Imitate: Recording multi-threaded programs for trace analysis and deterministic replay

Final Report

Vishal Mistry (vm03)

Supervisor: Dr. Paul Kelly
Second Marker: Dr. Tony Field
Department of Computing, Imperial College London

June, 2007

Abstract

As multi-threaded applications become more common due the performance advantage they provide, so do the race conditions that manifest themselves as a result of the interleaving of threads.

This project presents Imitate, a debugging platform that enables the recording of executions of multi-threaded applications so that they can be debugged later by deterministic replay and trace analysis. It is implemented as a kernel module for the Linux operating system and generates these traces by intercepting thread pre-emptions and system calls.

This report documents the implementation of Imitate and discusses its performance in relation to its goals.

Acknowledgements

I would like to thank my supervisor, Dr. Paul Kelly, for all the support, advice and inspiration he as given me throughout the duration of this project. Without him, this project would not have been possible.

I would also like take this opportunity to thank my second marker, Dr. Tony Field, for his help and enthusiasm in completing this project.

A special thank you also goes out to my tutors, Prof. Susan Eisenbach and Prof. Peter Harrison, for all their help and support through out all my time at university. Without them, I would not have made it through even the first year.

Contents

Chapter 1

Introduction

1.1 Motivation

Processor speeds have stagnated. Developers no longer get the "free" speed increases in their software every year and as a result, many have turned to multi-threading to improve the performance of their software. However, as many are finding out, debugging errors due to multi-threading is near impossible due to the need to re-create the exact conditions under which the bug occurred for the operating system as well as the application. This is not trivial.

Debugging is generally viewed as a difficult task, and often considered more difficult than actually writing the code. Errors that occur in production are dependent on many factors such as the actions of users (past and present), file system state and data received over the network. Reproduction of these factors is usually required to debug an application, but most scenarios do not allow for the collection of such data for various reasons. To debug multi-threaded applications, the thread schedule is also needed as the interleaving of threads can cause race conditions which only show themselves under very specific circumstances. There are tools such as SystemTap[1] which are able to capture the schedule and other information however, they do not have the ability recreate it when debugging.

This project is about the creation of a *portable* low-overhead framework and associated tools which will allow the recording of executions of applications (both single- and multi-threaded) such that they can be deterministically replayed in the future. The purpose of such a framework is to allow for the recording of applications in production environments so that errors resulting from non-determinism due to data races, incorrect synchronisation, and other concurrency issues, can be reproduced and debugged.

In addition to portability and debugging by replay, one of the other main goals is to allow the traces generated by the framework to be easily amenable to debugging by trace analysis as this can enable debugging to be automated. The framework can also have other uses, such as data recovery in the event of corruption and for attack analysis in the event of a security violation in the application being recorded. However, they were not of primary concern for the implementation provided.

There have been many record/replay solutions created previous to this project, but have all suffered from limitations, ranging from requiring dedicated hardware to not being able to capture thread schedules, that prevent them from being used to record applications in production environments. Most, if not all, of these previous solutions are also very difficult to install. These short comings of previous projects have contributed greatly in the decision to undertake this project.

[1] http://sourceware.org/systemtap

1.2 Approach

Imitate is implemented as a kernel extension/module along with user-space monitors that interact with the module to save and load traces. Both the kernel module and monitors behave in much the same way as that provided by the Echo [17] framework, however their implementation has been structured in such a way as to allow for porting across architectures by separating the architecture-dependent and architecture-independent sections.

In addition to this, unlike the Echo framework, Imitate does not rely on hardware performance counters and instead uses code instrumentation and the technique proposed by Russinovich and Cogswell [26] to assist in capturing the schedule. The reason for this decision is that in addition to reducing the amount of hardware dependent code, it also fulfils the aim of making the traces useful for trace analysis as the instruction pointer is recorded as part of the schedule.

