Echo: A deterministic record/replay framework for debugging multithreaded applications

Individual Project Report

Ekaterina Itskova
Imperial College, London

Project Supervisor: Dr Paul Kelly
Second Marker: Dr Dr Philip Leong

June 14, 2006

Abstract

Debugging is a hard task. Debugging multi-threaded applications with their inherit nondeterminism is very hard. Race conditions that arise from nondeterministic application behaviour may only be spotted in one of many runs of an application and thus can be very difficult to reproduce. This project presents Echo – a framework for recording and deterministically replaying execution of single-threaded and multi-threaded applications. Echo is implemented as kernel extension to the Linux operating system. It works by intercepting and logging all the system calls invoked by the traced application and their results. The obtained trace can later be replayed to the application as many times as might be require to find and fix an error in the application. To achieve deterministic replay when multi-threaded applications are recorded and replayed, Echo implements a technique of using hardware performance counters to count the number of instructions executed by a traced application before it is swapped out for another process at a context switch. By intercepting the context switches occurring in the system combined with counting instructions executed by a given process during its quantum, Echo obtains the thread ordering which is during the replay stage is forced onto the application that is being replayed to provide a deterministic replay of an earlier execution.

This report documents the techniques implemented in the Echo framework and presents and discusses the results of experiments performed to test the performance overhead incurred by Echo on application execution.

Acknowledgments

I would like to thank my supervisor, Dr Paul Kelly, for giving me an opportunity to work on this interesting and challenging project and for all the advice, insight and support he gave me while I was working on this project.

I would also like to thank my second marker, Dr Philip Leong, for his enthusiasm and support for this project.

A very special thank you is to my Mum and Dad for all the support they gave me over the years. They provided me with this fantastic opportunity of studying here and I very much hope I can make them proud.

I would also like to thank Adam for all the help and support he gave me during my four years of university. I probably wouldn't have even completed my first year if it wasn't for his amazing tutoring skills and just as amazing patience. Thank you for always being there for me.

Contents

Abstract **1**

Acknowledgments **2**

1 Introduction **2**
1.1 Motivations and aims . . . 2
1.2 Approach Overview . . . 3
1.3 Contributions . . . 3
1.4 Structure of this report . . . 4

2 Background **5**
2.1 Deterministic replay . . . 5
2.2 Bugnet (1988) . . . 6
2.3 BugNet (2005) . . . 6
2.4 DejaVu . . . 8
2.5 Jockey . . . 9
2.6 Flashback . . . 11
2.7 JReplay . . . 13
2.8 Performance counters . . . 13
2.8.1 Overview . . . 14
2.8.2 Performance monitoring hardware of Pentium 4 . . . 15
2.9 Summary . . . 17

3 Design Overview **18**
3.1 System architecture . . . 18
3.2 Echo driver . . . 19
3.3 Logger . . . 20
3.4 Re-player . . . 21
3.5 Summary . . . 22

4 System calls **23**
4.1 System calls interception . . . 23
4.1.1 Intercept function . . . 26
4.1.2 Replay result macro . . . 28
4.2 Process tracing . . . 29
4.2.1 Process state structure . . . 29
4.2.2 Initial process identification . . . 31
4.3 Summary . . . 32

5 Implementation of buffers 33
5.1 Types of buffers . . . 33
5.2 Shared buffers . . . 34
5.2.1 mmap method . . . 35
5.3 Double buffering . . . 38
5.4 Buffer race conditions . . . 39
5.5 Summary . . . 41

6 Thread schedule capture 42
6.1 Approach overview . . . 42
6.2 Context switch interception . . . 43
6.3 Performance counters . . . 44
6.3.1 Configuring the counters . . . 44
6.3.2 Interrupt handling . . . 46
6.4 Recording instructions executed . . . 47
6.4.1 Process creation, execution and termination . . . 47
6.4.2 Recording running duration . . . 48
6.5 Replaying instructions executed . . . 49
6.5.1 Handling counter overflow interrupts . . . 52
6.5.2 Single step interrupts . . . 53
6.5.3 Blocking and resuming of threads . . . 53
6.6 Summary . . . 55

7 Debugging tools 57
7.1 Motivation and overview . . . 57
7.2 Implementation . . . 58
7.2.1 Stop system call . . . 58
7.2.2 Stopping and resuming performance counters . . . 58
7.2.3 Replay debug macro . . . 61
7.3 Limitations . . . 61
7.4 Summary . . . 62

8 Testing and Evaluation 63
8.1 Testing . . . 63
8.2 Evaluation . . . 67
8.2.1 CPU bounded applications . . . 67
8.2.2 I/O bounded applications . . . 69
8.3 Limitations . . . 72
8.4 Summary . . . 73

9 Conclusions 74
9.1 Achievements . . . 74
9.2 Future work . . . 75

Bibliography 76

Chapter 1

Introduction

1.1 Motivations and aims

Multi-threaded applications offer various benefits such as more responsive user interface, simpler programming model when the application is trying to do more than one task at a time and higher throughput since CPU time is not wasted while some applications block and wait for resource availability. These and other reasons are what made multi-threaded application more and more popular in recent years. However, with benefits also come drawbacks and one of the big ones for multi-threaded applications is that they are notoriously difficult to debug.

Debugging is a hard task. Some errors that occur in the production environment can be hard to replicate even for a single-threaded applications, because reproducing them requires knowing exactly what the user has done that has lead to an application failure and this information is not always available. Debugging multi-threaded applications with their inherit nondeterminism is very hard. Race conditions that arise from nondeterministic application behaviour may only be spotted in one of many runs of an application and thus can be very difficult to reproduce. It is therefore very important to have tools that allow efficient debugging of these complex systems.

This project is about creating a tool that will allow to record execution of applications running in their production environments and to replay this execution deterministically at a later stage. Such a tool will allow developers to reproduce race conditions that occur in the application on every execution replay and hence it will make debugging multi-threaded applications more efficient. It will also eliminate the need for users to provide feedback on their actions that have lead to an application failure since developers would be able to replay the application execution and observe the failure themselves.

The aim of this project is to develop such a tool and hence the Echo framework was designed to fulfill the following main objectives:

- Provide a mechanism to record execution of single-threaded and multi-threaded applications.
- Provide a mechanism to replay execution of single-threaded and multi-threaded applications.
- Provide a mechanism for inserting debugging statements into the application after it has been recorded and replay the application without having to re-record its execution.

1.2 Approach Overview

The Echo framework works by intercepting system calls and recording their return values and any data returned by them to the user application. It uses hardware performance counters to count the number of instructions executed by each thread in the recorded application before they get swapped out. The values obtained form a thread schedule of the recorded application.

The sequence of logged system calls is replayed to the application at a later stage, recreating application's execution. For multi-threaded applications, a captured thread schedule must also be replayed to achieve deterministic replay. The Echo framework forces the recorded application to stick to the captured schedule by only allowing one thread to run at a time with the rest of the threads blocked until it is their time to run according to the recorded schedule. The Echo framework uses performance counter interrupts to interrupt a running thread and swap it out for the next thread to run.

The Echo framework is implemented as a kernel extension, which allows great flexibility in the amount of control gained over the execution of an application. More importantly, it makes possible to trace the execution of an application without having to instrument it or any of the shared libraries used by an application.

1.3 Contributions

There has been a lot research conducted in the field of the deterministic record/replay execution. Chapter 2 gives an overview of the major projects in this area. However, they all suffer from various limitations ranging from their inability to properly support multi-threaded applications to the fact that they only work for a particular API or on a particular customized architecture. The Echo framework attempts to fill in the gaps where other projects have failed.

The contributions of this project are:

- **A novel framework has been developed that allows to record and deterministically replay execution of an application.** This is the main contribution to the current research in this field. The developed framework has been tested and found to be able to record and correctly replay an execution of a multi-threaded application. A multi-threaded application has been replayed a number of times using a recorded execution data on all the runs and the results of all the replay runs equal to the result obtained from the application during the record stage.
- **An application to be recorded or replayed does not need to be instrumented or be written using particular API.** The Echo framework has been implemented as a kernel extension and hence it does not require instrumentation of a recorded/replayed application or shared libraries in order to record or replay an application. Moreover, the framework does not rely on any particular features of an application and hence it does not require an application to be written in any particular programming language.
- **A mechanism has been developed allowing to insert debugging statements into the application after it has been recorded and replay that application without having to recompile it.** The Echo framework provides support for inserting a small number of debugging

statements into an application after it has been recorded. The application can then be re-copiled and replayed using the data collected during the record stage without repeating the recording stage. The tests performed on the framework show that the replay results are correct after the debugging statements have been inserted.

1.4 Structure of this report

Chapter 2 discusses the state of the art in the field of deterministic record/replay execution. It reviews a number of influential projects in this field, highlights their similarities and differences with the Echo framework. It also covers performance monitoring hardware, which play a very important part in the implementation of the framework.

Chapter 3 presents a high-level overview of the Echo framework. It discusses different components of the framework and how they interact with each other to make up a functioning system.

Chapter 4 explains how the system call interception was implemented in Echo and also how the framework determines which process running in the operating system is being recorded/replayed.

Chapter 5 discusses the logging methods used by the framework. It talks about implementation of buffering execution data, what implementation problems have been encountered and how they have been solved.

Chapter 6 explains how the thread schedule is captured and forced upon an application by the Echo framework. It discusses how the performance counters were used to capture and replay a given thread schedule.

Chapter 7 discusses how the support for inserting debugging statements into a recorded application has been implemented.

Chapter 8 focuses on the evaluation of the framework. It explains how the testing of the framework was conducted an what experiments were made to assess its performance.

Chapter 9 concludes this report by talking about future work and lessons learnt from this experience.

Chapter 2

Background

This chapter investigates the research that has been done on execution record/replay frameworks and the use of performance counters for profiling purposes. It also reviews existing projects relevant to Echo, what can be learnt from these research efforts and how the current state of the art might be improved by Echo.

2.1 Deterministic replay

Deterministic replay is a technique used in debugging non-deterministic applications. Such a technique operates in two stages: record and replay. The purpose of the initial record stage is to capture the history of events in the monitored application and to record all the data required to reproduce this history at a later stage. During the subsequent replay stages, the application is forced to execute according to the captured history.

The main challenge (and an active research topic) with this approach is in determining what and how much information must be recorded to ensure deterministic replay. There are two main approaches to solving this problem: the content-based or data-driven execution and ordering-based or control-driven execution.

The content-based approach suggests that if we record every instruction executed by the monitored application with the correct input to it, then output would be the same and hence we would have deterministic re-execution. However, this method is impractical, as it would generate large amount of logged data required for re-execution.

The ordering-based approach suggests that we do not need to record every instruction to replay execution. We need, however, to record timings and input to the application coming from the outside sources such as I/O channels, program files or other threads. If these inputs are replayed at the same time as they were delivered in the original execution, an equivalent re-execution is obtained. This method benefits from reduced logs, however it suffers from the fact that some inputs cannot be produced using this technique, such as keyboards strokes. Thus, in practice a mix of content-based and ordering-based approaches is used in deterministic replay. [MR03]

There has been a lot of work done on the topic of execution record/replay frameworks. Most of the research has been focusing on ways to intercept and record information required to replay execution at a later stage. The approaches explored range from hardware implementations to user-space libraries.

2.2 Bugnet (1988)

One of the earliest work on record/replay frameworks, **Bugnet** [Wit88] is a UNIX tool designed to debug distributed C programs. It monitored the execution of distributed processes and gave users information about interprocess communication, I/O events and execution traces for all processes in a distributed application. Bugnet took periodic global checkpoints of the monitored application and allowed users to roll back and replay execution from a checkpoint.

The core of Bugnet implementation was the capturing and timestamping of all interprocess communication. In order to capture all messages, Bugnet defined a set of IPC primitives that had to be used by processes to send and receive messages. Once a message was intercepted, Bugnet attached a timestamp to it, so that it can be recreated at the correct time during replay.

Bugnet itself consisted of multiple processes running on a distributed network: a Global Control Module (GCM), Local Control Modules (LCM), a TRACER and a TRAP processes. During the record stage of execution, each TRAP process captures the messages sent and received by its application process, which are timestamped and stored to be used later. Once user decided to replay execution from a particular checkpoint, which are taken by TRACER process, TRAP arranges the history of captured messages to match the timestamp of the selected checkpoint. During replay, individual application process can be re-executing and sending message anew or not re-executing. In case of re-executing processes, TRAP checks all newly sent message with its history and any discrepancy in time or content of the message is presented to the user for debugging purposes. In case of process not re-executing, TRAP uses stored message history to deliver messages to these processes at their original times, thus simulating original execution.

The main limitations of Bugnet is the fact that it is orientated towards a particular type of systems and supports applications written to a particular API, unlike Echo, which is designed to handle generic Linux applications.

2.3 BugNet (2005)

One of the most recent projects in the area of deterministic replay is a tool called **BugNet** [SNC05]. BugNet is similar to Echo in its goals, which is to help developers to reproduce and fix errors in released code by recording information about monitored system during a production run and replaying it later. However, the way this goal is achieved is in BugNet is very different to Echo.

BugNet is based on a Flight Data Recorder (FDR) project [1801], which is a tool that enables deterministic replay for multiprocessor systems. BugNet also focuses on multiprocessor systems and makes use of dedicated hardware buffers to provide architecture support to record enough information to deterministically replay instructions preceding a system failure.

BugNet focuses on catching application level errors and hence it records and deterministically replays instructions executed by applications and libraries used by them, but not the operating system.

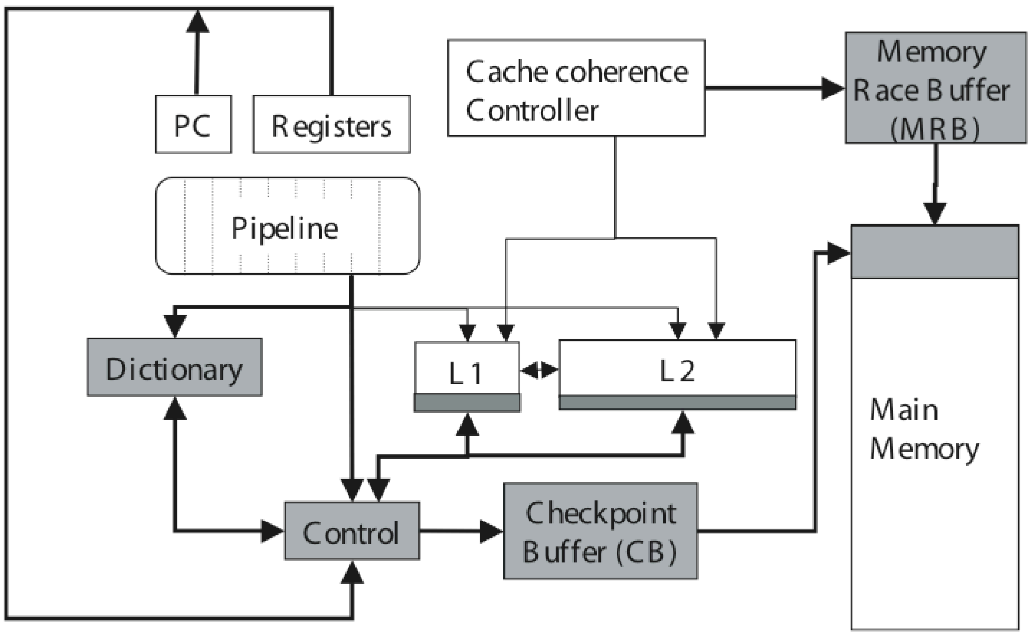


Figure 2.1: BugNet's architecture [SNC05]

BugNet's implementation approach is based around checkpointing. Checkpoints are taken after a specified number of instructions have been executed, at which point the current checkpoint is terminated and a new checkpoint is created. Checkpoints can be terminated by a program failure, at which point program logs will be collected to be available for debugging. Interrupts, system calls and context switches can also terminate checkpoints.

BugNet records the initial register state at the beginning of each checkpoint and then it records the values used by load instructions executed during a checkpoint interval when these instructions are the first access to a particular memory location during that checkpoint. This information is referred to as the First-Load log by the framework and is stored in a memory backend FIFO queue Checkpoint buffer (see Figure 2.1) along with FLL for other checkpoints until an error is encountered and a final log is stored on disk. If memory becomes full before the error is encountered, the logs corresponding to the oldest checkpoint are discarded.

Dictionary is a 64-enty full associative table, which is used as an optimization applied to recording load values. It holds the most frequently loaded values and instead of logging a full 32-bit load value for each load instruction in FLL, a 6-bit index to a dictionary table is stored if the value exists in the dictionary; otherwise a full load value is stored.

In order to use an optimization, which only logs first accesses to the memory location, BugNet associates a first-load bit with every word in L1 and L2 caches. At the start of the checkpoint all those bits are cleared. When a load accesses a word that has not been accessed before and hence has a word bit unset, then this load value is recorded in Checkpoint Buffer and the bit in the caches will be turned on. All subsequent accesses to that word will not be logged.

Threads in a multi-threaded application are replayed independently of other threads in BugNet as it records all data required to replay a particular thread. However, BugNet provides support for debugging data race conditions by allowing to infer the order of instruction executed across all threads.

The information required to support this feature is collected by the means of the Memory Race Log (MRL), which is stored in th Memory Race Buffer. The aim of the memory race log is to keep track of the shared memory ordering among threads. MRL is created at each checkpoint along with FLL. Every thread records its synchronization data in its local MRL and these MRLs are kept in synchronization with the FLLs for each thread. An entry into MRL is created every time there is a coherence reply message for a shared memory access. S. Narayanasami, G. Pokam and B. Calder in their paper "BugNet; Continuously Recording Program Execution for Deterministic Replay Debugging" state: "The purpose of each record is to synchronize the execution of the remote thread, which is sending the coherence reply message, with the execution of the local thread by recording both of their instruction counts. Having this information allows us to retrieve the ordering of memory operations across all the threads" [SNC05].

Limitations of BugNet include the fact that it only catches errors that are identified by the application itself or the operating system. It cannot catch errors resulting from incorrect programming logic. Moreover, it cannot capture bugs that arise from complex interactions of user process with operating system, since it only tracks application code.

Also, BugNet only supplies developer with the data enough to replay the last few checkpoints before the application error occurred. This makes its record/replay scheme unsuitable for profiling purposes, something that Echo focuses on.

2.4 DejaVu

In their paper, "Deterministic Replay of Java Multithreaded Application",[CS98] J. Choi and H. Srinivasan discuss **DejaVu** a record/replay tool that provides deterministic execution replay of concurrent Java programs by capturing Java thread schedule.

The approach used by DejaVu to capture thread schedule is independent of the underlying operating system and it is based on modifying the Java Virtual Machine. Authors of the paper considered using bytecode instrumentation to capture the thread schedule instead of modifying the JVM. However, they decided against this approach, which is adopted by JReplay claiming it incurs too much overhead. This claim is contradicted by the results obtained by JReplay framework.

This paper introduces a notion of a logical thread schedule, which based on counting the number of critical events occurring between thread swapping. Critical events in this context are all synchronization events performed by threads, for example monitorenter and monitorexit. Shared variable accesses are also treated as critical events. The approach to capture logical thread schedule is based on using global clocks that ticks at each execution of a critical event to uniquely identify each critical event. It also uses one local clock per thread to allow each thread to identify schedule intervals that belong to that thread.

A pair of clock values FirstCriticalEvent and LastCriticalEvent is recorded for each schedule interval by a running thread. To capture the schedule interval for each thread, DejaVu uses the fact that threads local clock isnt incremented while it isnt running and hence global clock and local clock values will vary. When a thread starts executing a critical event, it compares its clock value to that of a global clock, if values are different a thread detects the end of the previous schedule interval and the start of a new

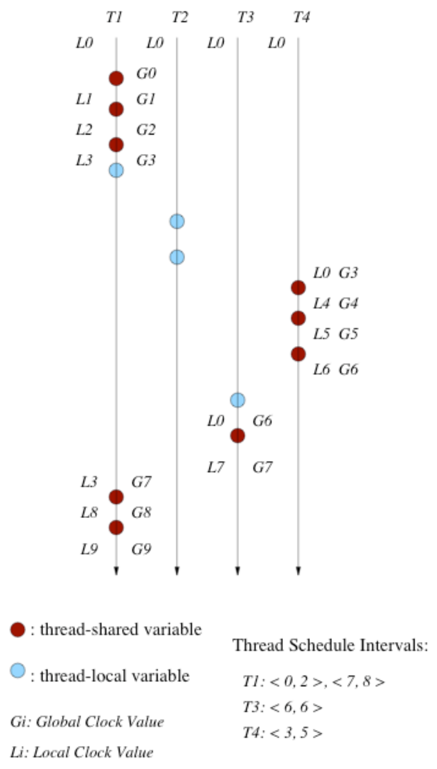


Figure 2.2: Identifying thread schedule [CS98]

schedule interval. Once the thread finish executing a critical event, it increments global clock and then synchronizes its local clock with the global clock (see Figure 2.2).

