, in the environment patch with the command not to schedule at this point. The patch is hence "$< 0x81023AC$: Don't schedule>". In the prevention-logging phase, when the execution reaches this point, the thread's priority is raised so that it does not get descheduled. After applying the patch, we run the server and perform the same sequence of operations to ensure that failures caused by the fault are indeed prevented.

5.2. Heap Buffer Overflow Fault in mutt

Mutt [8] is a text based mail user agent (MUA) for Unix based Operating Systems. It has many features including customizability, POP3 and IMAP support, and ability to handle multiple mailbox formats. According to the bug report[5], mutt version 1.4 has a known memory bug which is as follows. The Mutt Mail User Agent (MUA) has support for accessing remote mailboxes through the IMAP protocol. When mutt has to convert the name of the folder from its internal UTF-8 representation to UTF-7 it calls the function *utf8_to_utf7* in module *imap/utf7.c*. When this function does the conversion, it miscalculates the length of the output string, line number 152 in Figure 6. To form a faulty execution, we execute mutt for some time and then supply a UTF-8 folder name that contains some special characters. The heap buffer is overflowed and a segmentation fault is flagged. The jockey event log captures all the events necessary to replay the failure. We then take the application to the failure avoidance phase.

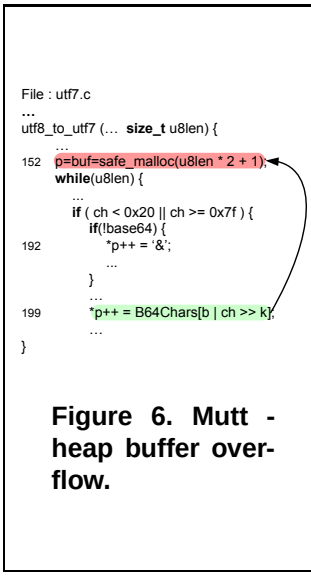

Figure 6. Mutt - heap buffer overflow.

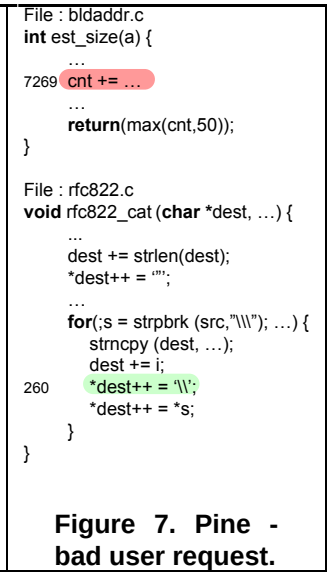

Figure 7. Pine - bad user request.

We detect the heap buffer overflow, line number 199 in Figure 6, using valgrind. We also detect the allocation point for this heap buffer, line number 152. We capture the PC of the malloc call at this point, $0x80A9FBA$. When we replay the program doubling the memory allocated at this point, the failure goes away. We record this in the environment patch with the entry : "$< 0x80A9FBA$: Pad Allocation>" and now continue the program execution with the failure avoided. We run the application for some time and again present the failure-inducing request. The jockey's malloc wrapper successfully makes the environment change and prevents the fault from happening again.

5.3. Bad User Request Fault in pine

According to [6], *pine* ver.4.44 has a bug that when triggered can overflow a heap buffer causing a crash. This can occur when pine processes the "From" field of email headers. Certain special characters in the header can cause the bug. Figure 7 shows the source code where the bug is present. The heap buffer *dest* overflows in line 260 in function *rfc822_cat()* as the amount of memory allocated to it is miscalculated in line 7269 in function *est_size()*. After capturing the event log corresponding to this fault, we first try to avoid the failure by padding memory. We track the heap buffer that was overflown and double the memory at this point but the bug does not disappear. Hence, we detect the request that caused the bug to occur and observe that the request is unusual as it is full of special characters in it. Hence, we decide to drop such requests and add an entry to the environment patch with the contents, "$<0x3A976422$,

Table 1. Overheads involved in each of the three phases - logging, avoidance and prevention-logging.