The decision to implement a kernel module as part of this project was taken to ease the interception and capture of system calls and the thread schedule. Recording both the system call data and the thread schedule is important for generating accurate traces as both of these contribute to the non-determinism experienced by an application. System calls are the main cause of non-determinism in single-threaded programs while the thread schedule is most important for multi-threaded programs.

1.3 Achievements and Contributions

There has been a lot of work carried out in the area of deterministic replay, however most projects have focused only on being able to replay only one cause of the non-determinism. Breaking this trend, this project attempted to allow for the replay of non-determinism from the thread schedule and from system calls.

Portability has also been an issue for past work. Most previous solutions have required the use of particular programming languages and API's in order to allow capturing of execution, and those that have not have required dedicated hardware.

The main contributions of this project are as follows:

- **A *portable* solution for the recording the execution of an application**. This is the largest contribution of this project. Imitate has provided an implementation of the recording of both system call data and the thread schedule in a portable way. Its portability has been assured by making sure that the architecture dependent and independent portions of code are loosely coupled and use only standard functions available in all modern hardware. The function of special hardware support that is usually required has been replace by support from program instrumentation.
- **An implementation for recording a thread schedule trace that is easily amenable to trace analysis and replay**. Imitate creates a thread schedule trace that can be analysed by tools as each entry in the trace stores the instruction pointer when a thread was swapped out and the number of backward control transfers made between the point it was swapped in and the point it was swapped out. This has enabled providing a way to pinpoint pre-emption points of threads without resorting to counting instructions in each quantum, which usually results in traces that difficult and slow to analyse without replay.
- **An implementation for recording an execution trace for an application that does not use a specific API**. Imitate can be used to record all applications regardless of the programming language they are written in and the API's they use. This is due the interception of non-deterministic events in the kernel and not in user-space.

Chapter 2

Background

This chapter presents an investigation into the issues behind non-determinism as a result of multi-threading and the approaches being employed to find and correct them. It also covers the issues relating to the area of deterministic replay.

2.1 Multi-threading and non-determinism

Traditional processes consist of only one thread of execution, and thus are mainly sequential in execution in most cases. Non-determinism can be introduced into a single-threaded process in the form of user input and non-deterministic functions such as `gettimeofday`. These usually do not cause many problems unless timing is important or logic errors exist within the code. Other non-deterministic events include context switches, signals, interrupts and network activity. Out a these, signals are most likely to cause bugs in a single-threaded program, as they require the program to jump to a different section of code as a result of the event.

Unlike traditional processes, multi-threaded processes contain many threads of execution which can all be accessing and modifying the same memory at the same time. Hence, non-determinism can manifest itself in many unexpected places due to interleaved memory accesses[1], including code that may be

[1]For multiprocessors, memory accesses will be interleaved by the memory controller, even though they may have been made at the same time by the processors on the system.

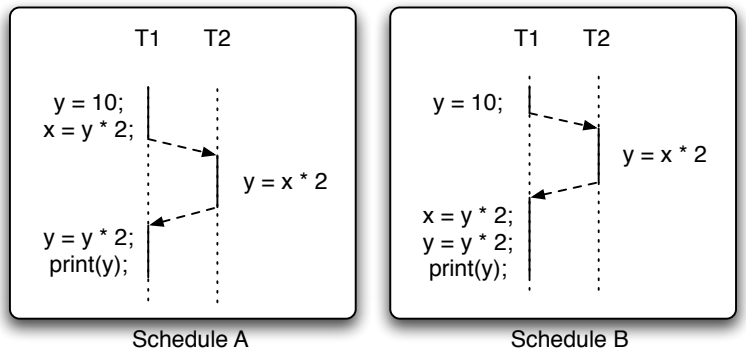

Figure 2.1: Thread interleaving. Schedule A will print `80`, schedule B will print `0`.

correct in a single-threaded program. On a uniprocessor system, memory can be accessed "at the same time" by different threads due to interleaving of their execution by the scheduler (see Figure 2.1). For multi-processor systems, multiple threads can physically be running on the system, one on each processor, and hence can all be accessing memory. Errors caused by the arbitrary interleaving of memory accesses are called *interference*.