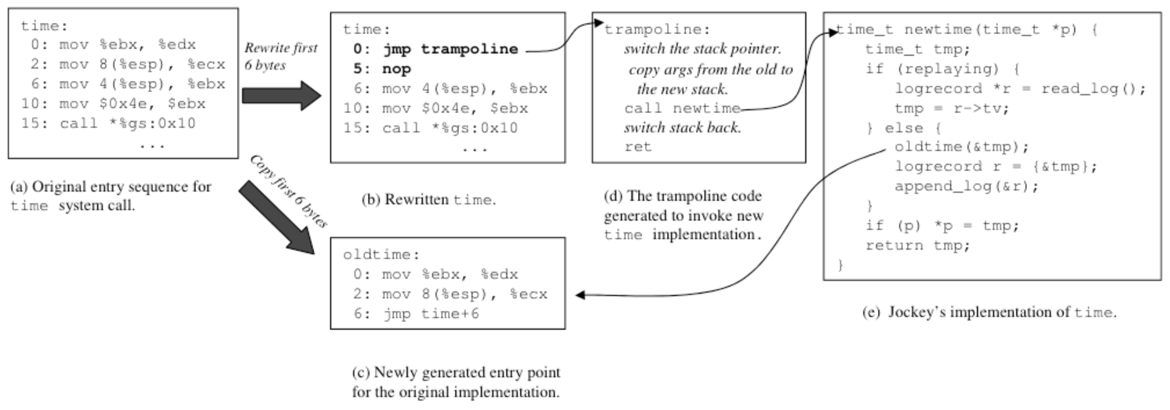
During the replay stage, DejaVu reads a thread schedule from a file generated at the end of the record mode. When a thread is created and starts execution, DejaVu supplies it with an ordered list of its logical schedule intervals. The thread sets its local clock to the value of the FirstCriticalEvent of the next schedule interval in the list it receives. The thread then waits until global clock value becomes the same as FirstCriticalEvent at which point the thread starts running. At the end of each critical event the thread checks whether global clock value becomes larger then LastCriticalEvent vavlue of the current interval, at which point the thread starts to execute the next schedule interval. When no more intervals are left, the thread terminates.

Main limitation of DejaVu is that it is designed specifically for Java programs. Also, it does not take into account I/O operations, which are likely to impact the execution of the program.

2.5 Jockey

This project is one of the most recent efforts in the research of deterministic replay frameworks.

In his paper "Jockey: A User-space Library for Record-replay Debugging" [Yas05], Yasushi Saito



```
time:
 0: mov %ebx, %edx
 2: mov 8(%esp), %ecx
 6: mov 4(%esp), %ebx
10: mov $0x4e, $ebx
15: call *%gs:0x10
          ...
```

```
time:
 0: jmp trampoline
 5: nop
 6: mov 4(%esp), %ebx
10: mov $0x4e, $ebx
15: call *%gs:0x10
          ...
```

```
oldtime:
 0: mov %ebx, %edx
 2: mov 8(%esp), %ecx
 6: jmp time+6
```

```
trampoline:
  switch the stack pointer.
   copy args from the old to
    the new stack.
  call newtime
  switch stack back.
  ret
```

```
time_t newtime(time_t *p) {
   time_t tmp;
   if (replaying) {
      logrecord *r = read_log();
      tmp = r->tv;
   } else {
      oldtime(&tmp);
      logrecord r = {&tmp};
      append_log(&r);
   }
   if (p) *p = tmp;
   return tmp;
}
```

Rewrite first 6 bytes
Copy first 6 bytes
(a) Original entry sequence for time system call.
(b) Rewritten time.
(c) Newly generated entry point for the original implementation.
(d) The trampoline code generated to invoke new time implementation.
(e) Jockey's implementation of time.

Figure 2.3: Recording and replaying of the **time** system call [Yas05]

presents **Jockey** a record/replay execution tool for debugging Linux programs. Jockey is quite similar to Echo in that it records invocations of system calls and CPU instructions with timing-dependency to later replay them deterministically. It also supports process checkpointing. However, Jockey is implemented in user-space whereas Echo is implemented as a kernel extension.

Jockey is implemented as a shared-object file and it runs as part of the monitored application. The heart of Jockey is libjokey.so, which is loaded before the monitored application starts execution. Jockey's initialization routine performs the following tasks before transferring control to the monitored application:

1. Rewrites first few instructions for each of the system calls in libc with timing or context dependent effects in order to be able to intercept them. Figure 2.3 shows what Jockey does in order to intercept system calls (with example of time system call). Jockey writes a jmp instruction in the first five bytes of the system call. It fills memory up with nop instruction until the next instruction boundary if the fifth byte is in the middle of another instruction. Jockey also copies the original first five bytes and stores it in the allocated memory region in case Jockey will have to run the original implementation of the system call. Jockey then manipulates the stack to make the system call arguments available to the new implementation of the procedure but to avoid corrupting the target stack. Finally, Jockey's implementation of the system call is invoked.

2. Similarly it patches CPU instructions with non-deterministic effects. In order to patch CPU instructions, Jockey has to determine which instructions to patch. It finds these instructions by looking in /proc/N/maps (N is the target process ID), which shows the virtual memory mappings of the target process. Jockey reads the headers of each mapped shared object file to find the location of the text sections and scans the text sections to find non-deterministic CPU instructions and patches them.

3. It captures the state of the process, which is required in order to ensure that programs environment is set up the same way during record and replay.

4. Finally it returns control to the monitored application.

The paper discusses the issues of resource segregation, which is an important issue for Jockey because it runs as part of the target program and thus share all resources.

- **Heap.** Jockey stores all its internal data in a memory mapped region at fixed virtual address, which is unlikely to be used by the monitored application.
- **Stack.** Jockey's approach to stack segregation is not to use the stack of the monitored application. When a system call is intercepted, Jockey saves the stack pointer into an internal variable, switches the stack to an internal buffer. It then copies the parameters to the system call from the old stack to the new stack and invokes its version of the system call. Once the call returns, Jockey restores the stack pointer.
- **File Descriptors.** File descriptors segregation is similar to the approach used to segregate the heap. Jockey moves file descriptors it uses internally (for example to store a checkpoint data to disk) to a fixed range, which is no likely to be used by the monitored application.

Shared-object file design has an advantage that it does not require modifications to the operating system or the target application to be made in order to record and replay program execution. However, it also has disadvantage of not being able to handle kernel-level multithreading, an issue that is addressed by Echo. Other limitations of Jockey include the fact that it can be compromised by a seriously misbehaving monitored application due to the way Jockey segregates shared resource from the monitored application. The way Jockey tries to minimize logging overhead can also be seen as a limitation. In order to reduce amount of information that has to be logged, Jockey lets some of the system calls to be re-executed during the replay stage, for example reading from a regular file. This means that the user cannot modify files that have been accessed by the monitored application between record and replay stages, which is something that is not always feasible.

2.6 Flashback

Flashback [SMSZ04] is another project in the field of deterministic replay. It offers similar functionality as Jockey and a subset of functionality offered by Echo. Flashback is implemented as an operating system extension and it provides deterministic execution replay and rollback to assist software debugging by capturing the interactions between the monitored program and the operating system which can happen through system call invocations, memory-mapping, shared memory usage for multi-threaded applications and signals.

Flashback works by taking checkpoints of the monitored application at particular points in time that can be specified either explicitly through calls to primitives provided by Flashback or interactively using a debugging tool that has support for Flashback framework. These checkpoints are stages in the execution of the monitored program, which the program may roll back to if required. Once the checkpoint has been invoked, Flashback enters a log mode during which it logs all interactions between the monitored process and the environment. Replay mode is entered when replay is invoked. During the replay mode, Flashback simulates the effects of original system interactions to the debugged process.

In their paper, S. M. Srinivasan, S. Kandula, C. R. Andrews and Y. Zhou concentrated a lot on the checkpointing mechanism implemented by Flashback. The approach adopted by Flashback is to capture the state of the system at a checkpoint uses a shadow process. A shadow process is a snapshot of the running process created by replicating the in-memory representation of the process in the operating system and its creation is achieved by creating a new structure in the kernel and initializing it with the contents of the

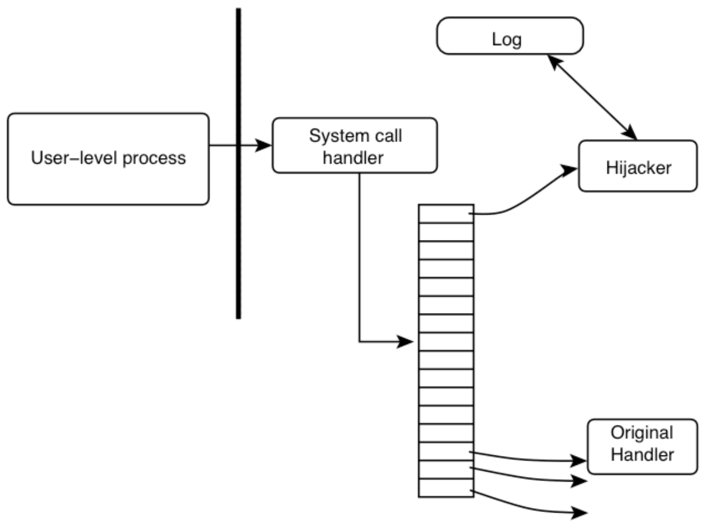


Figure 2.4: Hijacking system calls for logging and replay in Flashback [SMSZ04]

monitored process structure, such as registers contents, process memory, file descriptors and so forth. The pointer to the shadow process is stored in the current process structure. Multiple shadow processes can be created. If the rollback is requested, Flashback discards the current process state and rolls back to the previously captured shadow process.

Flashback supports rollback for multi-threaded applications by capturing the state of all threads of the process on checkpoint and rolling back all the threads to a previous executing point on rollback. This approach would allow to deterministically re-execute all threads, which would help identify any synchronization or data race conditions.

The paper also discusses the approach implemented by Flashback to capture the interactions between the monitored program and the system. Flashback intercepts system calls invoked by the monitored process during its original execution by replacing the original system call handlers with a function that does logging and replay (see Figure 2.4).

A very similar approach to intercepting system calls is implemented by Echo.

Main limitation of Flashback is that it requires modification to debugging tools to incorporate support for the framework. It allows programmers to modify the source code of the debugged program explicitly to include calls to Flashback primitives, however it does not allow the programmers to include their own debugging information to the program during the replay stage.

Flashback is also not suitable for profiling purposes because the replay mechanism does not allow the target program to be instrumented for the replay stage. Both debugging and profiling support issues are addressed by Echo.

Other limitations include the fact that recording and replaying of signals and deterministic replay of multi-threaded applications is outlined in the future work but it is not currently supported by Flashback.

2.7 JReplay

JReplay [Bau03] tries to achieve similar functionality to that of DejaVu a deterministic replay of multi-threaded applications by forcing the application to execute according to a specified thread schedule. However, instead of modifying the JVM, which is the approach used in DejaVu, JReplay enforces a specified thread schedule by instrumenting the bytecode of the original program.

Solution proposed by JReplay is independent of the underlying operating system and of the JVM implementation the program is running on. It is based on instrumenting the original program with calls to a special replayer class at all locations where the thread changes occur. These locations are identified by combining class name, method index and a bytecode offset of the instruction that triggered a thread change, the unique identifier of the thread that should be stopped running at that location and an iteration index specifying the number of successive invocations of the given bytecode instruction before the thread change occurs. This location information is stored in a text file, which overall specifies the thread schedule. JReplay does not offer functionality to obtain such a schedule, but instead assumes that such schedule is available.

To achieve deterministic replay, JReplay only allows only one thread to run at a time with all other threads blocked. The running thread is specified by the given schedule. To control which thread is currently running, JReplay assigns a lock object to each thread during the replay of the instrumented program. Threads are blocked and unblocked using these locks and Java synchronization mechanism. In order to transfer control from one thread to another, JReplay unblocks the next thread scheduled to run and then blocks the current thread.

The rest of the paper discusses bytecode instrumentation that needs to be applied to the target program to allow JReplay to control the thread execution of the program.

Even though, JReplay is an interesting and novel idea, it suffers from a number of limitations. First of all, it does not offer functionality to capture thread schedule and instead works on the assumption that such a schedule is somehow obtained and available. It does not support checkpointing and hence the program might require to run for a long time before it reaches the point where the error occurs. All subsequent replays must also start from the beginning of program execution.

JReplay only targets Java applications, however some types of Java applications such as applets, servlets and Java archives (jar) are not currently supported.

2.8 Performance counters

In order to achieve Echo's goals of using deterministic replay to debug and profile single and multi-threaded applications, it is necessary to consider the use of performance counters. Performance counters play a very important role in implementing support for replaying signals. The mechanism for signal replay is mentioned in Flashback and was introduced by J. H. Slye and E. N Elnozahy in their paper "Supporting Nondeterministic Execution in Fault-Tolerant Systems" [SE96]. The approach described uses software solution to emulate instruction counters, which can be used to determine timings between asynchronous events and thus replay them deterministically later. Echo uses a similar mechanism, however it does not rely on software emulation of counters and uses the actual hardware performance counters available on modern processors. Hence, it is very important to look into the capabilities of performance counters.

Performance counters can also be used to create profiles of the running application and since Echo's aim is to provide the users of the framework with performance monitoring tools it acts as another motivational factor to conduct research into performance counters.

2.8.1 Overview

Performance counters are part of the performance monitoring hardware tools available on most modern processors. Performance monitoring hardware usually consists of two parts: event counters and performance event detectors [Spr02a]. Performance event detectors and event counters can be configured to obtain data on various performance events such as pipeline stalls, cache misses, number and type of instructions completed by the program.

Structure and capability of performance monitoring hardware varies from processor to process, however the following overall features are available on the most processors:

- Performance event detectors can be configured to qualify event detection by the processors privilege mode. This allows counting performance events occurring only for the operating system or only for the user processes or both.
- Performance event counters can be configured to count events occurring under certain edge conditions. This edge detection feature is most often used for performance events that detect the presence or absence of certain conditions every cycle. For these events, an event count of one represents a condition's presence and zero indicates its absence.
- Performance event counter can also be configured to count the number of times the value it reports each cycle exceeds some threshold value.

  These and other features of the performance monitoring hardware make them useful for collecting performance profiles of applications:
- **Time-based profile** helps identify areas of the program that the applications spends most of it time on. Time-based profile is collected by interrupting the application at regular intervals and saving the program counter at each interrupt. When the program completes, a histogram can be drawn from the data collected showing the most frequently executed instructions.
- **Event-based profile** is similar to time-based profile, but it indicates the most frequently executed instruction that has caused a particular performance event. To collect event-based profile, the performance monitoring hardware interrupts the running application after a specific number of performance events have occurred. The mechanism used by performance monitoring hardware to support event-based sampling (EBS) is to initialize a performance counter with the overflow value minus the specified value of performance events N after which the sample should be taken. Once performance counter overflows, an interrupt occurs, which is handled by the performance monitor interrupt service routine (ISR). ISR saves required sample data and re-initialized the performance counter to cause another interrupt after N performance events. This technique is similar to that described in Flashback for handling signal replay and it is also similar to the one that is adopted by Echo.

Performance monitoring hardware suffers from some limitations. First of all, processors only have a limited number of performance counters that can count concurrently. Hence, it might be required to run the application a number of times in order to collect a variety of performance data. One of the most serious problems with most performance monitoring hardware is it the imprecision of the EBS. As described above, EBS is supported by signalling an interrupt when the performance counter overflows and saving the program counter value for the interrupted instruction. However, the fact that there can be a delay between the counter overflow and the signalling of the interrupt and also the fact that most modern processors are unable to correctly identify which instruction in the pipeline has caused an interrupt lead ISR to save the address of the wrong instruction. Other limitations of the performance monitoring hardware include lack of support for distinguishing between speculative and non-speculative event counts and lack of support for creating data-address profiles.

However, Intel's Pentium 4 processor overcomes most of these limitations and is discussed in the next section.

2.8.2 Performance monitoring hardware of Pentium 4

Intel's Pentium 4 processor overcomes most of the limitations of performance monitoring hardware discussed in the previous section. It has 18 event counters and a large number of event detectors (ranging from 43 to 45), allowing a significantly larger set of performance data can be captured concurrently. Pentium 4 also addresses the problem of counting non-speculative performance events by introducing instruction-tagging mechanisms. Another limitation overcome by Pentium 4 and that is most significant for Echo is the support for precise event-based sampling, which unambiguously determines which instruction has caused a performance event.

Non-speculative instruction count is obtained in Pentium 4 by instruction-tagging mechanism. As the processor decodes each instruction, it breaks it down into a sequence of simpler instructions, called microoperations. These microoperations are later tagged when they cause certain performance events, however, the counter counting the given event is not incremented until the tagged instructions pass through the retirement logic of the processor. At that stage, tagged instructions are counted providing a non-speculative event count. Tags of the instructions that never retire are discarded and hence do not contribute to the event count.

Pentium 4 supports four types of tagging mechanisms [1005]. Tagging mechanisms are independent of each other, which means that instructions tagged with one mechanism will not be counted by another mechanisms tag detector. Tagging mechanisms supported by Pentium 4 are:

- **Front-end tagging.** This mechanism tags microoperations responsible for the events occurring in the early stage of the pipeline related to instruction fetch, instruction count and trace caches.
- **Execution tagging.** This mechanism tags microinstructions that encountered execution events, such as instruction types.
- **Replay tagging.** This mechanism tags instructions that are replayed due to conditions such as branch misprediction or cache miss.
- **No tags.** This mechanism does not use tags.

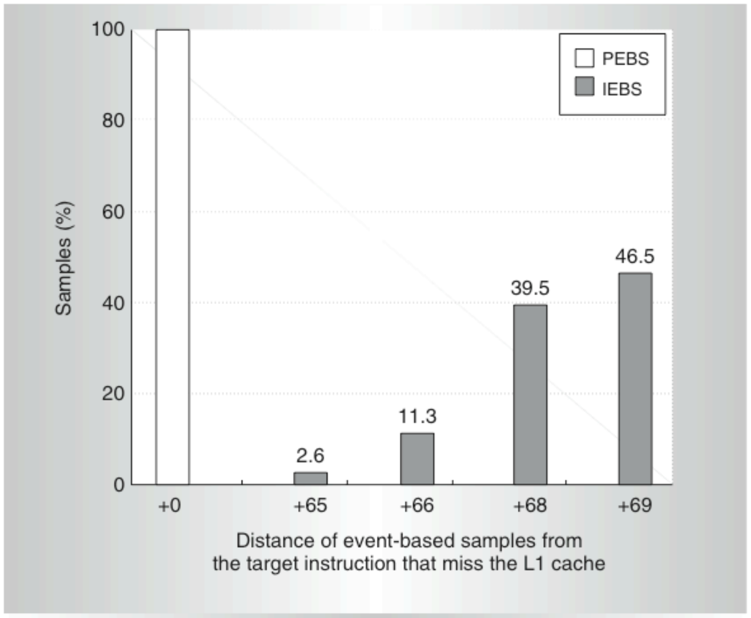


Figure 2.5: Accuracy comparison for PEBS and IEBS [Spr02b]

One major improvement of Pentium 4's performance monitoring hardware compared to other processors is its support for precise event based sampling. Pentium 4 uses a microassist and a microcode-assist service routine to precisely capture sample data. In order to use PEBS, the user must enable it and then configure the tagging mechanism and an event counter to count certain instructions as they retire. When the counter overflows, Pentium 4 inspects all the retiring microinstructions to find the tagged microinstruction, at which point a microassist occurs. A microassist halts the execution of instructions and the microassist service routine collects the sample data by storing the current program counter and the contents of general-purpose registers into a dedicated PEBS buffer specially allocated to hold samples. The precise sampling is ensured because the microassist is invoked immediately before the retirement of the instruction causing the overflow. It ensures that the data collected corresponds to the instruction causing an event and there is no latency between the retirement of the instruction and the invocation of the interrupt service routine.