Bug	Logging Phase			Failure Avoidance Phase					Prevention-Logging Phase		
	Orig. (secs.)	Logged (secs.)	Logged / Orig.	Trials	Valgrind (secs.)	Jockey (secs.)	Environment Change	Potential Trials	Logged (secs.)	Prevention (secs.)	Prev. / Logged
mysql-1	15.5	15.9	1.03	1	116	15.8	Scheduler	186	16.0	16.1	1.01
mysql-2	7.8	8.0	1.03	1	58	8.0	Scheduler	36	8.0	8.5	1.06
mysql-3	7.2	7.4	1.04	1	59	7.3	Scheduler	138	8.7	8.9	1.02
mysql-4	7.9	8.1	1.02	2	454	15.2	Ignore Req.	284	8.7	9.0	1.03
pine-1	6.1	6.8	1.11	2	314	13.2	Ignore Req.	55	8.1	8.4	1.04
pine-2	4.3	4.9	1.14	2	262	9.0	Ignore Req.	2624	6.1	6.2	1.02
mutt	6.7	7.9	1.18	1	197	7.7	Pad Mem.	11653	9.0	9.2	1.02
bc-1	6.5	7.4	1.14	1	285	7.4	Pad Mem.	6555	10.1	10.1	1.0
bc-2	4.3	4.5	1.05	1	190	6.2	Pad Mem.	533	6.2	6.3	1.02

pattern-string : Drop>", where the pattern string is the string of special characters that caused the failure.

6. Experiments

Table 2 shows the list of buggy versions of programs that we have used to evaluate our system. Each of the bugs belongs to one of the three possible types of environment faults we have looked at. We took the buggy version of each program and created an execution that runs for some time, between 4 and 15 seconds, and then introduced the failure. For example, in *mysqld*, we created a few clients and processed a set of standard requests from each client and then triggered the failure by issuing the failure inducing request. Since the executions were not too long, checkpointing was not triggered. After patching the failure by applying the appropriate environment change, we then run the application again for some time and try to introduce the failure as before. We ensure that the failure is indeed avoided by continuing execution beyond the point for a couple of seconds before terminating it. Now, let us describe the various experiments we have conducted.

Table 2. Benchmarks and the bugs

Program	Description	LOC	Description of bugs used
mysqld	Database (ver. 4.0.12) (ver. 3.23.56) (ver. 4.00)	508 K	a) Atomicity bug[4] (mysql-1) b) Atomicity bug[2] (mysql-2) c) Atomicity bug [3](mysql-3) d) Bad Req. bug [1] (mysql-4)
pine	Mail client (ver. 4.44)	212 K	a) Bad Req. bug [6](pine-1) b) Bad Req. bug [7](pine-2)
mutt	Mail client (ver. 1.4)	454 K	a) Heap Overflow [5](mutt-1)
bc	Calculator (ver. 1.06)	14 K	a) Heap Overflow [15] (bc-1) b) Heap Overflow [15] (bc-2)

Logging Phase. In Table 1, under Logging Phase, we present the running time of the application without logging (Original) and with logging (Logged). The overhead of logging, shown under *Logged/Orig.*, is between 2% and 18% and this shows that the logging mechanism is lightweight enough to be run along with the application at all times.

Failure Avoidance Phase. Under the *Failure Avoidance Phase* in Table 1, the number of tries to avoid each bug, under the column *Trials*, is also shown where each try corresponds to a different environmental change. All failures that were triggered by malformed requests needed more than one trial as we first checked if we could fix the failure by padding memory or changing thread schedules. We failed, and hence dropped the request. The column *jockey* shows the total time spent to replay the program using jockey, with the environment changed, to detect if the failure was avoided. The data shows that the jockey time for one trial is almost equal to the original time. The column *valgrind* shows the time taken to replay the program in *valgrind* to detect the regions corresponding to the failure and the time taken to perform analysis, like detecting the allocation point given a heap overflow. This incurs a slowdown of 7x-44x per trial. Note that this cost is incurred only in this phase due to the expensive analysis that is performed using *valgrind*. This overhead will not be present in the prevention-logging phase when the application runs normally. The column *Environment Change* shows the patch that avoided the bug and used to prevent it from occurring again. The column *Potential Trials* shows potentially the number of changes that have to be tried by an ad-hoc scheme. For instance, for *mysql-1*, 186 different scheduling decisions were made during the execution. In the case of *mutt*, memory was allocated at 11653 different points in the execution. Hence, an ad-hoc scheme that does not use any technique to find the region of the error has to potentially try all 186 scheduling points before the atomicity violation can be nailed. This number of possible trials could be overwhelming for such a system. Since our system focuses the failure to a region, only a few trials are needed.