To avoid interference, locking/synchronisation can be employed to make transactions, such as the one in Figure 2.1, atomic. However locking is not easy to implement even in the simplest of scenarios, regardless of the amount of experience in software development. As a result, problems related to locking such as the following are common:

Deadlock Occurs when threads need multiple locks, but attempt to obtain them in different order, thus causing the threads to block with no hope of recovery. For example:

- Thread A holds lock A and thread B holds lock B.
- Thread A tries to acquire lock B which is being held by thread B. Failing to obtain it, thread A blocks.
- Thread B tries to acquire lock A which is being held by thread A. Failing to obtain it, thread B blocks.
- Neither thread can now progress since the locks they need are held by the other.

Livelock/Resource starvation Occurs when individual threads can progress, but the system as a whole cannot. For example, in a system there are two threads, one that writes to disk and another that reads the written data. The writing thread has priority and the data being written is continuous. In this scenario, the reading thread would never get a chance to execute. Therefore, while the writing thread can progress, the whole system cannot.

Re-entrancy A function that cannot be executed concurrently by two or more threads is not re-entrant. Typically, this means that the function accesses some static or global data and hence, execution of this function by more than one thread at a time is likely to corrupt data.

Errors related to locking and memory corruption due to lack of synchronisation are particularly difficult to find and debug due to the fact that they only present themselves in very specific situations. Bugs such as these may manifest themselves for very long periods of time ranging from a few hours to days, and therefore are often impossible to reproduce, making debugging difficult.

2.2 User-level and Kernel-level multi-threading

Multi-threading can be implemented in two ways (1) user-level and (2) kernel-level. Most modern operating systems such as Linux and Windows have support for multi-threaded processes within the kernel, and hence it is more popular than user-level solutions. Kernel-level multi-threading also has a significant performance advantage, while reducing code complexity.

In kernel-level multi-threading the operating system has support for multi-threaded processes. This means that no additional work has to be done by the developer to facilitate multi-threaded applications (such as implementing a scheduler, managing thread stacks, etc.) as it is all handled by the operating system. All that developer needs to do is to use the system calls provided (typically via some threading library, e.g. `pthread`) to manage threads within their application.

In user-level multi-threading, the support for multi-threading is implemented in user-space (see Figure 2.2) as part of the program. Typically, this scheme is only used when the operating system does not support multi-threaded processes. Hence, the operating system does not know not know about threads and plays no part in handling the them. Due to lack of operating system support, creating applications

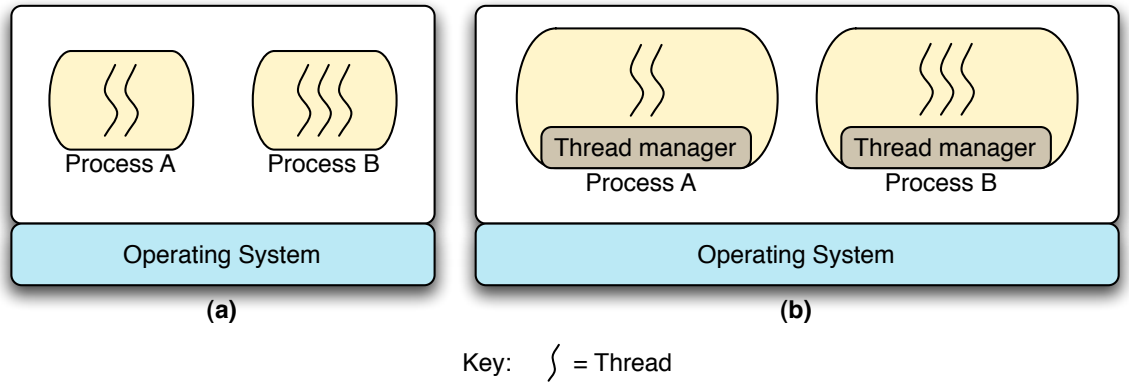

Figure 2.2: Difference between (a) kernel-level and (b) user-level threading. In kernel-level threading, thread support is provided by the operating system. In user-level threading, thread support is provided by the thread manager within the process.

using this scheme is significantly more complicated. The developer must often go to great lengths to implement operations such as non-blocking I/O since issuing a blocking system call from any thread will cause the process to to be blocked. Therefore, the other threads will also be blocked.