Brinkley Sprunt in his paper "Pentium 4 performance monitoring features" [Spr02b] compares accuracies of imprecise and precise event based sampling. He has created a program that uses a load instruction in a nested loop to generate a lot of cache misses. However, only one instruction in a nested loop causes a miss and it is preceded and succeeded by a large sequential block of loads that always hit the cache. He has created a precise and an imprecise event-based profile of the cache misses of the program. Results of this experiment are self-explanatory and are shown in Figure 2.5.

Performance monitoring hardware is very complicated and difficult to configure and use. However, it provides a very powerful tool to for collecting performance data. It also allows very fine-grained control over what is collected and the circumstances the data is collected at. Since collecting profiles using performance counters is done on the hardware level, the overhead of collecting profiles this way is kept to the minimum, which is undoubtedly is an advantage over some other profiling methods.

2.9 Summary

This section has presented the state of the art in the field of deterministic replay. After reviewing several projects in this area, it became evident that research of replaying applications deterministically for purposes of debugging them is a very active. There is a number of projects and tools that attempt to solve this problem, however they all suffer from various limitations ranging from their inability to properly support multi-threaded applications to the fact that they only work for a particular API or on a particular architecture. Echo will attempt to build a deterministic replay framework by bringing together and elaborating on some of the ideas presented in different projects.

This section has also investigated performance monitoring hardware and the features it has to determine whether it can be applied to solve problems associated with deterministic replay of applications. It has also been considered how performance monitoring hardware can be employed to create profiles of the running applications.

Chapter 3

Design Overview

This chapter discusses the design requirements of the Echo framework and how they were realised. It also provides a high level overview of the overall architecture of the project. It discusses the main components, which comprise Echo, the responsibilities of each component and interactions between the main components in the system.

3.1 System architecture

The goal of Echo project is to provide a framework which allows to record the execution of a running application and then deterministically replay this execution at a later stage. The execution of any application involves an interaction with the operating system. This interaction consists of the running application making system calls to the operating system asking the os to perform actions on its behalf. The operating system handles these calls, performs required actions and delivers the results back to the running application. Therefore, in order to capture and subsequently replay the execution of the application, the framework has to intercept the system calls the running application is making and record their relative sequence and their results. To replay an application, the framework must provide a way of replaying this captured sequence of system calls and their results to the application being replayed.

The situation becomes more difficult when multi-threaded applications are being replayed. An execution of a multi-threaded application is inherently non-deterministic. This is because on different runs of the application, the operating system can schedule threads differently due to the availability of resources, other processes running on the system and various other factors. The result is that even if the sequence of system calls is recorded for each thread, the scheduling of threads on replay execution of the application might be different to the one during the record execution and hence the recorded system calls trace will be invalid. Therefore, in order to replay a multi-threaded application deterministically, the thread schedule also has to be captured during the record execution and then forced upon the running application during the replay execution.

These two requirements give motivation for the design of Echo framework and for the technologies used in its implementation. The overall architecture of Echo is quite simple but the complexity of the system lies in the intricate details of each component. The framework is implemented as a kernel extension and it consists of three main parts: the Logger, the Re-player and the Echo driver. The Logger and the Echo driver constitute the recording part of the framework, whereas the Re-player and Echo driver are responsible for the replay side of the framework. The following sections will describe each component in a more detail.

3.2 Echo driver

This is the heart of the framework. Echo driver is implemented as a kernel module and it is responsible for capturing and replaying the sequence of system calls made by the running application and also for capturing the thread schedule of the application and forcing this schedule upon the application at replay.

When the module is inserted into the operating system, it overwrites the entries in the system call table with its own function. This effectively allows the Echo driver to intercept all the system calls that are made. If the system call is made by the application that we are interested in, the call is handled by the module so that the system call number, status and the result can be recorded. If the system call is made by any other application running on the system, no data is recorded and the system call proceeds as usual. On replay, if the system call is made by the application that is being replayed, the module handles it by supplying the result of a given system call recorded earlier and not letting the operating system to service the call in the usual way. For all other applications, the original system call handlers are invoked and the calls are serviced as usual by the operating system. The way system calls are handled is discussed in much more detail in chapter 4.

Another major responsibility of the Echo driver is capturing the thread schedule when a multi-threaded application being recorded. The main approach to capturing a thread schedule relies on identifying the points in the execution of the program where the threads get swap out by the scheduler. On replay, once the thread preemption points in the execution are reached by the recorded application, the current running thread is forcefully blocked and the next thread to run according to the recorded schedule is woken up and scheduled to run, thus forcing the recorded application to stick to the recorded thread schedule.

Two ways to implementing this approach of capturing and replaying the thread schedule were considered in this project:

- **Instrument the application and use a software counter to count the number of times the program makes backward control transfer.** This techniques is the same as described in ”Replay for Concurrent Non-Deterministic Shared-Memory Applications” paper by M. Russinovich and B. Cogswell [RC96]. The idea behind this technique is that an application to be recorded and the shared libraries used by the application are instrumented such that every time a program does a backward control transfer, a counter is incremented. A thread preemption point in the execution of an application is then uniquely identified by the instruction pointer at that point and a value of the counter. When the preemption event occurs at the record phase, the instruction point and the counter value pair are saved in the log. During the replay, the counter is set to the negative value of the value stored in the log and when the counter increment reaches zero, a function is called which places a breakpoint on the instruction associated with that count. When a breakpoint is reached by the process or thread, the current running thread or process is forcefully preempted, the new pair of instruction pointer and counter value is retrieved from the log, the counter is set up to count down to a new preemption point and the next thread is scheduled to run.

  This is a neat way of capturing and replaying a thread schedule, however it suffers from a major drawback, which is the need to instrument the recorded application and the shared libraries it uses. Instrumenting an application and the shared libraries could result in an increase in the initial delay as the application starts up to run, which might be unacceptable for applications running in a production environment. Therefore it was decided against using this approach. A more detailed discussion of this approach, its implementation and how the limitations of the approach were overcome can be found in chapter 6.

- **Use performance monitoring hardware to count the number of instructions executed by each thread before it gets swapped out.** This technique is the one that was actually implemented in Echo. The idea behind this approach is to count the number of instructions executed by each thread before it gets swapped out using performance counters that are available on most modern processors. The performance counter is configured such that it gets incremented for each retired instruction, thus providing a count for a number of instructions that get executed by a process. When a thread is preempted, the value in the counter is read and stored in the timing log file. The counter is then reset to start counting instructions executed by a next thread that is scheduled to run. On replay, we are using a feature of performance counters to overflow and cause an interrupt when a particular counter value is reached. We preset the counter to counter overflow value - the number of executed instructions that was recorded for the current thread. This way, an interrupt occurs when the counter increments to the overflow value, which means that the current thread would have executed the number of instructions it executed during the record phase. When servicing the interrupt, the current running thread is blocked, the next timing data is read from a buffer, the counter is preset with new value and the next thread is scheduled to run.

  Unfortunately, this technique suffers from a limitation that the interrupts cause by performance monitoring hardware are imprecise. This means that there is a potential delay between the interrupt occurring and when it gets delivered to the operating system and hence it cannot be guaranteed that no more instructions will be executed by the current running thread after the counter has overflown. This poses a question: if on replay we cannot preempt the current running thread precisely at the point it was preempted during the record stage then how can the exact thread schedule be forced upon the running application? This problem was solved by using a technique of single step interrupts similar to that used in stepping through an application execution when it is being debugged.

3.3 Logger

Logger is a part of the Echo framework which is responsible for identifying the application to be recorded to the Echo driver and also for writing out recorded process data to disk. Logger runs as a user process whose arguments are the path to the the directory in which the log files are to be created, the application to be recorded and the arguments to that application. When the logger starts running, it creates two log files:

- **Args log.** This file holds the program's arguments and also environment configuration saved by the logger so that when the application is replayed, the conditions of execution are as similar to the conditions of the recorded environment as possible.
- **Syscall log.** This file will contain the trace of system calls made by the application to be recorded.

- **Timing log.** This file holds the thread schedule in terms of the number of instructions executed by each thread before it gets swapped out.

Both files are created every time the logger process is run and hence a number of different execution histories for each application can exist simultaneously.

The Logger than proceeds by opening a connection to the Echo driver and notifying it that an application is about to be run in a record mode. This allows the Echo driver to set up the buffers shared with the Logger process into which the data about system calls and the timings between thread executions will be recorded. Implementation, setup and handling of buffers is discussed in more detail in chapter 5.

Once the buffers are set up, the Logger forks off a new process. This forked process is the process that should the Echo driver should trace and record system calls and thread schedule for. The Logger notifies the Echo driver of the pid of the new forked process. The Logger then executes the application that was supplied as an argument to the Logger process by calling execve.

From this point, the Logger has a sole responsibility of writing out the data in the syscall and timing buffers to disk when the buffers become full or whenever the child process started by the Logger, i.e. an application that is being recorded, finishes and exits.

3.4 Re-player

Re-payer is a process running in user space which is responsible for identifying the application to be replayed to the Echo driver and for delivering the process data logged during the record stage to the Echo driver. A single argument to the replay application specify the directory containing the log files created by the Logger process during the record phase: the syscall log, the timing log and the args log. The replay process opens args log to read in the path to the application binary to be run, the arguments to the application and the environment configuration.

The Re-player proceeds in the same way as the Logger. It opens a connection to the Echo driver and notifies it that an application is to be run in a replay mode, which allows the Echo driver to set up shared buffers. The Re-player retrieves the process data recorded in the syscall and timing log files and puts it into the shared buffers that were set up by the Echo driver.

Once the buffers have been setup and populated with the process' system call and timing data, the Re-player forks off a child process, which is the process to be replayed. It informs the Echo driver of the pid of the child process and calls execve to execute the application that was recorded at an earlier stage.

The responsibility of the Re-player from this point onwards and until the child process terminates is to deliver the process' system call and timing data recorded in the log files to the Echo driver by reading it into the shared buffers from the log files.

3.5 Summary

This chapter presented a high level overview of the architecture of the Echo framework. The framework consists of three main parts: the Echo driver, the Logger process and the Re-player process. The structure and the responsibilities of each Individual component comprising Echo were explained as well as how the components interact with each other to produce a functioning system.

The chapter has also introduced the approach of capturing and replaying the thread schedule using performance counters as developed in Echo. It has compared this approach with another technique for capturing thread schedules, which relies on instrumenting the application to record/replay and the shared libraries. The benefits and drawbacks of both methods were analyzed and the reasons were discussed for choosing the method implemented in Echo.

Chapter 4

System calls

This chapter presents a detailed description of how system calls interception and handling mechanism was implemented in the Echo framework. The chapter also discusses in detail how the framework keeps track of the process to be recorded/replayed.

4.1 System calls interception

A major part of the Echo driver is concerned with capturing and replaying sequences of system calls that are made by the traced application. When a system call is captured by the Echo driver, it has to provide a way of handling this call. At present, only a subset of system calls are handled by Echo framework. System calls that are handled by the module have two types handlers associated with them, which are defined and implemented in the module. The system calls that are not currently supported by the Echo framework have special empty body `handle_ignored` and `handle_ignored_response` functions associated with them.

Two types of handlers defined and implemented in the module are:

- `syscall_handler.` This function is invoked before the original system call is allowed to proceed and during the record stage it is responsible for recording the system call number. On replay, this function is used to feed the result of the system call back to the running application.
- `response_handler.` This function is only invoked during the record stage and it is responsible for recording the status of the system call once it has finished and any data that it delivered to the running application.

The module keeps track of what handlers are associated with each system call by keeping two arrays of handlers. The size of both arrays is the size of the system call table and each entry in both arrays corresponds to the entry into the system call table with the same index number. Thus, an array element `syscall_handlers[5]` contains a handler for the system call number 5, i.e. `open` system call.

When the Echo driver is loaded into the kernel, it goes through the system call table and copies the pointers to the original system call handlers into the `original_syscall_table` that is kept in the Echo driver. This is done so that the original system call handlers can be invoked from the Echo driver to service the system calls made by applications running in the system.

The Echo driver proceeds by overwriting the entries in the system call table with a pointer to an intercept function defined in the module. This way, whenever an application makes a system call, an intercept function is invoked in the module thus allowing the module to keep track of all the system calls made. The array of system call handlers and the array of system call response handlers are both initialized to `handle_ignored` and `handle_ignored_response` respectively. Then entries corresponding to the system calls that are supported by the framework are then overwritten with pointers to their respective handlers implemented by the module.

To get a better understanding of how this works in practice, consider a usual mechanism for servicing system calls. The library function puts the system call number into a register where an operating system expects it. It then executes a trap instruction to switch from a user mode to kernel mode and starts executing from a fixed memory address within the kernel. The kernel then inspects the system call number and dispatches to the correct handler by indexing into the system call table on the system call number. Since the system call table contains pointers to the system call handlers, the correct handler runs and services the request. Once the handler has finished, the control is returned to the library function at the instruction following the trap instruction [1801].

When the Echo driver is inserted into the kernel, the mechanism for performing system calls is modified in the following way. The library function still puts the system call number into a register and executes a trap. The kernel starts to run, examines they system call number and dispatches to the system call handler by indexing into the system call table. However, all the entries in the system call table have been overwritten with the pointer to `syscall_intercept` by the Echo driver and hence any system call made by the running application invokes `syscall_intercept` function. At this point, the steps taken to perform a system call vary depending on whether the application that has made a system call is being recorded or replayed.

Figure 4.1 shows the steps taken during the record stage to perform a system call:

1. `syscall_intercept` function invokes the `syscall_handler` corresponding to the current system call being serviced by indexing into the `syscall_handlers` array.
2. The `syscall_handler` records system call number of the current system call and returns
3. Since this is a record stage, the original system call handler has to run to service the system call. Therefore, the `syscall_intercept` calls the original system call handler by indexing into the `original_syscall_table` table which contains pointer to all the original system call handlers.
4. Once the system call handler returns, the `response_handler` function is called to record the data returned by the system call handler and the status of the system call. Once `response_handler` returns, the `syscall_intercept` function returns as well and the control is returned to the library procedure.

During replay, the original system call handler does not need to run, since the results of system calls are already available and retrieved from the log file. Hence, steps taken to replay a system call (shown in Figure 4.2) are as follows:

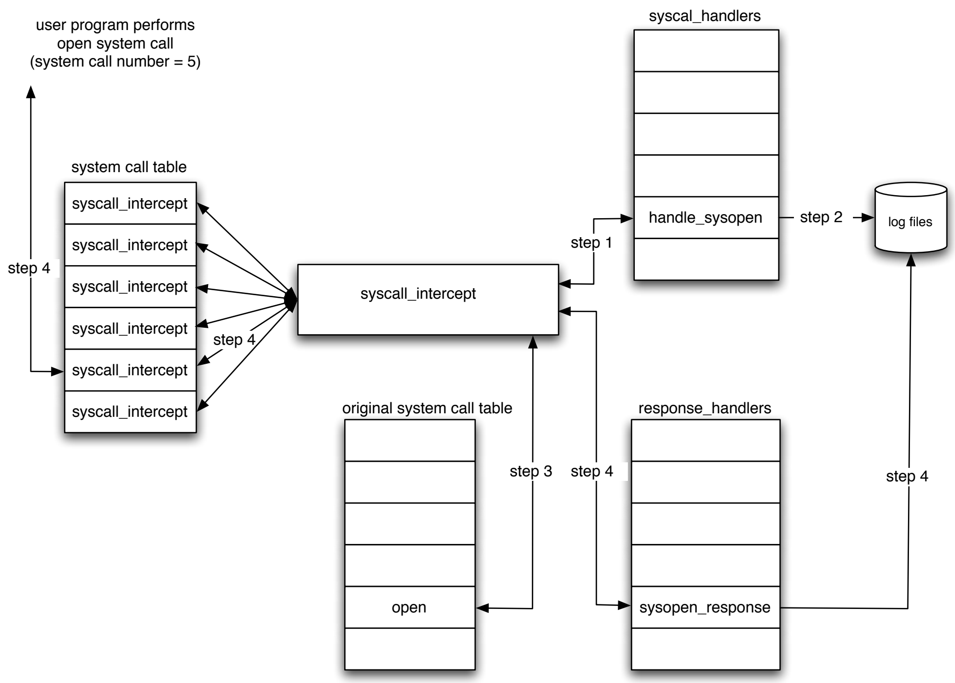


Figure 4.1: Interception of system calls during the record stage

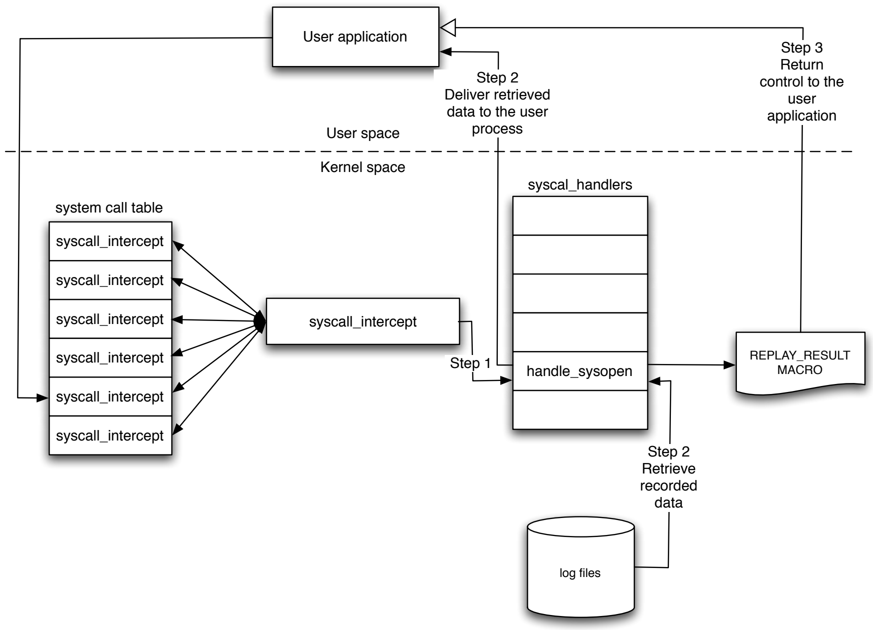


Figure 4.2: Interception of system calls during the replay stage

1. `syscall_intercept` function invokes the `syscall_handler` corresponding to the current system call being serviced by indexing into the `syscall_handlers` array.
2. `syscall_handler` retrieves the data and the status corresponding to the current system call from the log. It then copies retrieved data to the user space and then calls a special `replay_result` macro, which is used for returning the status of the system call.
3. `replay_result` puts the status of the replayed call into `eax`, which is where the status of the system call is expected to be found by the application that has invoked a system call. `replay_result` then cleans up the stack by manipulating `ebp` and `esp` pointers to peel off the address, which returns to the `syscall_intercept` function. It then jumps to the `syscall_exit` address through which all system calls return and hence returns control to the calling user process. This way, during replay stage, the control is never returned to the `syscall_intercept` function and hence the original system call handler is never invoked.

4.1.1 Intercept function

`syscall_intercept` function is the heart of the system call interception mechanism. Figure 4.3 shows its implementation.

When `syscall_intercept` is called, it invokes `handleSyscallRequest` function, which checks whether the current running process is the one that is being recorded or replayed and if it is, dispatches it to the `syscall_handler` corresponding to the current system call. When `handleSyscallRequest` returns, the original system call handler has to run.

To invoke the original system call handler, there are a few things that needs to be done. First of all, original handlers are stored in the `original_syscall_table` array, which must be indexed on the system call number to get the pointer to the correct system call handler corresponding to the current system call. Therefore, the system call number must be obtained and preserved for later use. This is done in line 5 where the system call number is put into `ebx` register. We now need to return the stack to the state it would have been in if the system call handler was called normally, without having to go through `syscall_intercept`. This is done by lines 6 – 8. The line `mov %ebp, %esp` and `pop %ebp` takes care of de-allocating any local variables and restoring the frame pointer and the `add $4, %esp` gets rid of the return address of the `syscall_intercept` caller. The stack is now in the state it would be just before system call handler is called, containing registers that were pushed onto the stack for as arguments for the handler.

Lines 9 – 11 are responsible for invoking the original system call handler. We get the address of the base of the `original_syscall_table` and then index into it using the system call number, which is stored in `ebx`.