Prevention-Logging Phase. Finally, we present the overheads, of performing logging and preventing failures, that are incurred in this phase. Table 1 shows these costs under Prevention phase. The column *Logged* shows the overhead of logging the execution in this phase with the bug fixed in the source code and the prevention mechanism turned off. The column *Prevention* shows the time taken to perform logging and prevention on the application with the bug present. The additional overhead of preventing failures beyond the logging overhead is shown under the column *Prev./Logged*. The overhead, which ranges from 0% to 6%, is low and is due to the fact that we could successfully merge the operation of checking the environment patch with the logging. The combined overhead of the logging with

prevention mechanism is between 2% and 19% and is low enough that it can be run alongside the application always.

7. Related Work

Logging techniques for applications executing on a multiprocessor have been proposed [17, 24] which use hardware support to capture shared memory dependencies of threads executing simultaneously. These techniques are very lightweight and can be used in our system to further reduce the overhead of logging. Our techniques can be applied to the event logs of these systems thereby making our system feasible even for applications running on muticore systems.

Recently, an execution fast forwarding (EFF) system was proposed [25] that uses logging to capture non-deterministic faulty executions, like our system. The EFF system then prunes the event log, to remove portions of execution that did not contribute to the failure, and applies expensive techniques like program tracing to the shortened execution to reduce the cost of debugging. Since our technique produces the log of the faulty execution, the log can be given to the EFF system to debug the program off-line.

A number of dynamic fault detection techniques [10, 11, 13, 19] exist that instrument the program to check for illegal memory accesses, deadlocks and data races at run-time and also have reasonable overhead. Such techniques can be used in our system to flag a fault when it occurs.

Avio [16] is a technique to detect atomicity violation bugs in programs. The main idea of the technique is to use a number of correct runs, with different interleavings in each run, of the application on the same input and discover atomic regions of the program. Once we have detected the point at which atomicity is possibly violated, we can pass this information to such a system that can use it to detect if an invariant exists.

Failure-Oblivious computing [21] is a technique that bypasses failures in applications by altering the behavior of the application when it detects accesses to unallocated memory. It manufactures values for incorrect reads and ignores illegal writes to let the application continue further execution without crashing. This approach needs modifications to the application and the correctness of the application cannot be guaranteed.

8. Conclusions

In this paper, we have presented a scheme that uses logging and environment patching to capture and avoid environment failures as and when they occur and prevent them from occurring again. We have also shown through case studies that our scheme can be successful against three types of environment faults and we have verified this on nine known bugs in real-world applications. We have also presented data which shows that the overhead of the logging and prevention mechanism is low enough, 2% to 19%, to justify it being run alongside the application at all times.

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

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

[3] http://bugs.mysql.com/bug.php?id=6678.

[4] http://bugs.mysql.com/bug.php?id=791.

[5] http://www.securiteam.com/unixfocus/5fp0t0u9fu.html.

[6] http://www.securityfocus.com/bid/6120.

[7] http://www.xatrix.org/advisory.php?s=7408.

[8] www.mutt.org.

[9] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN* 2000.

[10] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. In *PLDI*, pages 232–244, 2003.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, pages 63–78, 1998.

[12] Y. A. Feldman and H. Schneider. Simulating reactive systems by deduction. *ACM TOSEM*, 2(2):128–175, 1993.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX*, pages 125–138, , 1991.

[14] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *PODC*, pages 171–181, 1988.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS* 2006.

[17] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[19] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, pages 291–302, 2005.

[20] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP* 2005.

[21] 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. In *OSDI*, pages 303–316, 2004.

[22] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared memory applications. In *PLDI* 1996.

[23] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG*, pages 69–76, 2005.

[24] M. Xu, R. Bodík, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–133, 2003.

[25] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE* 2006.