2.3 Model Checking

Currently, one of the most successful techniques for detecting bugs in concurrent programs is to use model checking. This is a technique by which the concurrent behaviour of the program is idealised so it can be checked for problems. Examples of modelling languages include FSP (Finite State Processes), CCS and the various π-calculi.

Labelled Transition System Analyser [21](LSTA) is one of the simplest of the modelling systems and uses the FSP process algebra. The idea behind this system is that each process within a complex system can be modelled as a finite state automata. The combination of the processes running in parallel and interacting produce the the complex behaviour of the system.

LTSA provides a simple language to model the individual processes and a composition operator which can be used to model the behaviour of these processes running in parallel. Processes in LTSA consist of actions they can perform to progress to their next state. Continually running process are modelled by recursion:

```
SERVER = (request->service->reply->SERVER).
```

To model synchronisation, processes can contain an action with the same name as another process. Then, for this action to occur, all individual processes that are part of the composed process must all be able to perform the action as shown in Figure 2.3. To allow a more natural definition of process actions, LTSA also allows action renaming. This makes it possible to create individual processes that have different action names, but can synchronise with each other because the actions can be renamed in the composed process. For example in the following, the action `request` is renamed to `call`, and the action `wait` is renamed to `reply`:

```
SERVER = (request->service->reply->SERVER).
CLIENT = (call->wait->continue->CLIENT).
||CLIENT_SERVER = (CLIENT || SERVER)
                    /{call/request, reply/wait}.
```

```
MAKE_A   = (makeA->ready
              ->used->MAKE_A).
MAKE_B   = (makeB->ready
              ->used->MAKE_B).
ASSEMBLE = (ready->assemble
              ->used->ASSEMBLE).

||FACTORY = (MAKE_A || MAKE_B
              || ASSEMBLE).
```

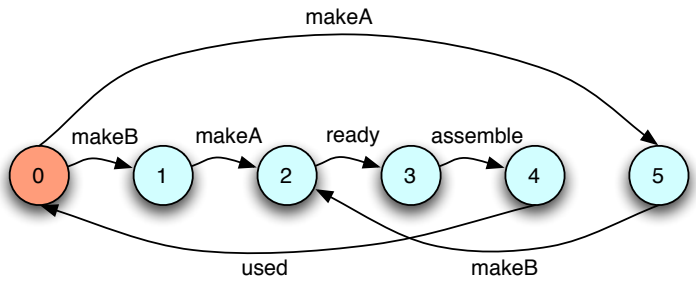

Figure 2.3: Synchronisation on actions by multiple processes. `MAKE_A`, `MAKE_B` and `ASSEMBLE` must all synchronise on the actions `ready` and `used`

Thus, the `CLIENT` and `SERVER` processes synchronise on the actions `call` and `reply`.

Locking and mutual exclusion can be modelled simply in LTSA by defining a `LOCK` process with the definition:

```
LOCK = acquire->release->LOCK.
```

and using it as necessary. Semaphores can also be modelled similarly.

Using LTSA, finding deadlocks becomes simple, as all deadlocked states have no outgoing actions. Searching for deadlocks is in fact automated. Deadlock is but one property of concurrent processes; it is possible to test liveness and other properties using LTSA by defining them with the `progress` and `property` keywords respectively. LTSA also has the ability raise or lower the priority of actions to model more complex systems.

The main limitation of modelling concurrent processes using LTSA or any other such system is the problem that the implementation of the system can still be incorrect despite having a correct model. Also, the state space of large system is often bigger than many of the modelling systems can handle. While it is possible to over come this problem by removing non-essential actions from the model, it is not ideal.