When the original system call handler returns, we need to store the return value of the system call, which is done in line 15. Lines 16 – 20 handle invoking the `handleSyscallResponse` function, which is only used during the record stage. `handleSyscallResponse` checks whether the current process is the one that is being recorded and if it is, it invokes the `response_handler` function corresponding to the current system call. `handleSyscallResponse` passes the registers onto the response handler and hence the registers must be provided to the `handleSyscallResponse` as an argument. Since we have cleaned up the stack in order to call the original system call handler, the local `registers` pointer (see line 1 of the Figure 4.1) is

```
static int syscall_intercept(unsigned int first_register)
{
  1.   struct register_set* registers = &first_register;
  2.   handleSyscallRequest(registers);

  4.  // Dispatch to original system call handler
  5.  __asm__("mov %0, %%ebx" :: "m"(registers->orig_eax));
  6.  __asm__("mov %ebp, %esp");
  7.  __asm__("pop %ebp");
  8.  __asm__("add $4, %esp");
  9.  __asm__("mov %0, %%eax" :: "m"(original_syscall_table));
  10. __asm__("movl (%eax,%ebx,4), %eax");
  11. __asm__("call *%eax");

  13. // Traced process has returned from system handler

  15. __asm__("movl %eax, 0x18(%esp)");
  16. __asm__("push %esp");

  18. handleSyscallResponse();

  20. __asm__("pop %esp");
  21. __asm__("jmp 0xc0107007");

  23. return 0;
}
```

Figure 4.3: Implementation of syscall_intercept function

not longer accessible. Therefore, we have to do `push %esp`, which points to the base of the registers that were pushed onto the stack as arguments to the system call handler. When the `handleSyscallResponse` function returns, we need to do `pop %esp` to clean up the stack again.

The response handler, invoked from `handleSyscallResponse` records the system call data into the log. The data recorded varies depending on the system call. For example, for the an `open` system call only the return value of the call is recorded, whereas for the the `read` system call the return value of the system call is recorded along with the data that has been read by the system call on behalf of the calling process.

Once the system call has been handled and the relevant data has been recorded, the `syscall_intercept` exits via `syscall_exit`, which is the way system calls exit normally. This is done by line 21, where `0xc0107007` is the address of the `syscall_exit` routine.

```
#define REPLAY_RESULT(X)
    1. checkUserSpaceLock();
    2. __asm__("mov %0, %%eax" : : "r"(X));
    3. __asm__("mov (%ebp), %ebp");
    4. __asm__("mov (%ebp), %ebp");
    5. __asm__("mov %ebp, %esp");
    6. __asm__("pop %ebp");
    7. __asm__("add $4, %esp");
    8. __asm__("movl %eax, 0x18(%esp)");
    9. __asm__("jmp 0xc0107007");
```

Figure 4.4: Implementation of REPLAY_RESULT macro

4.1.2 Replay result macro

The `replay_result` macro is used for replaying the return value of the recorded system call. What makes it special is the fact that it never returns to the function from which it has been invoked. This allows Echo driver to bypass returning to the `syscall_intercept` and hence invoking the original system call handler during replay.

`replay_result` is implemented as a sequence of preprocessor directives and its code is shown on Figure 4.4.

When the `replay_result` macro is invoked, the return value of the recorded system call is passed as a parameter to the macro. In line 2 this value is put into the `eax` register, where it is expected to be found by the operating system on the return of the system call.

Lines 3 – 4 manipulate the stack base pointer such that allows erasing the return addresses that were put on the stack by `syscall_intercept` and then in turn by `handleSyscallRequest` and therefore allowing `replay_result` never to return to the calling function. Consider a stack shown on Figure 4.5.

When `syscall_intercept` is called, the `ebp` gets pushed onto the stack to save `syscall_intercept` caller's frame pointer. When `handleSyscallRequest` is called from `syscall_intercept` function, the frame pointer of `syscall_intercept` is saved by pushing `ebp` onto the stack. Similarly, `ebp` pointer of `handleSyscallRequest` gets pushed onto the stack when a `syscall_handler` gets called. In order to bypass returning to the `syscall_intercept` function and instead exit through the `syscall_exit` procedure, we need to know the value of the `syscall_intercept`'s caller `ebp`. This is accomplished by `mov (%ebp), %ebp` instruction. The first one on line 3 retrieves the value of `handleSyscallRequest` base pointer and the second one on line 4 obtains the value of the `syscall_intercept`'s `ebp`.

We can now clean up the stack after `syscall_intercept` and get rid of the return address put on the stack by the `syscall_intercept`'s caller. This is done in lines 5 – 7. On line 8, we store the result of the system call and exit the via the same mechanism as system calls normally exit by jumping to `syscall_exit`.

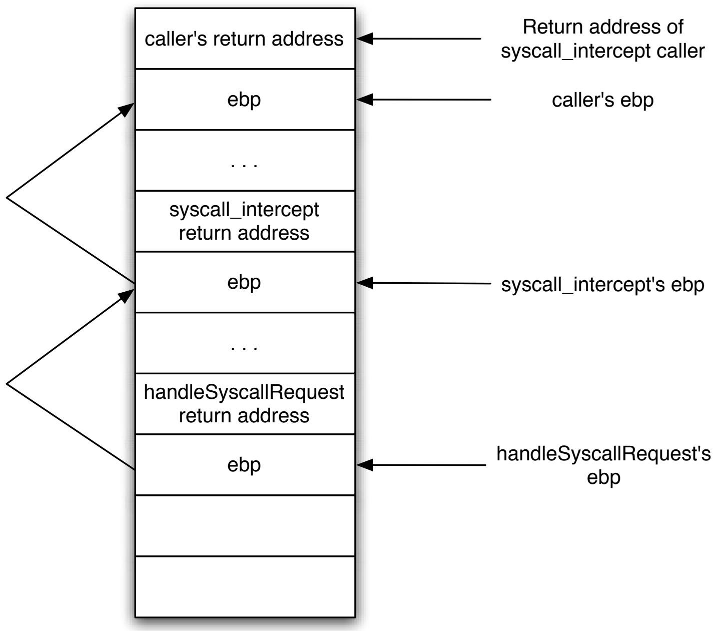

caller's return address
ebp
. . .
syscall_intercept return address
ebp
. . .
handleSyscallRequest return address
ebp
Return address of syscall_intercept caller
caller's ebp
syscall_intercept's ebp
handleSyscallRequest's ebp

Figure 4.5: Contents of the stack when REPLAY_RESULT macro is invoked

4.2 Process tracing

When recording or replaying application's execution, we need to make sure that we only capture or replay information that belongs to the application we are interested in. For single threaded applications it is enough to identify the application to be traced to the Echo driver to make sure that only the relevant information is recorded. This can be done by obtaining and storing the application's pid in the module and checking that the application making the system call has the same pid as the application being traced. The situation is more difficult for multi-threaded applications, since each thread created by the recorded application has to be identified to the Echo driver and traced. Similarly for applications that spawn new processes by using `fork`.

A generic mechanism, which enables tracing any number of processes simultaneously was implemented to support recording and replaying execution of multi-threaded applications.

4.2.1 Process state structure

Each process that is being traced by the Echo driver has a `process_state` structure associated with it. The `process_state` structure and all its fields are shown in Figure 4.6.

This structure is used by the Echo driver to record the state of each process that is being traced. The fields that are relevant for process tracing are:

- `logger`. This is a pointer to the `logger_state` structure, which describes the Logger or Re-player

```
struct process_state
{
  struct logger_state* logger;

  unsigned long mode;
  unsigned long paused_mode;
  unsigned long current_call;
  unsigned long child_id;
  unsigned long pid;
  unsigned long block_in_kernel;
  struct semaphore userspace_lock;
  struct process_state* next_in_read_queue;
  struct process_state* next_in_child_queue;
};
```

Figure 4.6: Process state structure

process that is used to record or retrieve execution data from log files. A single Logger is used to record the execution data of a traced process and all the threads that it spawns. However, if there are a number of different applications being recorded or replayed in the system at the same time, separate Logger processes will be used for each of the applications.

- `mode`. This field is used to identify whether a current running process is being recorded or replayed. There are two modes a process can have: `MODE_RECORD`, `MODE_REPLAY`. They are self-explanatiory and are set by the Echo driver when a command is received from the Logger or the Re-player to record or replay an application.
- `child_id`. This field is used to hold the id of the child process spawn by the traced application. It should not be confused with the `pid` of the process. In the Echo framework, the initial process to be traced is started by the Logger or the Re-player and hence the traced application itself is a child of either a Logger or a Re-player process and it has a `child_id` of 1. The threads spawn by the traced application are in turn its children and will be assigned ids from 2 onwards. The reason for using `child_ids` and not just relying on the `pids` of the traced threads is because `pids` of threads will change between record and replay and hence if we record the thread schedule using the `pids` of threads, we will not be able to use this information during the replay. The `child_id` field, however, is determined by the Echo driver and does not change between record and replay stages, which allows the Echo driver to reliably record and replay the thread schedule.
- `pid`. We still record the `pids` of threads because we need this piece of information to be able to block and resume threads to force an application to stick to a recorded thread schedule.

`paused_mode` is used to indicate that the process contains special debugging code and while this debugging code is being executed the system calls should not be replayed. This is discussed in more detail in chapter 7. The fields `block_in_kernel`, `usesrspace_lock`, and `next_in_child_queue` are used when thread timings are recorded and replayed and are discussed in more detail in chapter 6. `next_in_read_queue` field is used in implementation of buffering and its purpose is discussed in chapter 5.

The Echo driver keeps an array `processes` of process structures. The size of the array is 65536, which is the maximum number of the processes that can run on Linux operating system. Therefore, the Echo driver can trace any of the processes running on the system. Every entry in the `processes` array is initialized to `NULL` when the Echo driver is loaded into the kernel. Every time a new thread is created or a new child has been forked by the running application, a new `process_structure` is being created and stored in the `processes` array, which is indexed on the process pid. Therefore, entries in the `processes` array that are not `NULL` represent the processes that are being traced by the Echo framework. A `NULL` entry in the `processes` array means that the process with a `pid` corresponding to the index of an entry is not being traced by the Echo framework and hence its interactions with the operating system are ignored by the Echo driver.

The Echo framework keeps track of the current running process by using the `current` task structure defined in `linux/shed.h`. The `current` is the structure in which the kernel keeps all the information it needs about the current running process. The Echo driver references the `current->pid` field to obtain the `pid` of the current running process. It then uses this information to work out whether the current running process is one the traced processes by indexing into the `processes` array on the `pid` of the current running process. The Echo driver defines this entry as `CURRENT_PROCESS_ENTRY` as follows:

```
#define CURRENT_PROCESS_ENTRY  (processes[current->pid])
```

The `CURRENT_PROCESS_ENTRY` is used whenever the Echo driver needs to determine whether the current running process is being traced by Echo.

4.2.2 Initial process identification

The last section explained how the the Echo driver keeps track of the state of each thread spawned during the execution of the recorded/replayed application and also how the Echo driver is able to keep track of which thread is currently running. This section goes into detail of how the parent application is initially identified to the Echo driver.

The application to be recorded is identified to the Echo driver by the Logger process. The Logger process takes the path to the application to be recorded as an argument and executes it once the Echo driver is ready to trace it.

When the Logger process starts running, it opens a connection to the Echo driver and executes `MONITOR_PARENT` command to identify itself and to allow the Echo driver to setup the shared buffers with the Logger which are needed to record system call and timing information about the recorded program. The Echo driver also creates a `logger_state` structure, which describes the Logger process. Once this is done, the Echo driver is ready to trace an application.

In turn, the Logger process does a `fork` to spawn the application to be recorded as its child. It then executes `MONITOR_CHILD` or `REPLAY_CHILD` command depending on whether the process is to be recorded or replayed. This is a way of notifying the kernel that it should begin tracing. The Echo driver creates and fills in a `process_state` structure for the application to be traced. When the control, is returned to the Logger process, it executes `execve` to start the traced application.

Threads that are subsequently created by the initial traced application also need to have a `process_state` structures created for them and stored in the `processes` array in order for the Echo driver to correctly identify that they are also need to be traced. This is done when the `clone` system call is handled by the Echo driver, which gets invoked whenever a new process is spawned. The steps taken by the Echo driver to include the newly spawned thread to the set of traced process are identical to those taken for the initial traced application: a new `process_state` structure is created and filled for each new traced thread and this structure is stored in the `processes` array.

4.3 Summary

This chapter has presented a detailed description of how system calls interception and handling mechanism was implemented in the Echo framework. The Echo driver overwrites the system call table with the `syscll_intercept` function which allows the framework to intercept and monitor all the system calls the are made by running applications in the system. A mechanism of using handler functions to collect, record and replay information about system calls was developed, which allows the framework to handle record and replay of different system calls differently depending on the functionality they provide.

The chapter also discusses a mechanism used by Echo to keep track of what processes are being traced by the framework. The mechanism developed works by creating a `process_state` structure for each traced process and storing it in the `processes` array, which is indexed on the `pid` of the process. The traced processes are identified by indexing into the array, whose entries are all `null` except for those that contain process state structures of the traced processes.

Chapter 5

Implementation of buffers

This chapter discusses how the shared buffers between the Echo driver and the Logger and Re-player processes were implemented. It also discusses the problems that arose during implementation of buffering and how these problems were solved.

5.1 Types of buffers

There are two types of buffers implemented in Echo framework:

- **syscall buffer.** This buffer is used for recording return values of system calls and data that results from a system call being made, for example for a `read` system call data that has been read on behalf of user process making a system call is recorded in this buffer. The contents of the syscall buffer is stored in the syscall log file.
- **timing buffer.** This buffer is used for recording the number of instructions executed by each thread before it gets swapped out. The contents of this buffer are stored in the timing log file.

These buffers are shared between all the threads that were spawn by the initial recorded application. The entries written to each buffer consist of the header and the body. The header for an entry in the timing buffer contains the `child_id` of the thread to which the current entry corresponds. The body of a timing buffer entry consists of a integer, which represents the number of instructions executed by a thread in before it was swapped out for another traced thread.

The body of the syscall buffer entry contains the data returned by the system call. The header of the syscall buffer entry contains three pieces of information:

- The `child_id` of the current running process.
- The return value of the recorded system call.
- The size of the data length of the data returned by the system call that is to be written to log.

During the replay stage, the data that have been stored in the syscall and timing log files are retrieved by the Re-player process and read into the syscall and timing buffer respectively so that it is available for the Echo driver to deliver to the replayed application in the case of the syscall data. The timing data is used during replay to stop and swap processes once they have executed the same number of instructions as that specified by the records in the timing buffer.

5.2 Shared buffers

Both syscall and timing buffers are implemented as shared buffers between the Echo driver and the Logger or Re-player processes. The reason for this is that the system call and timing data that is recorded or retrieved by the Echo driver must also be accessible by the Logger process or the Re-player process. The Logger process must write the contents of the buffers to disk to store it, whereas the Re-player must retrieve the data previously stored in the log files and make it available to the Echo driver. Such design of involving separate user processes to writing and retrieving log data makes sure that the application that is being recorded or replayed will not be required to write or read its own log data and therefore the will be no danger of it being blocked on those operations. This obviously benefits performance.

The data recorded or required by the Echo driver could alternatively be made available to the Logger or obtained from the Re-player by copying the contents of buffers to and from the user and kernel space. Thus, for instance, the Logger process would have its own buffer allocated to it in the user space and the Echo driver would have a kernel space buffer allocated to it. The Echo driver would then make the data it recorded available to the Logger process by copying the contents of its kernel buffer to the user space buffer allocated for the Logger process. The Logger could then write out the contents of its user space buffer to disk. Similarly, on record, the Re-player would read the data from the log file into its user space buffer and the Echo driver would have to copy that data into the kernel buffer allocated to it. This design would result in a lot of unnecessary copying of data, which would have an adverse effect on performance.

When the Logger or Re-player process starts running, it opens a connection to the Echo driver by calling `open` as follows:

```
int fd = open("/dev/kmodule", O_RDWR);
```

The file descriptor `fd` returned by `open` is then used whenever the Logger or the Re-player processes need to communicate with the Echo driver.

Once the connection to the Echo driver is opened, the Logger or Re-player executes a `MONITOR_PARENT` command to indicate to the driver that it needs to prepare for recording or replaying an application. The Echo driver then creates the system call buffer and the timing buffer as follows:

```
CURRENT_DATA_BUFFER = vmalloc(BUFFER_SIZE);
CURRENT_TIME_BUFFER = kmalloc(TIME_BUFFER_SIZE, GFP_KERNEL);
```

The size of the timing buffer in the current implementation is only 10K and therefore the memory for it is allocated using `kmalloc`. However, since `kmalloc` can only allocate memory for up to 128K, `vmalloc` was used to allocate memory for a system call buffer. This is because the system call buffer needs to be big enough (in this implementation its size is 655K) to cope with amount of system call data generated by the running application without the need to block the application that is being recorded or replayed.

Once the buffers are allocated in the Echo driver, the Re-player or Logger processes are able to map them into user space and subsequently access them from user space. This is done using the `mmap` function:

```
syscall_buffer = (char*) mmap(NULL, BUFFER_SIZE, PROT, MAP_SHARED, fd, 0)
timing_buffer = (char*) mmap(NULL, TIME_BUFFER_SIZE, PROT, MAP_SHARED, fd, 0)
```

where the `PROT` parameter is set to `PROT_WRITE` for the Re-player process and to `PROT_READ` for the Logger process. The file descriptor `fd` provided as a parameter to the `mmap` indicates that the operation should be performed by the Echo driver. In turn, the Echo driver provides its implementation of the `mmap` function and makes the kernel aware of this alternative implementation by registering it in the `file_operations`structure along with any other functions supported by the Echo driver:

```
static struct file_operations fops =
{
  .open = module_open,
  .ioctl = module_ioctl,
  .mmap = module_mmap
}
```

Thus, whenever a `mmap` function is called with a file descriptor corresponding to the Echo driver, a `module_mmap` function gets invoked, which is implemented in the Echo driver.

5.2.1 mmap method

The responsibility of the `mmap` method implemented by the device driver is to build a page table for the address range being mapped and if necessary handle any page faults that might result. For a discussion of page tables and memory management in Linux, see [16].

There are two ways to build a page table. One of them uses `remap_page_range`, which allows to build all of a page table at once. The other method is to use the `nopage` VMA method, which allows to build a page table one page at a time. The `nopage` method is specific to each driver and it is called by the kernel whenever a user process attempts to access a page in a VMA that is not present in memory. The responsibility of the `nopage` method is to locate and return the `struct page` pointer of the page that the user process wants.

The Echo driver employs both methods. It uses a more sophisticated `nopage` method to map a system call buffer and it uses a simpler `remap_page_range` method to deal with mapping a timing buffer. The reason for that is that the timing buffer is quite small and it is critical for it to be kept in memory during the recording or replaying of an application, therefore we lock the pages in memory and use a straightforward method of building a page table. The system call buffer, on the other hand, is quite big and it is not so important for it to be kept in memory all the time. Thus, a nopage methods was used to deal with situations when the requested system call buffer page is not in memory and has to be located by the Echo driver. Figure 5.1 shows the implementation of the module_mmap function.

The CURRENT_MMAP_SELECT is an integer variable associated with each Logger or Re-player process that allows to select which buffer is being mapped. In the case of MAPPING_SYSCALL, a nopage method of building a page table was used and hence we have to make the kernel aware of the implementation of nopage function provided by the Echo driver. This is done by registering driver's implementation of a nopage method in the vma_operations_struct as follows:

```
struct vm_operations_struct vm_ops =
{
  nopage: vma_nopage,
};
```

and setting the vm_ops field in the vma_area_struct sturcture to the address of the vm_ops structure defined in the Echo driver, thus indicating to the kernel what methods can be invoked on the mapped memory area.

When the system call buffer page requested by the Logger or Re-player is not in memory, the vma_nopage method implemented by the Echo driver will be invoked by the kernel to locate the requested page. The implementation of vma_nopage function is given below:

```
struct page* vma_nopage(struct vm_area_struct* vma, unsigned long address, int write_access)
{
  struct page* page_ptr = vmalloc_to_page(&(CURRENT_DATA_BUFFER[address - vma->vm_start]));
  get_page(page_ptr);
  return page_ptr;
}
```

The vmalloc_to_page returns the pointer to the page associated with the address obtained from vmalloc [27]. In our case, it will return the pointer to the page, associated with the address, which represents a particular offset into the system call buffer. The offset into the buffer is computed by address - vma->vm_start, where the address is the address causing a page fault and vma->vm_start is the address of the start of the buffer.

The get_page function needs to be called to make sure that the page usage count gets incremented. This is necessary because the kernel maintains this count for every page and when the count goes to zero, the kernel knows that the page may be placed on the free list. When a VMA is unmapped, the kernel will decrement the usage count for every page in the area and if the count doesn't get incremented for a page that is being swapped in, the usage count will become zero prematurely, which will result in unpleasant consequences [16].

```
static int module_mmap(struct file* filp, struct vm_area_struct* vma)
{
  switch (CURRENT_MMAP_SELECT)
  {
    case (MAPPING_SYSCALL):
    {
      printk("Mapping syscall buffer to logger memory space.\n");

      vma -> vm_ops = &vm_ops;

      break;
    }

    case (MAPPING_TIMING):
    {
      struct page* pstart;
      struct page* pend;
      struct page* page;

      vma -> vm_flags |= VM_LOCKED;
      vma -> vm_flags |= VM_RESERVED;

      printk("Mapping timing buffer to logger memory space.\n");

      pstart = virt_to_page(CURRENT_TIME_BUFFER);
      pend = virt_to_page(CURRENT_TIME_BUFFER + TIME_BUFFER_SIZE);

      for (page = pstart; page < pend; page++)
      {
         printk("Reserving page.\n");
         SetPageReserved(page);
      }

      remap_page_range( vma, vma -> vm_start, (unsigned long) virt_to_phys(CURRENT_TIME_BUFFER),
                        TIME_BUFFER_SIZE, PAGE_SHARED);
    }
  }

  CURRENT_MMAP_SELECT++;

  return 0;
}
```

Figure 5.1: Implementation of module_mmap function

5.3 Double buffering

In order to record the execution of an application, we need to ensure that the size of the buffers that is used to hold data related to the execution is big enough to fit in the data generated throughout the execution time of the recorded application. Unfortunately, this requirement is unreasonable, especially for the and there are two major reasons why it cannot be fulfilled:

- **Application vary in their levels of communication with the system.** Some applications are very computationally intensive and therefore they might spend most of their time performing mathematical operations with very little interactions with the operating system. This kind of applications will not, in general, perform a lot of system calls and hence will not generate much data to be recorded in the system call buffer. Therefore, this type of applications will require relatively small buffers to hold the execution data they generate. However, some applications communicate a lot with the operating system, e.g. I/O bound applications, and therefore they will generate large traces of system calls that they make. For this programs a large sized buffers are required to hold the execution data they generate. Unfortunately, it is very difficult to predict how much execution data will a given application generate before it runs and hence it is impossible to predict what size should the buffers be for a given application.
- **The size of the buffers required may exceed the size of the available memory.** Even if we could predict how big the buffer should be for a given application, in the worst case scenario, this size could well exceed the size of the available memory.

In order to overcome these limitations, a double buffering mechanism was implemented in Echo framework. The idea behind double buffering is to divide the buffer into two segments and to present these two segments as two separate buffers. When the first segments fills up, the Echo driver will start write data into the second segment while the Logger process is writing out the contents of the segment that is full to disk. When the second segment becomes full, the Logger should have finished writing out the contents of the first segment to disk and therefore the first segment is now available for the Echo driver to write data into it.

However, it should be noted that the double buffering has its limitations and there are problems, which cannot be solved with double buffering. These problems arise from situations where the traced application generates execution data to be recorded at a much higher rate than the rate with which these data can be written to disk. In this case, both buffer segments will become full before the Logger process will be able to write one of them to disk. The way round these situations is to block the recorded application and therefore prevent it from generating any more data before the Logger has an opportunity to write out the buffer segment to disk. Whilst this solution has an adverse effect on performance, the situations causing this problem are unlikely to occur very often.

The implementation of double buffering uses a well known producer-consumer model, which employs semaphores. Once the application to be recorded starts running, the Logger process takes on the role of the consumer whereas the Echo driver is the producer. The implementation uses two semaphores:

- `CURRENT_LOGGER_LOCK` is used to notify the kernel whether the Logger process is ready to receive data to write out to disk.

- `CURRENT_CALLBACK_LOCK` is used to notify the Logger process whether there is data to write out.

The semaphores are initialized to 0 when the first `MONITOR_PARENT` command is received by the Echo driver from the Logger. Once the recorded application is started by the Logger, the Logger sits in a `while` loop and calls the kernel with the `LOGGER_CALLBACK` command. When this command is received by the Echo driver, the following semaphore operations are performed:

```
up(CURRENT_LOGGER_LOCK);
down(CURRENT_CALLBACK_LOCK);
```

The `up` operations ensures that the kernel is aware that the Logger is available to write out buffer segment to disk. The `down` operation notifies the Logger whether there is any work to be done. If `CURRENT_CALLBACK_LOCK` is greater than 0 when the `down` is performed, the Logger goes away and writes the contents of the buffer segment out to disk. However, if the `CURRENT_CALLBACK_LOCK` is 0 when the `down` operation is performed, the Logger blocks and waits for the buffer segment to fill up, at which point the Echo driver will perform an `up` operation on `CURRENT_CALLBACK_LOCK` semaphore.

The Echo driver is responsible for recording execution data into the buffer and ensures that all writes to the buffer go through a dedicated `writeToStream` function. This function checks if the current segment is full and if it is, it performs `down(CURRENT_LOGGER_LOCK)`. At this point, if the Logger is available, the kernel can proceed and setup the structure, which will let the Logger know how much data to write to disk. The Echo driver then performs `up` operation on `CURRENT_CALLBACK_LOCK` semaphore to wake up the Logger process and signal to it that the write out data is now ready.

If, however, the Logger is not available, because it hasn't finished writing out the previous segment yet, the `down` operation ensures that the kernel blocks the application being recorded until the Logger finish writing out the previous segment and hence becomes available again. This mechanism ensures that the recorded application doesn't keep running and generating execution data if the both buffer segments are full. This situation arises due to the limitation of double buffering techinique as discussed above.

5.4 Buffer race conditions

When the traced application is multi-threaded, we need to watch out for race conditions in writing to the buffer and hence access to the buffer must be protected. This is done by introducing two semaphores:

- `CURRENT_WRITE_LOCK` is used to protect writes to the buffer by the Echo driver when the application is being recorded.
- `CURRENT_READ_LOCK` is used to protect reads from the buffer by the Echo driver when the application is being replayed.

Whilst the write lock is sufficient to ensure that the buffer access is protected and race conditions free during the record phase, the situation is more difficult when a multi-threaded application is being replayed. This is because, even though Echo framework controls the thread execution during the replay stage, it only does so in the user space. Therefore, when threads enter the kernel, the framework doesn't monitor their execution and hence their interactions remain nondeterministic. The reasons for such design and its implementation are discussed in detail in chapter 6.

Since the thread interaction within the kernel is nondeterministic during the replay stage, it is possible that the order in which they read log entries from the buffer is not the same as the order in which those entries were written. This would result in the replayed execution being invalid, because the system calls and their results retrieved from the log will not be the same as expected by the replayed application. Hence a mechanism must be implemented to make sure that any thread that reads log entries from the buffer only reads its own entries.

The implemented mechanism works by suspending any thread that tries to read buffer log entries out of order. When a thread comes in to read an entry from the buffer, the entry header is read from the buffer and the thread ID of the current thread is compared with that of the ID of the expected thread to read the log retrieved from the entry's header. If these IDs are not the same, then the current thread is trying to perform out of order read from the buffer. In this case, the current thread is suspended and added to the read queue.

In order to suspend the current running thread, the following functions provided by Linux were used:

```
set_current_state(TASK_UNINTERRUPTIBLE);
schedule();
```

`set_currrent_state(TASK_UNINTERRUPTIBLE)` changes the state of the current running thread from `TASK_RUNNING` to `TASK_UNINTERRUPTIBLE`. This means that the current running thread is being suspended in such a way that it can only be woken up by an explicit `wake_up`. `schedule()` is called to ask the kernel to schedule another process.

The suspended process is put on a read queue, which is implemented as a linked list of `process_state` structures. The start of the queue is pointed to by the `CURRENT_READ_HEAD` pointer, which is unique for each Re-player process. The suspended process is always added to the start of the read queue and hence the `CURRENT_READ_HEAD` points to the `process_state` structure of the most recently added process. The `process_state` structure of the process that was added to the queue prior to the newly added process is pointed to by the `next_in_read_queue` field of the `process_state` structure of the newly added process. It, in turn, holds the pointer to the `process_state` structure of the next process in the queue, and so on until the end of the read queue is reached.

In the case where the current running thread is the next thread that is expected to read from the buffer, it is allowed to proceed. Once the read from the buffer is performed, the next buffer entry header is read to determine the process ID of the next scheduled process to read data from the buffer. The read queue is then scanned to determine whether the process with the ID of the next process to read from the buffer has been suspended. If such process is found, it is woken up by the issuing an explicit wake up call by executing `wake_up_process(find_task_by_pid(current_process -> pid))`. The read queue pointers are manipulated to reflect that the woken process was take from the read queue.

5.5 Summary

This chapter focused on the implementation of buffering system within the Echo framework. In particular, the following topics were discussed:

- The types of buffers used in the framework and what kind of data gets written into each type of buffer.
- Reasons for using a shared kernel buffering system to record the system call data and the timing data generated by the traced application. The implementation of shared buffers was also covered in great detail.
- A mechanism of double buffering. The reasons for implementing double buffering and how it was realized in the Echo framework.

The chapter concluded by discussing a problem of buffer race conditions, its implications and the solution to the problem that was implemented in the framework using semaphores.

Chapter 6

Thread schedule capture

This chapter discusses in detail how the thread schedule for a traced application is being captured during the recording stage and forced upon the application during the replay stage. It focuses on the use of hardware performance counters to count the number of instructions executed by each thread and how the performance counter overflow interrupts are used to stop the running thread after it has executed a given pre-recorded number of instructions. It also explains how the threads are being blocked and resumed during the replay stage to ensure that the thread scheduling is identical to that of the record phase. The chapter also discusses the problems that arose during the implementation of this part of the project and how the Echo framework solves these problems.

6.1 Approach overview

Chapter 3 ”Design Overview” gave a brief overview of how the thread schedule is captured at record stage and then forced upon the traced application at the replay stage. In this chapter we go into more detail about how such mechanism was implemented.

Most modern operating systems support kernel threads. This means that the operating system is in control of scheduling of threads and will swap out the current running thread after it has been running for a given period of time determined by the operating system or if it has been blocked on a system call. Clearly, any thread will only be able to execute a particular number of instructions during the time it was given to run. The approach used in Echo to capture thread schedule is based on this kernel threading model and is very intuitive: count and record the number of instructions executed by each traced thread during the record stage and during the replay stage only allow traced threads to run for a number of instruction that was previously recorded for them.

The implementation of the approach, however, is more subtle and needs to consider and deal with potential problems. First of all, in order to know when the traced thread starts running and when it gets swapped out we need to be able to intercept context switches occurring in the system and to distinguish between the processes we are interested in that are being restarted or swapped out and all other processes that are being started or swapped out by the kernel. We also need a fast and reliable but also a flexible way of counting the number of instructions that were executed by traced threads.

Performance counters were used to count the number of instructions executed by each thread. Since they are available as part of the performance monitoring hardware that can be found on most modern processors, using them for counting purposes will have a negligible effect on the overall performance of the recorded application. Moreover, they can be configured to count events satisfying some specific conditions and hence allow for the flexibility that is required in implementing this approach. During the record stage, when one of the traced threads starts running, the performance counter is configured such that it gets incremented for each retired instruction, thus providing a count for a number of instructions that get executed by a traced process. When a thread is preempted, the value in the counter is read and stored in the timing log file. The counter is then reset to start counting instructions executed by a next thread that is scheduled to run.

Performance counters provide another useful feature, which is crucial to the implementation of this approach – they have ability to cause interrupts when the value in the counter reaches the overflow value. This feature is used during the replay stage. When one of the traced threads starts running, the performance counters are setup such that the counter overflow value will be reached after the number of instructions previously recorded for this threads gets executed. Registering an interrupt handler to service these counter overflow interrupts enables the Echo driver to control thread execution schedule by suspending the current running thread and restarting the next thread according to the recorded thread execution schedule.

The rest of this chapter focuses on how each individual part comprising this approach is implemented in the Echo framework.

6.2 Context switch interception

In order to be able to count the number of instructions executed by each traced thread, we need to be able to intercept context switches occurring in the system. This is done by inserting a hook into the kernel, which makes sure that the kernel calls a function defined and implemented by the Echo driver every time a context switch occurs.

Kernel hooks are a mechanism of inserting and invoking specific, user-defined handler routines at desired location in the kernel. The hook to intercept context switches is inserted into the `context_switch` function. This function gets invoked on every context switch that occurs in the system. It is defined in `sched.c` file in the Linux kernel. File `sched.c` contains code, which is responsible for the scheduling that is done by the kernel and other housekeeping performed by the kernel which is required when scheduling and swapping processes.

In order to be able to use the kernel hooks mechanism, we need to define a hook to be inserted. This is done as follows:

```
void (*p_recordRunningDuration)(task_t* prev, task_t* next) = NULL;
EXPORT_SYMBOL(p_recordRunningDuration);
```

`p_recordRunningDuration` is a pointer to the context switch handler defined and implemented in the Echo driver and `EXPORT_SYMBOL` is used to export this pointer to the rest of the kernel and kernel modules. The hook is inserted in the `context_switch` function before the kernel does any work to process a context switch. When the Echo driver is loaded into the kernel, the context switch handler is registered to be called at this hook:

```
p_recordRunningDuration = recordRunningDuration;
```

where `recordRuningDuration` is a pointer to the context switch handler implemented in the Echo driver.

This way, once the Echo driver is loaded, whenever a context switch occurs and the `context_switch` function in the kernel gets invoked, it calls the `recordRunningDuration` function implemented in the driver. `recordRunningDuration` function is responsible for enabling and disabling performance counters depending on whether the processes involved in the context switch are being recorded or replayed. The functionality and implementation of `recordRunningDuration` is discussed in much more detail later in this chapter.

When the `recordRunningDuration` has finished its job, it returns to the `context_switch` function and the kernel continues to perform a context switch as it would normally.

6.3 Performance counters

Use of performance counters to capture and replay a particular thread schedule is a central concept in the Echo project. Performance counters offer a reliable and flexible way of counting the number of times a particular even occurs during an execution of an application. In the case of recording and replaying thread schedule, an event to be counted is the number of committed instructions executed by a traced application when it runs in user mode. This is clearly quite a complex requirement and requires configuring performance counters in such a way that all these conditions are met before the value counted by them can be of any use to the Echo framework.

6.3.1 Configuring the counters

Each of the 18 performance counters on a Pentium 4 processor has a **counter configuration control register (CCCR)** and a number of **event selection control registers (ESCR)** associated with it. Performance counters are configured to count an occurrence of a particular event by setting up their corresponding `ESCR` and the `CCCR` with flags describing the event.

The purpose of a `ESCR` is to select specific events to be counted. Figure 6.1 shows the layout of the register. When setting up an `ESCR` , the event select field is used to select a specific class of events to count, such as executed instructions. The even field mask is then used to select one or more of the specific events within the class to be counted. For instance, when counting instructions executed, the following events can be counted: retired instructions, non-retired instructions (for example, due to the branch misprediction). The `OS` and `USR` flags allow counts to be enabled for events occurring when the operating system code and/or application code are being executed.

The `CCCR` register controls the filtering and counting of events as well as interrupt generation. Figure 6.2 shows the layout of an `CCCR` register. Since each performance counter is controlled by an associated `CCCR` , in order to start event counting, the `CCCR` must be setup. To initiate the counting of events, the `Enable` bit in the `CCCR` must be set. The `ESCR select field` must be set to the address of the `ESCR` register, which is configured to select events to be counted. The `OVF_PMI` flag determines whether a performance

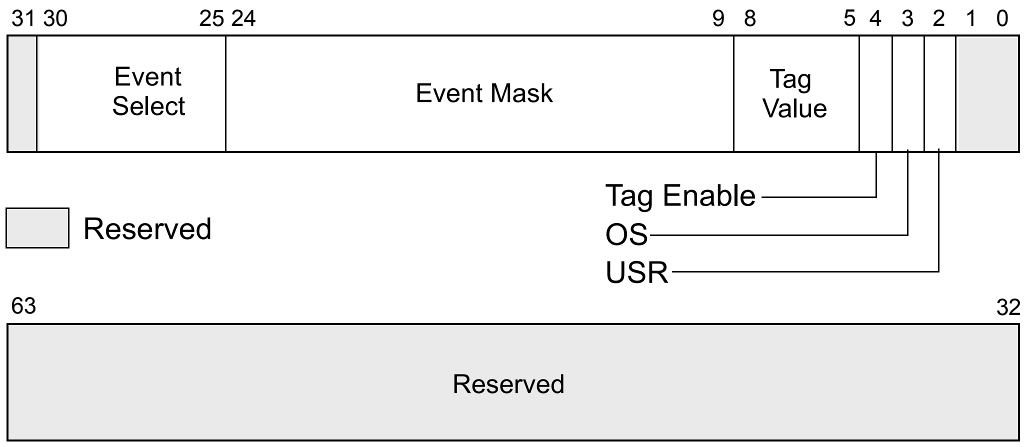


Figure 6.1: Layout of the ESCR register [1005]

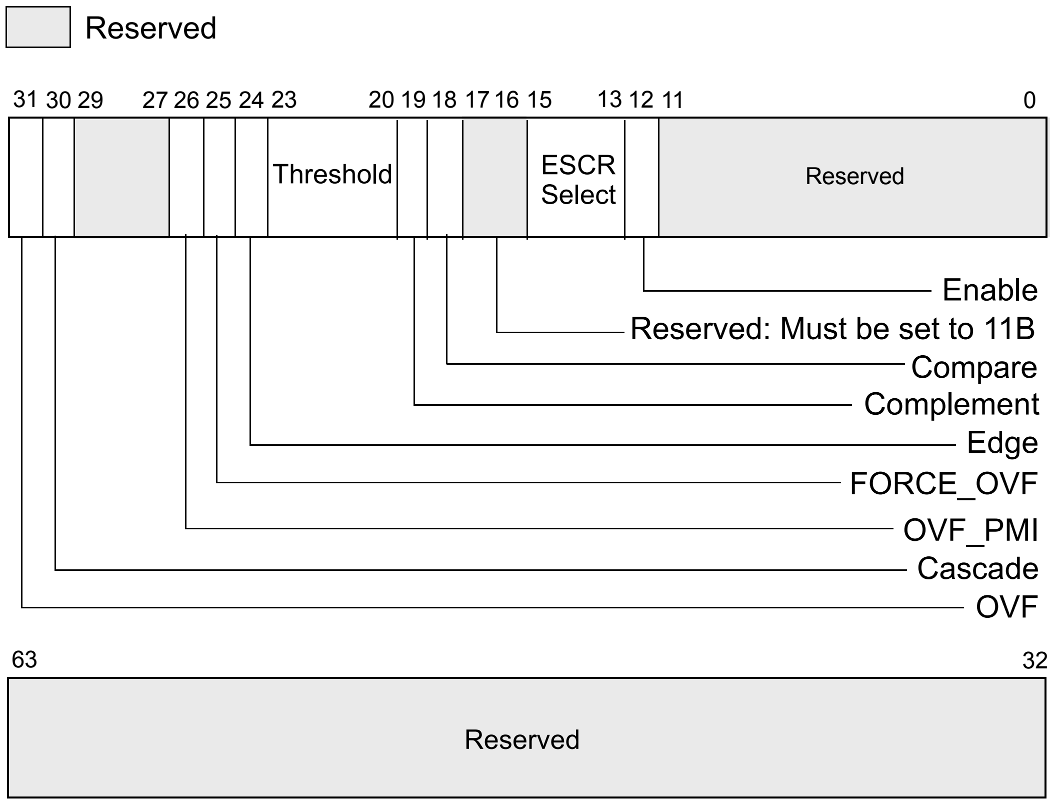


Figure 6.2: Layout of the ESCR register [1005]

monitoring interrupt should be generated when the counter overflow occurs. Hence, this bit must be set in order to enable performance counter interrupts.

A facility to read and write the performance counters is provided by the operating system and consists of:

- `wrmsrl(unsigned long msr, unsigned long long val)` function uses the `wrmsr` Pentium instruction to write the value provided in `val` to the counter at the address specified by the `msr`.
- `rdmsrl(unsigned long msr, unsigned long long val)` function uses the `rdmsr` Pentium instruction to read the value in performance counter at address specified by `msr` into a `val` variable.

6.3.2 Interrupt handling

In order to make use of the performance counter interrupts generated when a particular counter overflow value is reached, the Echo driver must install an interrupt handler which will receive and deal with this kind of interrupts.