2.4 Debugging and error detection

Debugging is an essential part of the software development cycle. It accounts for the removal of the majority of the bugs that may have manifested themselves during development.

2.4.1 Traditional debugging

Most developers take a methodical approach to debugging consisting of the following steps [30]:

1. Recognising that a bug exists
2. Isolating the source of the bug
3. Identifying the cause of the bug
4. Determining a fix for the bug
5. Applying a correction and testing it

Of those steps, steps 2 and 3 are considered the most difficult, requiring repeated executions to reproduce the bug for analysis, and are often carried out with the aid of an interactive debugger such as `gdb` or `jdb`. Debuggers such as these work by attaching themselves to the program being debugged and then monitor it. They typically allow the developer to stop execution (by manual intervention and setting breakpoints), single step through the code, analyse (and possibly change) program data such as variables, and view the stack.

The functionality provided by debuggers is useful in isolating source of the bug as the developer may run through the program line by line while viewing the effect of the code on the program state. Once the erroneous program state is achieved, the developer will know the sequence of actions that lead up to it and hence be able to deduce the cause of the bug.

However, this traditional style of debugging is not efficient for multi-threaded programs because there are many flows of execution, and thus many stacks. To get an idea of what the program is doing as a whole, the developer must view one thread at a time and continually switch between them. Even then, this will likely reveal only logic bugs in threads, and not those as a result of interactions between threads. There is also an issue with single-stepping because the developer must decide which thread to single-step, reducing its usefulness.

2.4.2 Reversible debugging

As mentioned in subsection 2.4.1, often one of the hardest parts in debugging is to identify the cause of the bug. The reason for this is that repeated executions of the program are required such that the bug can be recreated for analysis. To avoid this, there has been work done on reversible debugging where one can step *backwards* through the program as well as forwards [5, 11]. This can significantly speed up the analysis of the buggy code as it can be run as many times as necessary without restarting the program.

The solution described in the thesis "Detailed program tracing and replaying" [11] achieves reversible debugging by using the Valgrind[2] execution environment to record the changes made to memory, registers and the program counter on particular events as the program progresses. This allows the state at any arbitrary point X in the program to be recreated during "replay", by restoring the last *recorded state* and replaying the program code up X. Thus, backward stepping in the modified version of GDB (GNU Debugger) created as part of the thesis is supported by restoring the state of the program corresponding to the backward step.

Clearly, reversible debugging presents a significant efficiency advantage over traditional debugging. Currently the main limitations of the current products for reversible debugging is that they do not support signals or multi-threaded programs.

2.4.3 Debugging multi-threaded programs

The last section noted that traditional debugging is not efficient for multi-threaded processes. For this reason, there has been some work into how to create interactive debuggers for multi-threaded processes.

These debuggers typically adopt a "stop the world" model [22] where by whenever the developer wants to pause execution for analysis, all the threads in the program being debugged are suspended. Then when execution is restarted, all threads are resumed simultaneously. Still, there remains the issue of identifying individual threads since thread ID's will change on every run of the program. Hence, some threading packages also support thread naming [19] (e.g. The .NET threading package