Interrupts generated on performance counter overflows are generated through the local APIC (Advanced Programmable Interrupt Controller). The "IA-32 Intel Architecture Software Developer's Manual" describes the operation of APIC as follows: "Upon receiving a signal from a local interrupt source [such as performance-monitoring counter interrupts], the local APIC delivers the interrupt to the processor core using an interrupt delivery protocol that has been set up though a group of APIC registers called the local vector table or LVT. A separate entry is provided in the local vector table for each local interrupt source, which allows a specific interrupt delivery protocol to be set up for each source" [1005]

In order to install the interrupt handler for performance counter interrupts, the operating system kernel has to be modified in a number of ways. First of all, we need to define a new interrupt vector to be used on performance counter overflow interrupt. This is done by inserting the following line:

```
#define LOCAL_EBS_VECTOR        0xee
```

into the `irq_vectors.h` file in the operating system, which contains `#defines` for all of the interrupt vectors available on a given platform. A `entry_arch.h` file must also be modified to include the following line:

```
BUILD_INTERRUPT(ebs_interrupt, LOCAL_EBS_VECTOR)
```

`BUILD_INTERRUPT` will build the assembly code for the ebs interrupt. This code will save registers on the stack and call an interrupt handler routine for the `ebs_interrupt`.

The interrupt handler is registered with the APIC by adding a `set_intr_gate(LOCAL_EBS_VECTOR, ebs_interrupt)` line into the initialization function of the APIC. This function is defined and the `apic.c`. The interrupt handler `smp_ebs_interrupt` is also added to the `apic.c` and it gets invoked when a performance monitoring interrupt occurs:

```
void smp_ebs_interrupt(struct pt_regs *regs)
{
  ack_APIC_irq(); /* acknowledge the receipt of interrupt to the local APIC */
  if (*p_handle_ebs_interrupts != NULL)
     p_handle_ebs_interrupts(regs);
}
```

`p_handle_ebs_interrupts` is a kernel hook to get the interrupt handler to invoke a function in the Echo module on the receipt of performance monitoring interrupt. A handler implemented in the Echo driver is registered with this hook when the Echo driver is loaded into the kernel.

6.4 Recording instructions executed

Ability to precisely record the number of instructions executed by each particular thread is one of the main concepts of the project. The Echo framework distinguishes between the instructions executed by the process running in user mode and the instructions executed by the process running in a kernel mode and only user mode instructions are recorded. The reason for it is that since the Echo driver services intercepted system calls differently depending on whether the traced application is being recorded or replayed, the number of instructions executed in kernel mode will be different between the record and replay stages. Therefore, if the instructions executed in kernel mode were included in the count, it would not be possible to reliably determine when a given thread is supposed to be stopped and swapped for the next thread in the schedule during the replay stage. This implementation requirement results in the fact that the threads are not monitored and not controlled by the Echo driver when they run in the kernel. This leads to potential buffer race conditions as discussed in chapter 5 and also has implications for thread suspension and restart as done during the replay stage. This is discussed in more detail later in this chapter.

To accommodate for this implementation requirement, the performance counters must be configured to only count user mode instructions. This is done by setting `USR` flag in their corresponding `ESCR` and disabling the `OS` flag. The counters must also be configured to only count instructions that have retired and disregard any instructions that are on a path taken from a mispredicted branch. This is done by setting `ESCR`'s event mask bit to `NBOGUSNTAG`.

6.4.1 Process creation, execution and termination

Controlling process creation, execution and termination is crucial for capturing and replaying a thread schedule and therefore the Echo driver implements handlers for each of the system calls that is responsible for these states in a process' lifetime:

- `handle_sysclone`. On handling the `clone` system call, the Echo driver creates a new `process_state` structure for the newly cloned process and adds it the `processes` array in order to be able to identify that this newly created process should be traced by the framework. It also initializes a `userspace_lock` semaphore for the created process, which is used throughout the replay stage in the execution of a traced application as one of the ways of stopping and resuming the created process when to make sure the thread schedule is replayed correctly.

If the traced application is currently running in record mode, the process is allowed to start running straight away, since during the record stage we are interested in the unaltered behaviour of the traced application. However, during the replay stage, only one traced thread is allowed to run at a time and what this thread is is determined by the captured thread schedule. Therefore, during the replay stage, the newly created process should be allowed to run straight away, if its id is the same as the id of the process next scheduled to run according to the recorded schedule, otherwise it should be blocked until it is its turn to run. The `handle_sysclone` routine performs this comparison and acts accordingly.

Finally, since the Echo driver has the authority to block and resume processes, it must keep track of all the blocked processes and be able to find and unblock the next process scheduled to run. Therefore, the Echo driver keeps a linked list of processes that have been blocked by it. This list is implemented in a similar fashion to the one used to keep track of which thread has been suspended when trying to read the log file out of order. The `logger_state` structure that is used to describe the Logger process contains a `child_queue_head` pointer, which points to the start of the blocked processes list. The newly created process is added to the start of this list and the `next_in_child_queue` pointer in its `process_state` structure points to the `process_state` structure of the process that used to be at the head of the list. This purpose of this list is explained in more detail later in this chapter.

- `handle_sysexecve`. The handler for the `execve` system call is responsible for resetting the performance counters for each of the thread that was created by the original traced process.
- `handle_sysexitgroup`. This handler gets called when any process stops running and exits. However, it does not do any work unless the exiting process is the initial traced process, in which case the handler must ensure that the timing data about the last quantum of the initial traced process and the contents of the syscall buffer is recorded.

6.4.2 Recording running duration

Once the traced thread is running and the performance counters have been initially setup for it, all subsequent changes in its state until its termination are a result of a context switch performed by the operating system and hence they are intercepted by the Echo driver and are processed by `recordRunningDuration` handler.

The arguments to the `recordRunningDuration` function are pointers to the `task_struct` of the process to be swapped out (`prev`) and the process that is to be swapped in (`next`). When `recordRunningDuration` function is invoked, it must determine whether either of the `prev` or `next` processes that are involved in the current context switch is being traced by Echo.

There are three possible scenarios of traced processes being involved in a context switch:

- **Neither of the processes are being traced by Echo.** The Echo driver uses the `pid` of the `prev` process to index into the `processes` array of the `process_state` structures, which it keeps to identify which processes running on the system are being traced by the framework. If the entry in the array is `NULL`, then the current `prev` process to be swapped out is of no concern to the Echo driver as it is not being traced by the framework. In this case, no action is taken by the Echo driver.

- **`prev` process is the one that is being traced by Echo and the last context switch did not involve the any other processes being traced by Echo.** When one of the processes traced by Echo is involved in a context switch, the Echo driver stores its `pid` in a `CURRENT_RUNNING_PID` variable for future reference. It also reads the value in the performance counter into a `CURRENT_RUNNING_INSTR` variable, which represents instructions executed by a traced process since the last context switch. When a new context switch occurs and the `prev` process is one that is traced by the framework, its `pid` is compared to the `pid` of the last traced process that was involved in the context switch, which is stored in the `CURRENT_RUNNING_PID` variable. If the `pid`s are the same, which would be the case if no other traced process has been involved in the context switch since the last time a traced process was swapped out, the value in the performance counter is read. This value is then added to the `CURRENT_RUNNING_INSTR`. The updated `CURRENT_RUNNING_INSTR` will hold the number of instructions executed by a given traced process since it was last swapped in for another traced process. This updated value is stored in the Echo driver and is not written to the timing buffer yet. This is because the number of executed instructions that is recorded is the number of instructions that is executed by a thread before it gets swapped out for another thread that is traced by the Echo framework.

- **`prev` process is the one that is being traced by Echo and the last context switch involved one of the other processes traced by Echo.** In this case, the `CURRENT_RUNNING_PID` will not equal to the `pid` of the `prev` process that is getting swapped out and hence the current `CURRENT_RUNNING_INSTR`, which holds the number of instructions executed by the traced process that was last swapped out before the current context switch, must be recorded to the timing log. The `CURRENT_RUNNING_PID` is then updated to hold the value of the `prev` process' `pid`. The `CURRENT_RUNNING_INSTR` is reset to the value read from the performance counter. This value is the number of instructions executed by the `prev` process during its last quantum.

Performance counters are reset to 0 on every context switch for every process that is next scheduled to run irrespective of whether it is being traced by Echo or not. This allows the Echo driver to only concern itself with the `prev` process and whether any information about instructions it has executed during the last quantum should be recorded or updated. The `next` process that is scheduled to run has no impact on the setup of the performance counters or the number of instructions executed in the last quantum and hence is not considered by the Echo driver. Resetting performance counters on each context switch irrespective of the processes that are involved in it also means that the value read of the performance counter when the context switch is processed always represents the number of instructions executed by the process to be swapped out in the last quantum.

Figure 6.3 shows part of a `recordRunningDuration` function that deals with recording instructions executed by traced threads:

6.5 Replaying instructions executed

During the replay stage, when the parent application executes `execve`, the first entry in the timing buffer is read. The entry contains the id of the process to be replayed, which is assigned to a

```
static void recordRunningDuration(task_t* prev, task_t* next)
{
  unsigned long instructions_in_context;
  rdmsrl(READ_ADDRESS, instructions_in_context);

  if ((processes[prev->pid] != NULL) && (CURRENT_MODE == MODE_RECORD) &&
       (CURRENT_RUNNING_PID != AWAITING_EXEC))
  {
    if (prev->pid != CURRENT_RUNNING_PID)
    {
      if (CURRENT_RUNNING_PID != INITIAL_PID)
           recordRunningPIDEntry();            /*write entry into the timing buffer */

      CURRENT_RUNNING_PID = prev->pid;
      CURRENT_RUNNING_INSTR = instructions_in_context;
    } else {
      CURRENT_RUNNING_INSTR += instructions_in_context;
      instructions += instructions_in_context;
    }
  }
                            .
                            .
                            .

  /* Reset performance counters */

  wrmsrl(CCCR_ADDRESS, CCCR_FLAGS);
  wrmsrl(ESCR_ADDRESS, ESCR_FLAGS);
  wrmsrl(READ_ADDRESS, 0);
}
```

Figure 6.3: Implementation of `recordRunningDuration` function

`CURRENT_REPLAY_CHILD` variable to be referenced when the context switch is serviced. It also contains a number of instructions executed by the thread with that id before it was swapped out for another traced thread. This value is put in the `replay_instructions` field of the `logger` structure. The `replay_instructions` of the thread currently being replayed is pointed to by the `CURRENT_REPLAY_INSTR` variable and is used when servicing a context switch.

When a context switch occurs, the Echo driver once again has to determine whether either of the processes involved in the switch are being traced by the framework. On replay, however, the driver is concerned with both the `prev` process and the `next` process to be scheduled. There are a number of possible scenarios of `prev` and `next` processes involvement in the context switch that are dealt with by the Echo driver:

- **Neither of the processes are being traced by the framework.** In this case no action is taken and the `recordRunningDuration` function returns to allow the kernel to complete work required to perform a context switch.
- **`next` process to be scheduled is traced by the Echo driver.** The Echo driver determines that `next` process is being traced by the framework by indexing into the `processes` array on the `next->pid` and finding that the entry in the array in not `NULL`. In this case, the performance counters have to be setup so that an interrupt is caused when `next` thread has executed a number of instructions as that recorded for it during the record stage. This is done by presetting a performance counter to the value of at which the counter overflows less the number of instructions that should be executed by a `next` thread and by configuring the `ESCR` and `CCCR` registers to enable interrupts. However, as was already discussed, the mechanism of delivering an interrupt on performance counter overflow is imprecise, which means that by the time the operating system starts to handle the interrupt, more instructions could have been executed by the traced process. A way round this problem was implemented in the framework and it involves using single step interrupts as used in debugging when stepping through an application. Single step interrupts and the way they were used in the Echo framework is covered in more detail later in this chapter.

  The single stepping is very expensive in terms of performance overhead incurred by them. Therefore, the Echo driver uses the performance counter overflow interrupts to get through most of the executed instructions and when there is only a few more instructions left to execute for a thread, the performance interrupts are disabled and the stepping interrupts are enabled instead. The number of instructions that should be stepped through is defined by the `IMPRECISE_OFFSET` variable. Therefore, when a `next` process scheduled to run is one of the processes traced by Echo, the Echo driver checks whether the number of instructions left for the process to execute is greater than the `IMPRECISE_OFFSET`. If it is, the performance counter interrupts are enabled and the value the counter is preset with is the number of instructions executed by the thread as recorded less the value of the `IMPRECISE_OFFSET`. Otherwise a single step interrupts are enabled.

  It should also noted that an additional check is made to make sure that the `next` process that is to be scheduled to run and that is being traced by Echo is in fact the process that the Echo driver expects to run next. Since the driver does not monitor or control traced threads running in the kernel, a situation might arise that multiple traced are running in the kernel. Hence, even though the `next` thread is a traced thread, it might not be the one that is next scheduled to run according to the thread execution schedule captured at record stage. The additional check is performed by comparing the id of the `next` thread with the id of the thread that is expected to run next. This situation will not arise in user mode as the driver blocks threads that are not expected to run yet and only the expected thread to run is active.

- **`prev` process to be swapped out is traced by the Echo driver.** When the entry in the `processes` array indexed on the `prev->pid` is not `NULL`, the `prev` process to be swapped out is one of the processes traced by Echo. In this case, the Echo driver disables the performance counter interrupts and updates the remaining instructions to be executed by the `prev` process by taking away a number of instructions that the `prev` process has executed in the last quantum. Since the set of performance counters used to count down to an interrupt is different from a set of counters used to count executed instructions, the counters to count executed instructions remain active and are used the same way they were used during the record stage. It is therefore possible to obtain the number of instructions executed by a process during the last quantum by reading from the performance counter. The remaining instructions to be executed for the `prev` process is the number of instructions until it should be swapped out for another traced thread as recorded in the thread schedule captured during the record stage.

  It should be noted that the Echo driver is not concerned here with whether the `next` process is also a traced process. This is because it knows that it cannot be another traced processes since all traced processes that are not expected to run at a given time are blocked and are not going to run in user mode. The only way another traced process is scheduled to run as `next` process is if it is running in the kernel. But in this case we don't need to be concerned about setting the performance counter interrupts or stepping interrupts for it, because it will not be able to run in user mode and hence no user mode instructions will be executed. However, since there is a possibility that multiple traced threads are running in the kernel at the same time, an additional check is performed to make sure that the `prev` process is the expected process to be swapped out. This is done the same way as before by comparing the id of the `prev` process with the id of the process that is expected to be swapped out.

6.5.1 Handling counter overflow interrupts

When a performance counter overflows, an interrupt is generated, which through hooks installed in the kernel is serviced by a `ebs_handler` routine implemented in the Echo driver. The interrupt handler reads the number of instructions executed by the process between it was swapped in at the last context switch and the interrupt occurring from the performance counter. It then computes the number of instructions that remain to be executed by the current running thread. This is done by subtracting the value read from the counter from the value of the remaining instructions which gets updated during context switches when specific conditions are met, as explained above.

Since the interrupt would have caused when the thread has executed most of instructions it was supposed to run for, the calculated value of remaining instructions should be between the value of `IMPRECISE_OFFSET` and 0. This condition is checked and if the remaining instructions calculated satisfies this condition, the performance counter interrupts are disabled and the single step interrupts are enabled.

6.5.2 Single step interrupts

Before the Echo driver could start using the single steps interrupts for purposes of replaying a process, a kernel hook needs to be installed to make sure that the handler implemented in the Echo driver is called whenever the single step interrupts occur.

The kernel hook is installed using the same technique that was used to install a hook to intercept context switches. It is inserted into the `do_debug` function defined in `traps.c` file. `do_debug` is invoked whenever a given process is being debugged, e.g. with using `ptrace` utility. The handler implemented in the Echo driver is then registered with this hook when the Echo driver is loaded into the kernel.

Single step interrupts are enabled by setting a `TF` flag in the `EFLAGS` register of a traced process. The single step mode is the highest-priority debug exception and hence it is delivered to the operating system very promptly, allowing the Echo driver to stop the replaying process precisely after it has executed an expected number of instructions. Linux provides a function for setting the `TF` flag and hence to enable single step interrupts we just need to do the following:

```
set_tsk_thread_flag(current, TIF_SINGLESTEP);
```

where `current` is the pointer to the current running process.

When the single step exception occurs, the `handleStepException` handler is invoked in the Echo driver. The handler computes the number of remaining instructions to be executed, which is done the same way as in the `ebs_handler`. If this computed value is greater than 0, the process still has more instructions to step through before it should be swapped out and hence the `handleStepException` exits without doing any work. If, however, the computed value equals to 0, the process has executed all the instruction it should have according to the recorded schedule and hence it should be swapped out and the next process in the schedule should run. In this case, the `handleStepException` disables the single step interrupts by clearing the `TF` flag in the process' `EFLAGS` register, blocks the current running thread and calls `scheduleNextChild` function, which is the function implemented by the Echo driver to wake up the next thread to run according to the recorded thread schedule.

6.5.3 Blocking and resuming of threads

In order to be able to properly replay threads execution, the Echo driver has to control the traced threads when they run in the user mode. This entails allowing only one traced thread to be active at a time. Therefore, the Echo driver has to be able to block and resume traced threads.

When the traced application starts running, no child threads has yet been created by it and hence it is the only process that is active at the start of the replay. The creation of child processes requires invoking a `clone` system call and since it is handled by the Echo driver, the driver has control over the state of the child threads when they are created.

When a new thread is created by the traced parent process, the Echo driver blocks it immediately without letting it run. This is done as follows:

```
set_task_state(find_task_by_pid(child_process -> pid), TASK_UNINTERRUPTIBLE);
child_process -> block_in_kernel = BLOCK_USERSPACE;
```

`BLOCK_USERSPACE` indicates how the child thread has been blocked. The Echo driver distinguishes between two ways of blocking the child thread:

- **Blocked in user space.** When the process is stopped in user space, it is indicated to the Echo driver by setting the `BLOCK_USERSPACE` flag in the `process_state` structure of the stopped process. The process that is stopped in user space is blocked by setting it state to `TASK_UNINTERRUPTIBLE` by calling `set_tast_state` function. To resume the process, an explicit wake call call is issued to the process by calling `wake_up_process` function.

- **Blocked in kernel space.** This case is trickier. In the case of traced process, blocked in kernel means that the last instruction that was executed by the traced process was the last user space instruction before the process has entered the kernel and was switched to kernel mode. Since the Echo driver does not control the threads running in the kernel, it cannot block the thread once it has entered the kernel. However, it cannot allow the thread to leave the kernel either and start running in user space unless it is supposed to according to the recorded thread schedule. Therefore, a mechanism has to be developed which allows the threads to run in the kernel and at the same time ensure that they are blocked before they return to user space.

  To accommodate for this requirement, the Echo driver employs semaphores. Each process has a unique `userspace_lock` semaphore associated with it, which is initialized to 0 when `clone` system call is handled by the Echo driver. The Echo driver uses the `block_in_kernel` flag with `BLOCK_ENABLE` and `BLOCK_DISABLED` values to indicate whether a given thread should be prevented from returning to user space or not. To prevent a thread from returning to user space, the Echo driver must check whether the `BLOCK_ENABLE` flag is set for a given thread and if it is, the `down` operation on a `userspace_lock` semaphore corresponding to a given thread must be performed. This check and blocking of the thread on a semaphore is performed by the `checkUserSpaceLock` function implemented by the Echo driver.

  Threads return to user space on completion of a system call and hence `checkUserSpaceLock` must be called before the results of the system call are replayed. The `REPLAY_RESULT` macro is responsible for replaying results of system calls and hence it invokes `checkUserSpaceLock` before it does any work to replay the system call result.

  To unblock the thread, the `up` operation on the semaphore is performed and the `block_in_kernel` flag is set to `BLOCK_DISABLED`.

To determine whether the child thread is in the kernel or the user space, the Echo driver looks at the `eip` register of the thread and compares it with `SYSENTER_PAST_ESP`, which is the address of the point where the process is switched from the user mode to the kernel mode. If the process' `eip` equals to the `SYSENTER_PAST_ESP`, then the process has entered the kernel and hence a `BLOCK_ENABLED` flag is set to make sure that the process is blocked before it tries to return to user space. If the process' `eip` does not equal the `SYSENTER_PAST_ESP`, the process is running in user space and hence has to blocked straight away. This is done by setting its state to `TASK_UNINTERRUPTIBLE`. Figure 6.4 shows part of the `handleStepException` code, which is responsible for blocking the running traced thread.

```
if (regs -> eip == SYSENTER_PAST_ESP)
{
  printk("Context switch occurred inside kernel\n");
  CURRENT_BLOCK_IN_KRNL = BLOCK_ENABLED;
}
else
{
  printk("Context switch occurred in user space\n");
  CURRENT_BLOCK_IN_KRNL = BLOCK_USERSPACE;
  set_current_state(TASK_UNINTERRUPTIBLE);
}
```

Figure 6.4: Blocking processes

`scheduleNextChild` function is responsible for waking up the thread that should run next according to the recorded thread schedule. When this function is called, it reads the timing buffer to establish the id of the thread to be scheduled next and then goes through the list of blocked traced threads until the thread with the same id as the next scheduled thread is found. The found process is then woken up by either doing an `up` operation on a semaphore, if the thread was blocked in kernel space, or by calling `wake_up_process` routine if the thread was blocked in the user space. Figure 6.5 shows the implementation of the `scheduleNextChild` function.