[2] http://www.valgrind.org

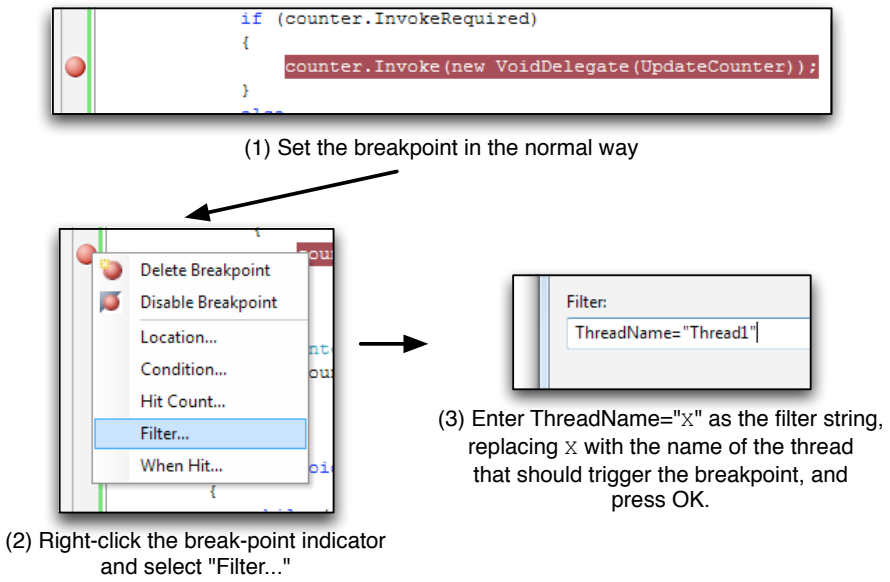

Figure 2.4: Using thread naming and conditional breakpoints to debug multi-threaded code in Visual Studio 2005

- `System.Threading`). Thread naming can also actively assist debugging, by allowing the setting conditional breakpoints based on thread names. In Microsoft Visual Studio 2005, this can be achieved as shown in Figure 2.4. This way, it is possible to find errors within the code that only surface when part of a particular thread.

Despite the work being done on interactive debugging for multi-threaded software, it is still significantly less useful than for single-threaded software. This has led to work being carried out on automatic race detection.

A popular free tool for race detection is Helgrind, which is implemented as an extension ("skin") for the Valgrind debugging and profiling suite, and incorporates the *lockset* algorithm described in [28] with improvements from [16]. There are also a few commercial tools available, such as the Intel Thread Checker [3] and the Sun Thread Analyser[4].

The basic method by which Helgrind operates is to monitor the locations in memory which are accessed by more than one thread, and record the locks that were held by the accessing thread at the time. Then, it is possible to find which lock is used for mutual exclusion, for that memory location, by checking the locks held by other threads when accessing the same location and intersecting the sets. If no such lock can be found, it can be reasonably deduced that there is no consistent locking strategy for that location. Hence, Helgrind reports this to the developer as it may be a cause for data races. To reduce the likelihood of false reports, Helgrind also takes into account the thread execution ordering: if thread executions cannot overlap – for example, due to a `pthread_join` call – then it will not report shared variable access as unprotected.

The main limitation of using tools such as Helgrind is that they significantly degrade the performance of the application, and thus cannot be used in production environments where the execution paths of the program maybe different enough that certain memory locations may have never have been monitored during debugging for the race-detection analysis.

[3] http://www.intel.com/cd/software/products/asmo-na/eng/threading/219785.htm
[4] Previously called the Race Detection Tool. http://developers.sun.com/sunstudio/downloads/tha/index.html

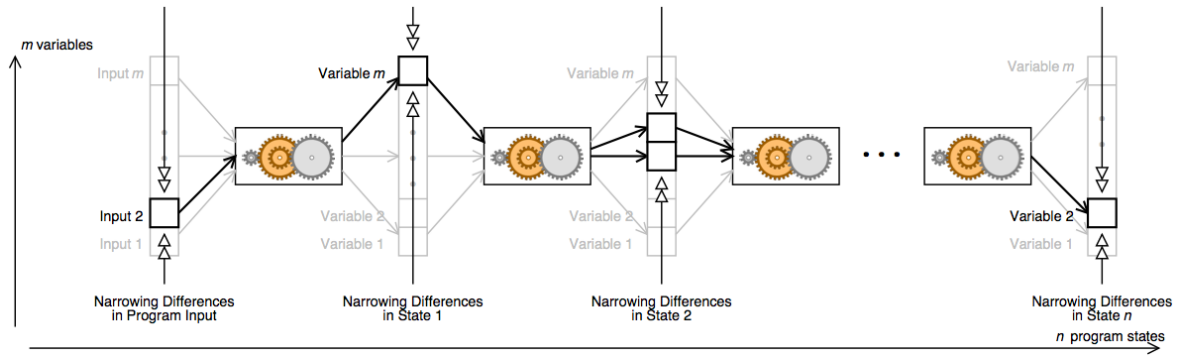

Figure 2.5: Narrowing a cause-effect chain. In each state, out of m variables, only few are relevant for the failure. These can be isolated by narrowing the state difference between a working run and a failing run [34].