6.6 Summary

This chapter has explained a mechanism that was implemented in the Echo framework for capturing and replaying the thread schedule of an arbitrary application. While the current state of the art techniques for recording and replaying thread executions have focused on software instrumentation, Echo has demonstrated a less intrusive hardware approach that uses performance counters that are readily available on most modern processor designs.

This project has successfully demonstrated the ability to record a thread schedule by hooking the context switch method in the Linux operating system that allows an opportunity to read and configure the hardware performance counters at the start and end of each quantum. Using this information, we can build up a database of the thread ordering and the duration of each run, for use during replay.

In addition to recording information about an arbitrary thread schedule, this project has successfully developed and demonstrated a new method for replaying a thread schedule through the combination of imprecise hardware performance interrupts and precise step debugging interrupts, both of which are available of modern processor designs.

```
while (current_child != NULL)
{
    if (current_child -> child_id == CURRENT_REPLAY_CHILD)
    {
      switch (current_child -> block_in_kernel)
      {
          case (BLOCK_ENABLED):
          {
             up(&(current_child -> userspace_lock));
             printk("Unblocking child to allow re-entry: %d\n", current_child -> pid);
             break;
          }
          case (BLOCK_USERSPACE):
          {
             printk("Unblocking child in userspace\n");
             wake_up_process(find_task_by_pid(current_child -> pid));
             break;
          }
      }

      current_child -> block_in_kernel = BLOCK_DISABLED;
      break;
    }

    current_child = current_child -> next_in_child_queue;
}
```

Figure 6.5: Resuming processes

Chapter 7

Debugging tools

This chapter discusses the mechanism implemented in Echo, which allows inserting debugging statements into the traced program after it has been recorded and replaying it without the need to record the program's execution again. The chapter focuses on the implementation details of this mechanism and highlights its limitations.

7.1 Motivation and overview

Debugging applications is hard. Debugging multi-threaded application is even harder due to their nondeterministic nature. The Echo framework allows multi-threaded programs to be replayed deterministically, which helps a great deal in finding errors in the multi-threaded applications. However, once the application has been recorded, it cannot be changed if it is to be replayed correctly. This is because changing the application, such as adding statements, will change the execution of the program. New system calls will be made, which were not recorded during the record stage, the number of instructions executed by each thread during its quantum will be different to that recorded previously and hence the recorded execution trace will become invalid.

This requirement of the record/replay mechanism is bad news for using the framework for debugging applications. Even though external debugging tools are available, a lot of developers still use a tried and tested method of using print statements to debug their applications. Therefore, the fact that the recorded application cannot be modified robs them of this debugging method.

Thus, a mechanism was developed within the Echo framework to allow insertion simple statements, such as print statements into the application after it has been recorded. To use this mechanism, a developer wishing to insert a statement into the application would have to define and implement a function which performs the operation a developer wishes to add to the application for debugging purposes. (From now on, we will assume that the statement to be added to the recorded program is a print statement.) A `REPLAY_DEBUGX` macro should then be inserted and invoked in a place of the desired addition of a print statement. `X` indicates how many arguments are passed to the macro. A developer should pass the name of the his debugging function and the argument to that function as arguments to the `REPLAY_DEBUG` macro. The program can then be recompiled and replayed using the previously recorded execution without having to record the application again.

7.2 Implementation

In order to be able to add statements to the recorded application without invalidating a recorded trace, we need to develop a way of letting the Echo driver know when the debugging statements start executing and when they have finished executing. If this information is available to the Echo driver, the driver can temporarily suspend replaying the results of system calls and counting executed instructions when the debugging statements start and then resume replay of system calls and counting of instructions when the debugging statements finish.

7.2.1 Stop system call

To notify the Echo driver of start and finish of debugging statements, a new system call `stop` has been implemented. The handlers for this system call are inserted in place of the handlers for `syscall_badsys` when the Echo driver is loaded into the kernel. The `stop` system call acts as a toggle between suspending and resuming replay of a recorded application. When it is called for the first time, the replay is suspended, when it is invoked again after that, the replay is resumed.

The replay of system calls is suspended by setting the mode of the current running process from `MODE_REPLAY` to `MODE_PAUSED`. The mode of the process is checked by the auxiliary functions `handleSyscallRequest` and `handleSyscallResponse`, which are called from the `syscall_intercept` and are used to dispatch to the appropriate system call handler implemented in the driver. If the mode of the current running process is `MODE_PAUSED` the auxiliary functions return without doing any work or invoking handlers for any of the system calls that are being made by the current running process. This way, when the mode of the process is set to `MODE_PAUSED` all the system call made by that process are not being handled by the Echo driver and are allowed to go through and be serviced by the operating system as normal.

To resume replay of the application, the mode of the process must be set back to `MODE_REPLAY`, which is done when the `stop` system call is invoked and the current mode of the process is `MODE_PAUSED`. Thus, `stop` toggles suspending and resuming of system calls replay.

7.2.2 Stopping and resuming performance counters

During the execution of inserted debugging statements, the performance counters must also be temporarily suspended, so that no extra instructions will be counted by them during the execution of debugging statements and hence the recorded thread schedule will be preserved. The counters are suspended and resumed during the servicing of the `stop` system call. Once the counters are suspended and the mode of a process is changed to `MODE_PAUSED`, the `recordRunningDuration` function that is invoked on every context switch ignores the current process and does not manipulate the performance counters for it.

To suspend performance counters from counting extra instructions and causing the interrupts at a wrong time, the Echo's handler for `stop` system call must disable them. It must also disable the second set of performance counters which is used to count the number of instructions executed by a process on every quantum. This is because this information is used by `recordRunningDuration` function to determine how many more instruction remain for the process to execute before it should be swapped out for another traced process. If this value is wrong, a traced thread will end up running for a different number of instructions than that recorded for it and therefore the outcome of the replayed execution will not be the same as the outcome of the original execution.

Disabling the set of performance counters that are responsible for causing an interrupt is done by a call to `disable_ebs_interrupts` routine implemented in the Echo driver. The set of performance counters that is used to count the instructions executed in the last quantum are disabled by writing 0s to its `CCCR` and `ESCR` registers. However, additional manipulation of the set of counters counting instructions executed in the last quantum is required to take into account the instructions executed when a `stop` system call was invoked.

To execute a system call, the system call number must be put into `eax` register and then a trap instruction `int $0x80` must be executed. Since the performance counters are not stopped until the system call is handled, these two additional instruction are counted when the `stop` system call is made to suspend replay. Thus, the value in the performance counter counting instructions executed during a quantum must be adjusted by subtracting two extra instructions that were executed during the replay stage and were not executed during the record stage. Figure 7.1 shows the fragment of the `handle_sysstop` routine, which deals with suspending the performance counters.

When the replay of an application is resumed, both sets of performance counters are re-enabled. The performance counters responsible for causing an interrupt are re-enabled by calling `enable_ebs_interrupts` function provided by the Echo driver. The set of performance counters counting the instructions executed in the last quantum are re-enabled by writing the configuration flags into the `ESCR` and the `CCCR` registers.

```
/* Disable interrupts to prevent race conditions with the
    context switch hook function */

 __asm__("cli");

 /* Stop the performance counter from recording user space
    instruction executions */

 wrmsrl(CCCR_ADDRESS, 0);
 wrmsrl(ESCR_ADDRESS, 0);

 /* Read the performance counter to determine the instructions
    executed in the current context, updating the replay
    statistics accordingly */

 unsigned long instructions_in_context;
 rdmsrl(READ_ADDRESS, instructions_in_context);

 instructions_in_context -= 2;

 instructions += instructions_in_context;
 CURRENT_REPLAY_INSTR -= instructions_in_context;

 /* Clear the performance counter to indicate no further
    instructions have been executed if a context switch occurs
    before the child is un-paused */

 wrmsrl(READ_ADDRESS, 0);

 /* Disable the interrupt performance counter */

 disable_ebs_interrupts();

 __asm__("sti");
```

Figure 7.1: Fragment of the implementation of handle_sysstop function

```
#define REPLAY_DEBUG3(FUNCTION, ARG1, ARG2, ARG3)
1. __asm__("movl $284, %eax; int $0x80");
2. __asm__("pushl %0" :: "m"(ARG3));
3. __asm__("pushl %0" :: "m"(ARG2));
4. __asm__("pushl %0" :: "m"(ARG1));
5. __asm__("call " #FUNCTION);
6. __asm__("add $12, %esp");
7. __asm__("movl $284, %eax; int $0x80");
```

Figure 7.2: REPLAY_DEBUG macro

7.2.3 Replay debug macro

The `REPLAY_DEBUG` macro is used to make sure that the system call replay and performance counters are suspended and resumed as required and to make sure that insertion of an additional print statement leaves the original program as undisturbed as can possibly be.

Figure 7.2 shows the code of `REPLAY_DEBUG3`, that is a `REPLAY_DEBUG` macro, which can takes 3 arguments and a name of a debugging function implemented to perform desired debugging operations.

The `REPLAY_DEBUG` is implemented as a sequence of preprocessor directives. In line 1, `REPLAY_DEBUG` makes a `stop` system call notifying the Echo driver that the next few instructions are debugging instructions inserted into the recorded application and hence they should not be replayed. It then pushes the arguments to the debug function implemented by the developer onto the stack in lines 2 – 4 and then calls the debugging function in line 5. When the debugging function returns, the `REPLAY_DEBUG` macro cleans up the stack in line 6 and then in line 7 it makes `stop` system call again to notify the Echo driver that the debugging statements have now been executed and so the replay of the recorded application should be resumed.

7.3 Limitations

Although the described technique allows to insert statements into the recorded application without having to re-record it, it has a number of limitations:

- **No new temporary variables can be created and used in the recorded application.** Creation of new temporary variables will result in the compiler producing extra instructions that Echo will not be able to account for. Also the layout of the execution might change, for example variables being stored in different registers, which could lead to a different number of instruction requiring to be executed after recompilation. Therefore, only variables that have already been defined and used in the recorded program and statically allocated data can be passed to the user-defined debug function via the `REPLAY_DEBUG` macro.

- **The application cannot be recompiled using an optimizing compiler.** An optimizing compiler might change the layout of the recorded application, which might make the captured execution trace invalid.

7.4 Summary

By recording and replaying thread schedules using the performance counter technique described in the previous chapter, it is possible to add additional instructions into the replay binary assuming the additional instructions are not counted as part of the schedule when executed. In order to ignore the additional instructions, we have introduced an additional system call into the Linux kernel to signal to the kernel module when a thread is entering and exiting a block of code that was not in the original binary and therefore not recorded in the original schedule. We have successfully demonstrated that a developer can use this mechanism to replay a previous execution of their application with the source code instrumented with debugging statements provided by the Echo project.

Chapter 8

Testing and Evaluation

This chapter discusses the testing strategies that were employed to test the correctness of the framework's functionality. It also presents preliminary experimental results of the framework's performance.

8.1 Testing

A number of test applications have been used to assess the correctness of the Echo's record and replay mechanisms.

Linux `ls` utility

`ls` utility, which lists the contents of the current working directory was used to test the correctness of recording and replaying single threaded applications. The tests performed involved:

- Running `ls` natively and noting the results returned by it. The `ls` is then invoked with the Echo framework in record mode in the same working directory. The results returned by the `ls` operating under the Echo framework are noted and compared with the original results returned by `ls` running normally. The compared results were identical, which indicates that the record mechanism is functioning properly.
- `ls` was invoked a number of times under the framework in record mode to test that the process execution can be recorded a number of times without compromising the integrity of the system. The results of each record run were noted and on comparison with the original native run were found to be identical. The `ls` utility was executed natively again to make sure that the integrity of the system has not been compromised by the Echo framework. The result returned was identical to the results obtained on all the previous record runs and also to the result obtained from the first native execution of the utility.
- To test the correctness of the replay the `ls` utility is executed under the Echo's framework in replay mode. The log files required for the replay of execution were generated already during the testing of correctness of the record mechanism and are used in this test. The results of invoking `ls` in the

replay mode are recorded and compared with the results obtained during the native execution of `ls`. They were found to be the same. A new file is then created in the current working directory and the `ls` utility was invoked natively again. The results obtained reflect the fact that a new file was created. `ls` is then invoked again in replay mode. The results obtained do not show the new file that has been created after the application execution was recorded, which is the desired result. `ls` is then invoked again in record mode, followed by running it in replay mode. The results obtained now show the new file being created.

- Final set of tests performed using the `ls` utility involve invoking the utility in record and replay mode a number of times and then invoking it natively to make sure that the operation of the Echo framework does not compromise the integrity of the operating system.

fctest

This test was written to test the ability of the framework to trace multiple processes, to confirm the correctness of the record and replay mechanisms and to test the correctness of handling some important system calls, such as `read` and `clone`.

`fctest` performs the following operations:

1. It takes a filename of a file to read from as an arguments.
2. It opens the specified file for reading.
3. It reads the first 3 bytes from the file and prints out what it has read.
4. It then creates a new process using `fork`.
5. The child process proceed by reading the next 3 bytes from the file printing them out.
6. The parent process proceeds by trying to read 3 bytes from the `STDIN`.
7. When the input that the parent process is waiting for becomes available, the parent process print it out to the `STDOUT` and terminates.

The following tests have been performed using `fctest`:

- `fctest` was invoked natively and using the Echo framework in record mode. The results of both executions were compared and were found to be the same. This test was repeated a number of times to make sure that the correct operation of the system continues after the framework has been employed and also to make sure that the framework can cope with recording application's execution multiple times.
- `fctest` was invoked in replay mode to test whether a successful replay execution can be obtained. This test has highlighted a few subtleties about the replay mechanism that were not at first considered but which were subsequently fixed. These subtleties include:

  - Possibility of system call buffer race conditions as multiple threads are allowed to run in the kernel at a time. This problem was fixed by introducing semaphores to protect access to the system call buffer and thread suspension when an out of order access to the system call buffer was attempted during the replay stage.
  - Imprecise delivery of the performance counters overflow interrupts. This problem was fixed by using single step interrupts available as part of the debugging facilities supplied by the operating system.

  Once the measures were taken to fix the problems with the replay mechanism, the results obtained from running `fctest` in replay mode were identical to the results obtained from the previous test.

- The contents of the input file were modified and the `fctest` invoked in replay mode. The results did not reflect the fact that the contents of the input file were modified, which is the desired result. The test proceeded by invoking the application in record mode followed by replay mode. On this run, the results obtained from the replay execution reflected the fact that the input file was modified, which is the desired result, which supports the claim that the framework's record and replay mechanisms are working correctly.

threaded

This test was designed to test the ability of the Echo framework to replay a multi-threaded application. This test also tested the ability of the framework to deal with a large number of threads interacting with one another and its ability to replay race conditions occurring in the application accurately.

`threaded` is a multi-threaded program, which works as follows:

1. It takes as an argument an integer, representing the number of threads that should be created.
2. It then opens a predefined data file in read mode.
3. It then creates as many threads as was specified by the argument to the program.
4. The threads start running and each thread performs the following steps of a `for` loop:
   (a) It reads one byte from the opened file into a buffer.
   (b) It prints out its id.
   (c) It then updates the `total`, which is a variable that is shared amongst all threads. The update includes multiplying the current value stored in `total` by 3, add the integer value of the character that it read from the file and add a number, representing its id.
   (d) It then sleeps for a period of time. This period is influenced by the value of the `total`.
5. Once all threads have gone around the loop a specified number of times, the parent process prints out the value of the `total` and exits.

Clearly, this program suffers from race conditions since the access to the shared variable is not protected in any way. In fact, the value of the `total` is different on every run of the program, which makes it a good candidate for testing correctness of Echo's replay of multi-threaded applications and ability to capture race conditions. The following tests were performed using `threaded` program:

- `threaded` was invoked under the framework in record mode and the result of the execution was noted. The program was then invoked in the replay mode and the result of the replayed execution was compared with the one obtained during the record stage. They were found to be the same. The program was executed in replay mode again for a number of times. The result of all the replay executions was identical to the result of the first replay execution and hence to the result of the record execution.
- The program was invoked in record mode a number of times to make sure that invoking the program under the control of the framework in record mode results in different output results every time like the original execution. Indeed, every recorded execution resulted in different results. This suggests that the framework fulfills its promise of retaining such bugs as race conditions and thus making a useful debugging tool.
- The number of threads to be created by the `threaded` program was varied between as few as 3 and as many as 10. The previous tests of recording and replaying the program a number of times and comparing the results were performed. The results were all as expected, which provided another strong evidence for the correctness of the Echo's record and replay mechanisms.

Inserting debugging statements into the recorded application

A few tests have been performed to test whether the mechanism developed in Echo of allowing to insert debugging statements into a recorded application and then replaying it without having to re-record the application works:

- `fctest` program was executed under the Echo framework in record mode. Then, a following debug function has been implemented for `fctest` program:

  ```
  void doDebugging(int fd, char* xs, int read_bytes, int argc, char** argv)
  {
    printf("Argc = %d\n", argc);
    int i;
    for (i = 0; i < argc; i++)
      printf("argv[%d] = %s\n", i, argv[i]);
  }
  ```

  A `REPLAY_DEBUG5` macro has been inserted into the `fctest` before line in which the parent application forks off a new process. `fctest` was then recompiled and replayed using the record data obtained earlier. The results were as expected with the debugging statements printed out in the right place of the program's execution.
- A similar test was conducted using `threaded` program. In this case, the debugging function was implemented to print out the value of the `total` on every iteration of a `for` loop after the value has been updated by one of the running threads. The program was the recompiled and replayed using the record data obtained earlier. The results were as expected, with the value of `total` printed on every iteration of the loop. The final value of `total` was also as expected, identical to the value obtained during the record stage.

In both cases, the test programs were then changed to their original code, recompiled and replayed again. As expected, the debugging statements did not feature in the output of the programs anymore and the outcomes of the programs' executions were identical to the results produced by these programs before the debugging statements were inserted.

8.2 Evaluation

We now turn to some preliminary experimental results of the performance of the Echo's framework. The main focus of these tests is the performance overhead incurred during the recording stage in the operation of the framework.

The applications used for these experimental results include both CPU and I/O bound applications.

8.2.1 CPU bounded applications

Gzip

Gzip is a famous compression algorithm and part of the SPEC2000 benchmarks suite. SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory [28]. Gzip thus represents a good example of a CPU bounded application.

In this experiment, a file of varying size was given to the gzip application to compress. Figure 8.1 shows a graph of the time taken to compress an input file by the original gzip application and also by the gzip application running under the framework in record mode.

The x-axis shows the size of the input files in KB and the y-axis shows the time taken to compress an input file in 0.01 seconds. The overhead incurred by Echo is small, which is an expected result. This is because the number of system calls performed by a CPU bounded application tends to be small and hence:

- The less time is spent in the system, the less intervention from the framework is experienced by the application. This results in less overhead incurred by Echo
- The amount of execution data to be logged is small. This results in the Echo driver having to do less writes of the buffer to the disk, which leads to less overhead.

Go, ijpeg and Whetstone

The trend of low overhead exerted by the Echo framework on the CPU bounded applications continues when the Go, ijped and Whetstone benchmarks were used in the evaluation experiments.