2.4.4 Automatic debugging

Recently, research into automatic debugging has experienced some success. An algorithm for this is described in the paper "Isolating Cause-Effect Chains from Computer Programs" [34]. The main idea is to focus on the *program states* at different points in the execution rather than the source code when searching for the cause of a bug. In particular, it considers the difference between the program states on executions where the bug in question does not occur and executions where it does. The intuition behind this is that these states are not only the cause of the failure, but an *effect* of the program code, and thus can be used to locate the faulty code.

The algorithm proposed in the paper actually works in two steps, (1) isolating the failure-inducing input and (2) isolating the relevant states necessary to deduce the necessary cause-effect chain.

The first step is achieved by using the Delta Debugging algorithm [33]. This algorithm works by taking two runs of the program being debugged, $r_{✔}$, a run where the program does not exhibit the bug, and $r_{✘}$ where it does. Then, by iteratively creating and testing the configuration to the program using the differences between the configurations of $r_{✔}$ and $r_{✘}$, it can find the minimal change in the configuration of $r_{✔}$ that causes the failure. This can also be seen as the minimal change to the configuration of $r_{✘}$ that does not cause the failure.

Once the minimal configuration change between $r_{✔}$ and $r_{✘}$ is found, it is possible to isolate the program code involved that causes the bug. This is achieved by looking at the differences in the *program states* on failure-inducing input and on non-failure-inducing input. However, a small change in input to the program may cause large changes in program state. In order to isolate the variables relevant to the failure, it is necessary to isolate the failure-inducing program state (see Figure 2.5). Zeller achieves this by applying delta debugging *on the program states*. The isolated variables constitute the *cause-effect chain* that lead from the root cause to the failure. Hence, they can be used to isolate the bug within the source code.

While delta debugging is obviously a breakthrough approach in debugging, there are, of course, limitations. Firstly, there is the fact that it can only find one cause of failure from several potential causes. Also, there are cases where it will run a large number of iterations to find a large difference, which may not be useful to the programmer due to size. Most importantly for this project, it cannot be used to find errors that are a result of multi-threading since they do not occur on every run. However, it may be possible to apply delta debugging by combining it with a record/replay system [14].

Aside from delta debugging, another approach towards automatic debugging has been to automatically generate complex inputs to programs such that they may trigger bugs [10]. The basic idea of this

approach is to run the program with a 'symbolic' input, that is, input that is initially allowed to be *anything*. As the program runs, the instrumentation that is compiled into the program checks whether the instructions use the symbolic values. In the case that an instruction does, it is not executed and instead is added to a set of input constraints. The constraint is generated from the test that is done on the input. Then, whenever the program carries out a dangerous operation such as a pointer de-reference, the constraints on the current path are checked to see if *any* allowed values cause a bug.

There are obvious advantages to this approach of debugging over interactive debugging, but as with delta debugging, it will not work on multi-threaded applications due to the inability to handle non-determinism.

2.5 Deterministic replay

Deterministic replay is a method of removing non-determinism from program execution.

It works in two stages: record and replay. The record stage is used to monitor the running process and capture the events that occur, their ordering and their timing. The replay stage then uses this information to force the reoccurrence of these events at the exact moments they occurred during recording, thus making the run of the program deterministic.