Go is part of a SPEC95 benchmark suite and it is used to test integer performance. ijpeg is also part of the SPEC95 benchmark suite and it is described as "compression/decompression of in-memory images based on the JPEG facilities. The benchmark application performs a series of compressions at

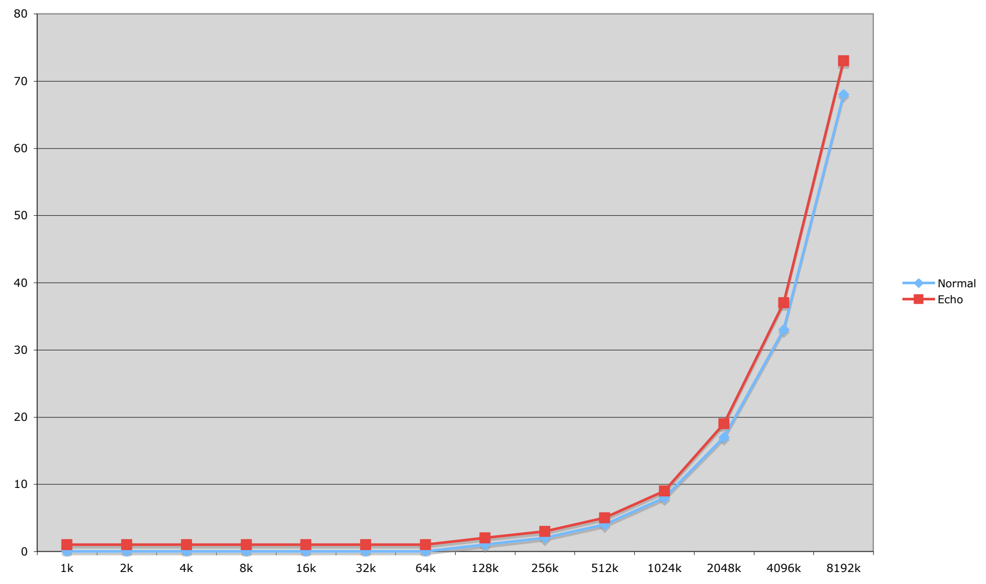


Figure 8.1: Overhead of recording execution of Gzip using Echo framework

different quality levels" [28]. The common utility implementing an algorithm has been modified, similarly to Gzip, to perform compression using memory buffers rather than reading/writing files. Whetstone is a benchmark test which attempts to measure the speed and efficiency at which a computer performs floating-point operations. It contains several modules that represent a mix of operations typically performed in scientific applications, such as `cos`, `sin`, `sqrt`, `exp` and others [25].

These benchmarks were executed on their own and then under the Echo framework. The results can be seen in Figure 8.2.

The results of the performance overhead of the CPU bounded applications are very promising and show that Echo can be employed to record and replay computationally intensive applications as the overhead of running these applications under the framework incurs a negligible overhead.

Benchmark	Normal	Echo	Overhead
Go	275	276	1.003636364
ijpeg	25	27	1.08
Whetstone	1248	1249	1.000801282

Figure 8.2: Overhead of recording execution of Go, ijpeg and Whetstone benchmarks

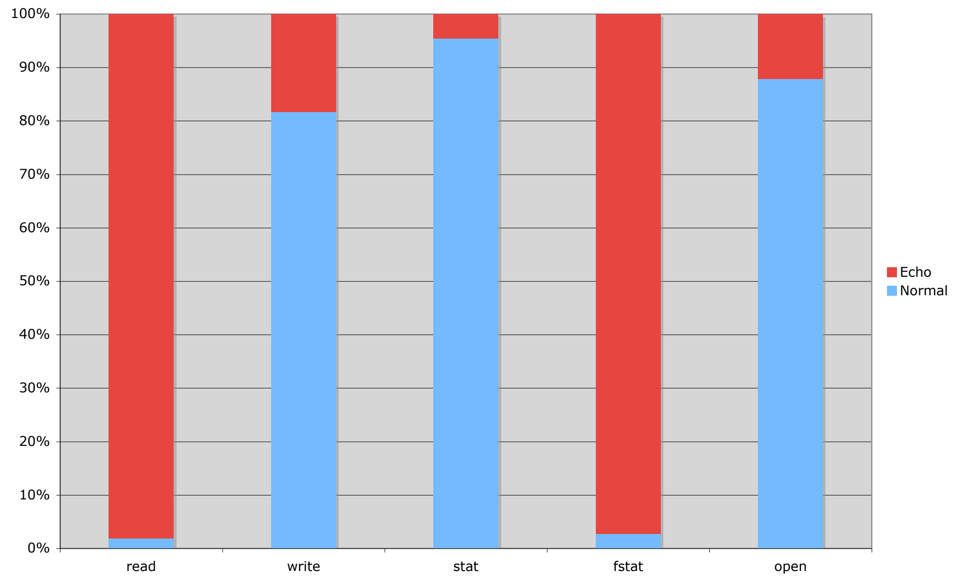


Figure 8.3: LM system call latency overhead

8.2.2 I/O bounded applications

Unfortunately, the results of performance overhead of recording I/O bounded applications with Echo framework is very large. A number of tests have been conducted to investigate performance overheads incurred by Echo and possible reasons for these overheads.

LMBench [26]

The system call latency has been investigated using the LMBench suite of benchmarks. The system call latency benchmark repeatedly performs a single type of request of the operating system (e.g. read; write; etc) to calculate the average duration of time the system requires to complete the request. This benchmark was used to determine the maximum cost that might be incurred by an application running under Echo, a cost that would be approached as an application increases the percentage of time it spends performing input / output operations.

Figure 8.3 depicts the results that were obtained through analysis using this benchmarking suite. As expected the read system call incurs the greatest performance hit, by a factor of 50, since all data read from the disk is copied to a temporary vmalloc'd buffer, which in turn is copied to the Echo system call buffer, delaying the read system call return. A similar situation occurs with fstat and would occur with stat, although at the time of writing this report, the stat system call is not currently handled and hence approaches native speed. Both the open and write system calls incur a minimal overhead since in both cases only the return value is stored in the system call buffer.

It is clear from the LMBench latency benchmark that a significant bottleneck exists within Echo when handling system calls that write data to a buffer in user space. This bottleneck could be significantly reduced by using a linked list of data buffers as opposed to the current system where data is copied from user space into a temporary buffer before being written in the main system call buffer.

fread performance

In order to better understand the bottleneck affecting read system calls, identified by LMBench, a new test was developed to read a 500KB file in a decreasing number of system calls. As the size of the read requests increase, fewer system calls are required to read the full contents of the test file. Since fewer system call requests are performed, the overhead associated with entering the system, checking the read arguments, reading the data, copying the data to a user space buffer and so forth is reduced, as demonstrated in Figure 8.4 (”Normal”). Using the typical 60K buffer size for the Echo system call log (”Small” in Figure 8.4), the buffer is filled and written many times before the file is fully read and therefore no performance gain can be realised by increasing the read chunk size. Using an infinite sized buffer for this test (”Large” in Figure 8.4), a performance gain is realised and the graph resembles the native performance gain although with the system call overhead experienced when running applications under Echo. Figure 8.5 shows the overhead experienced using the fore-mentioned read benchmark. As the rate at which data is requested from the system increases, the rate at which Echo must write out system call data increases, causing a bottleneck that requires the process generating the data to be blocked. This in turn reduces the amount of parallelism available and hence increases the overhead incurred by the process. Using a larger buffer size greatly reduces the incurred overhead.

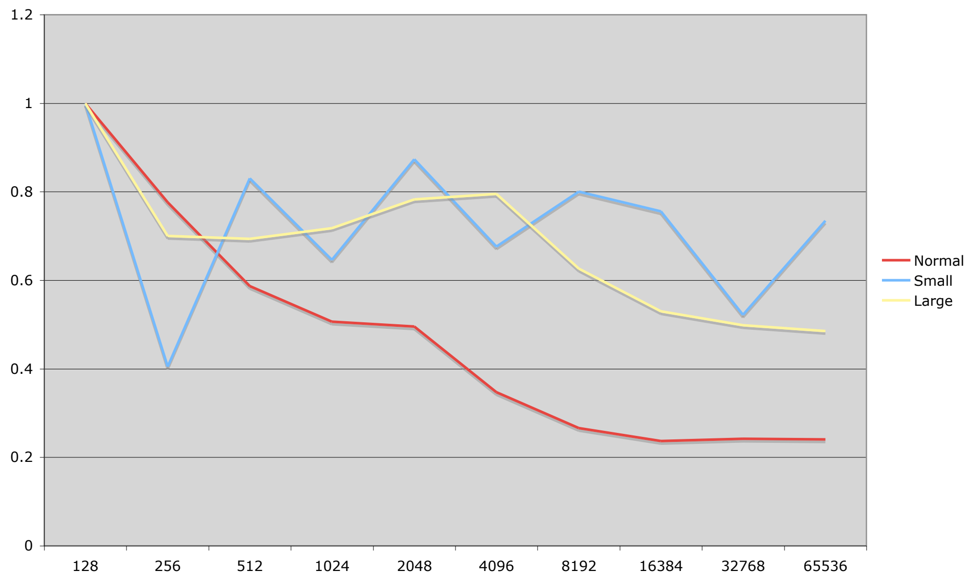


Figure 8.4: fread performance overhead

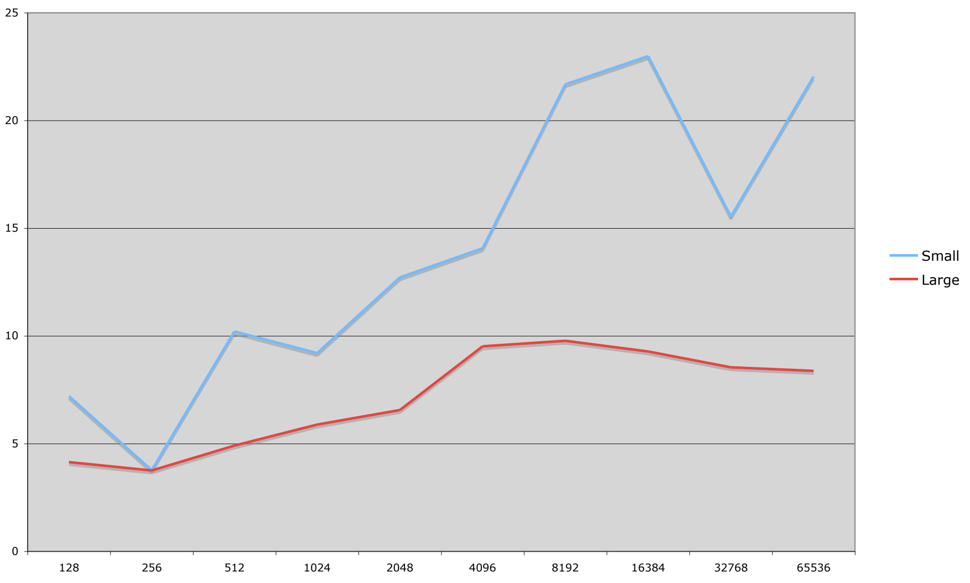


Figure 8.5: fread overhead experienced with different size buffers

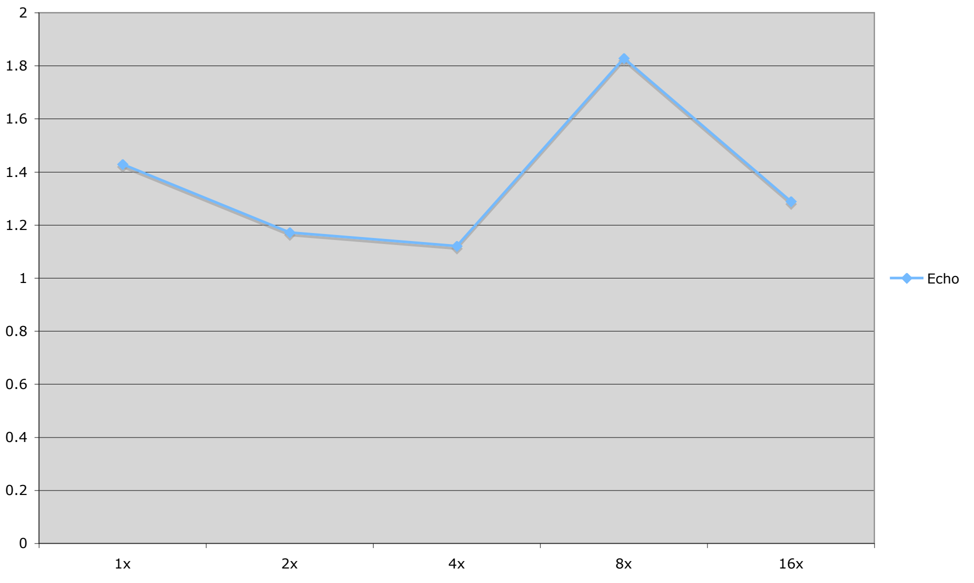


Figure 8.6: Overhead of recording execution of G++ using Echo framework

G++

The previous two tests have helped to identify the overheads associated with system call latency. In order to determine how the increase in system call latency affects regular applications with more balanced CPU and IO time, the impact of Echo on the C++ compiler, G++, was measured. Figure 8.6 measures the overhead incurred when compiling increasingly larger bodies of code starting from a sub-second compilation time (1x) up to a piece of code that natively incurs a 16 second compilation time (16x). Since G++ performs a significant amount of processing, the overhead incurred during the total execution is far lower than the system call overhead measured by LMBench, ranging from 10% to slightly over 80%. As expected, the results indicate that the overhead incurred by G++ is dependent on the program being compiled as opposed to the duration of time that is required to natively compile the code.

8.3 Limitations

The evaluation results discussed in this chapter suffer from a number of limitations, which stop them from being truly representative:

- The Echo framework currently only supports a subset of the system calls used in Linux. This is a limitation of the current evaluation due to the fact that the potential overhead of handing the system calls that are not currently handled by the Echo driver is not taken into account in this evaluation.

- The Echo framework currently does not support recording or replaying of signals. This limitation of the Echo driver makes it impossible to record and replay certain types of applications, for example graphical applications and hence they cannot be included in the evaluation tests.
- The experiments were performed on only one particular platform. All the testing and evaluation have only been performed on a Pentium 4 machine running Linux operating system.
- A more comprehensive set of test data and a larger number of experiments is required to be able to see the real picture.

8.4 Summary

In this chapter we have evaluated the performance impact of Echo on CPU and I/O bounded applications. As expected, applications running on Echo experience an overhead that is proportional to the amount of time they spend performing I/O requests. For CPU bounded applications like GZiP the overhead is negligible, whilst applications that read from files or network sockets incur a very large performance overhead. Applications that use a balance of CPU and I/O time, such as G++, incur a fairly substantial overhead that would need to be addressed before a framework such as Echo could become widely used as a convenient and effective debugging tool.

Although the performance overhead recorded in this chapter is significant, recent code reviews have shown great potential for optimisations and we estimate that the overhead for data intensive system calls such as read could be reduced to less than 200%.

Chapter 9

Conclusions

9.1 Achievements

This report has presented a documentation of a novel approach in recording and replaying execution of nondeterministic applications and an implementation of this approach, which resulted in the development of the Echo framework. The correctness of the framework in recording and replaying multi-threaded and single-threaded applications was tested and the results of the tests were discussed in this report. Preliminary performance tests were also performed on the framework and their results are discussed and analyzed in this report.

The key achievements of this project are:

- A novel approach for recording and replaying multi-threaded applications have been developed. One of the main objectives of this project was to investigate whether a performance monitoring hardware can be used to capture and replay a particular thread schedule. The technique for capturing and replaying a particular thread schedule developed as a result of this investigation indeed relies on performance counters as a main mechanism for collecting accurate information about the number of instructions executed by a process between context switches. The approach also depends on the ability of performance counters to generate interrupts on overflow, which is used to force the application to stick to a recorded thread schedule during the replay stage of the execution.
- A framework that uses the developed approach has been implemented. The Echo framework has been implemented as a result of the project. The Echo framework supports recording and replaying nondeterministic multi-threaded applications using the performance counters method which was developed during the project. The Echo framework offers an added advantage that, unlike some other projects in this field, it does not instrument the original application or the shared libraries, not does it require the application to be recorded or replayed to be written using specific API.
- A mechanism was developed that allows inserting debugging statements into the recorded application and replaying it without the need to re-record the application's execution. The project was motivated by the benefits that the creation of the deterministic record/replay framework would offer to the tedious and difficult task of debugging multi-threaded applications and therefore part of the project was to investigate whether it would be possible to add statements into the recorded application and

then replay it without having to re-record it. This investigation has lead to the development of a mechanism, which indeed makes it possible, albeit for a limited purpose.

- The correctness of the framework's operations has been tested and the results obtained were very positive. The soundness of the record/replay mechanism and the mechanism for debugging recorded applications has been tested using a small subset of applications. The results obtained form a strong evidence to suggest that the Echo framework developed during this project operates correctly and that one of the main goals of the project – creation of a record/replay framework for multi-threaded applications was achieved.

- The performance of the framework was evaluated and results for the incurred overhead were obtained. The evaluation of the Echo's performance show that there is a very small overhead that is being incurred on the computationally intensive CPU bounded application. On the other hand, the overhead of recording I/O bounded application is great, especially for a `read` system call. However, since the Echo framework was not implemented with a focus on optimizing performance, these preliminary results should be taken negatively and in fact, performance of the framework should be largely considered as a topic for future work.

9.2 Future work

There is a lot of scope for future work. A few of the possible improvements and extensions to the Echo framework include:

- Addition of the support for signals. Currently Echo does not provide support for signals. Extending the framework to provide support for signals will increase the number of applications that could be recorded and replayed using Echo, which will make Echo into a more rounded system and possibly lead to it being used in the industry.

- Increasing the set of the handled system calls. Currently, Echo handles a small subset of system calls available under the Linux operating system. Adding support for a larger number of system calls will increase the number of applications that can use the framework, which in turn can be used to explore new areas for applicability of the Echo project.

- Introducing mechanisms such as checkpointing to aid debugging even further and also to reduce the size of the log files kept. Checkpointing would be a useful addition to the Echo framework as it could allow record/replay only a part of the application's execution. This might have a benefit of decreasing the size of log files used to keep the execution data, however it might also have an adverse effect on performance as checkpointing is a costly operation. An interesting investigation into the trade offs of checkpointing can be conducted.

- Optimizing the performance of the framework. An interesting investigation into the overheads of the framework's performance can be conducted. Once the reasons for those overheads are established, various optimization techniques can be used or developed to try and reduce the overheads.

Bibliography

[1005] *IA-32 Intel Architecture Software Developers Manual. Volume 3: System Programming Guide*, September 2005.

[12] Brink and abyss, pentium 4 performance counter tools for linux.
`http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm#References`.

[13] Digital hermit – unix and linux solutions.
`http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html`.

[14] The linux kernel programming guide.
`http://www.linuxvoodoo.com/resources/guides/lkmpg/`.

[16] Linux device drivers (an online book).
`http://www.xml.com/ldd/chapter/book/`.

[1801] *Modern Operating Systems*. Prentice Hall Interantional, 2001.

[20] Gcc-inline-assembly-howto.
`http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[21] Using assembly language in linux.
`http://asm.sourceforge.net/articles/linasm.html`.

[22] Cross-referencing linux.
`http://lxr.linux.no/`.

[23] Kernel korner - sleeping in the kernel.
`http://www.linuxjournal.com/article/8144`.

[24] Kernel locking techniques.
`http://www.linuxjournal.com/article/5833`.

[25] Whetstone benchmarks.
`http://www.keil.com/benchmks/whets.htm`.

[26] Lmbench - tools for performance analysis.
`http://www.bitmover.com/lmbench/`.

[27] Kernel development.
`http://lwn.net/Articles/2853/`.

[28] Standard performance evaluation corporation.
`http://www.spec.org/`.

[Bau03] Marcel Christian Baur. Instrumenting java bytecode to replay execution traces of multithreaded programs, 2003. Diploma Thhesis, Swiss Federal Institute of Technology Zurich.

[CS98] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. 1998. Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon, August 1998.

[MR03] M. Christianens et al. M. Ronsse, Koen De Bosschere. Record/replay for nondeterministic program executions. volume 46, pages pp. 62 – 67, September 2003.

[MX03] Mark D. Hill Min Xu, Rastislav Bodik. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. June 2003.

[RC96] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, 1996.

[SE96] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. pages pp. 250 – 261, June 1996. In Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing.

[SM99] Richard Stones and Neil Matthew. *Beginning Linux Programming.* Wrox Press Ltd., 1999.

[SMSZ04] C.R. Andrews S. M. Srinivasan, S. Kandula and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. June 2004. In USENIX Annual Tech. Conf.

[SNC05] G. Pokam S. Narayanasamy and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. 2005. In Proceedings of the International Symposium on Computer Architecture, June 2005.

[Spr02a] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):pp. 64 – 71, July/August 2002.

[Spr02b] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):pp. 72 – 82, July/August 2002.

[Wit88] Larry D. Wittie. Debugging distributed c programs by real time replay. pages pp. 57 – 67, May 1988. In ACM workshop on parallel and distributed debugging.

[Yas05] Saito Yasushi. Jockey: A user-space library for record-replay debugging. Technical report, HP Labs, 2005.