There are two approaches which one could use to create a recording, or history, of an executing process:

1. **Order based recording** – recording the order and timing of the occurring events. Thus, replaying the exact history will result in the same execution as on the original run given the same inputs from external sources such as data read in from files, network and inter-process communication. This is the approach taken by Russinovich and Cogswell [26].
2. **Data based recording** – recording executed instructions and the data that they used. This method has a large disadvantage in that it usually results in very large log files. This approach is taken by the BugNet (2005) project [23].

While both approaches do work individually, to optimise the performance of the solution, typically a combination of the two approaches is used. In the case of order based recording, some external data must be stored in order to achieve re-execution, and hence some data based recording is usually carried out.

In this project we are considering the capture and deterministic replay of an entire execution. However, there has also been some work into selective capture and replay [24] to deterministically replay certain subsystems within a program. The method for selective capture described in the paper is based on Java, but can be adapted to any object oriented language that has a subset of the features of Java. The basic idea is to group objects into sets of *observed* and *unobserved* objects and monitor the interaction of the observed set with the unobserved set. During replay, the subsystem that was monitored is replayed independently using the log of the monitored interactions with the unobserved objects.

2.5.1 Execution traces

The execution traces generated as part of the record process can be used for several purposes in addition to replay.

One of the most useful things that could be achieved with an execution trace with regards to debugging is a trace simplification to pinpoint the code causing a bug. This is similar in concept to delta debugging, however it can be used to find errors that are a result of concurrent behaviour due to the fact that a trace essentially represents a *possible* serial execution of a concurrent program. Similarly, it could

assist in debugging interactive programs as input can be replayed. While no known systems exist at present, as mentioned in subsection 2.4.4, this could theoretically be achieved using deterministic replay in conjunction with delta debugging for uniprocessors.

With the progress being made in the automatic classification of executions [15], traces could also be used develop a model such that the executing program reports an error before any damage is done (e.g. data being corrupted). In combination with trace simplification and debugging, the system could also greatly increase the success of public beta testing. Instead of simply asking users to report errors, traces could be sent back and processed through the system to check for possible bugs.

Another use for execution traces could be to carry out coverage analysis. Coverage analysis is the process of determining the amount of code that is being covered by automated tests such as unit and functional tests. Execution traces can assist in discovering areas and scenarios not yet tested in a much more thorough fashion, particularly for interactive applications. This is because software maybe used much more differently by end-users than the developers may have envisaged. As such, it is unlikely that these scenarios will have been covered by unit tests.

In a similar fashion to coverage analysis, traces could also be used to find features that are no longer used (unused features increase code complexity and benefit no one) and to find features that need to be added by analysing usage patterns.

2.6 Code instrumentation

Code instrumentation is the process of modifying a program to add code that does not contribute to the program's functionality. For example, to add debug statements, performance monitoring and profiling to a program.

There are several techniques to add instrumentation to program:

1. By modifying the source code by hand
2. Modifying the compiler to add it during compilation
3. Patching the compiled binary on disk
4. Dynamically modifying the binary in memory

Typically, modifying source code by hand is used only to add a small amount of instrumentation. For example, to add debug statements and to monitor the performance of a particular subsystem or section of code. This technique is also used when the instrumentation being added is complex.

Modifying the compiler is another solution and only really used to add instrumentation that takes advantage of the various analyses carried out during compilation or is general in nature. For example the Sun C, C++ and Fortran compilers can take a flag `-xinsturment=datarace` to add data race instrumentation during compilation to a program.

Patching a binary on disk is often used to instrument software that may not have source code available. This method usually requires parsing the machine code within the binary and then adding the instrumentation required. One major concern when carrying out such an operation is to maintain addresses when shifting code. This is a difficult task and often trampolines are used to avoid this problem. This is the method by which instead of shifting segments of code to insert the instrumentation, it is inserted at the end of the binary and a branch is made to it. Upon completion, another branch is made back to the the original location.

Dynamically modifying the binary in memory is the most difficult of the solutions. It is generally used to apply instrumentation selectively based on the current run of the program, though the fact that no modifications to the on-disk executable are needed is a significant advantage. In principle, it is similar

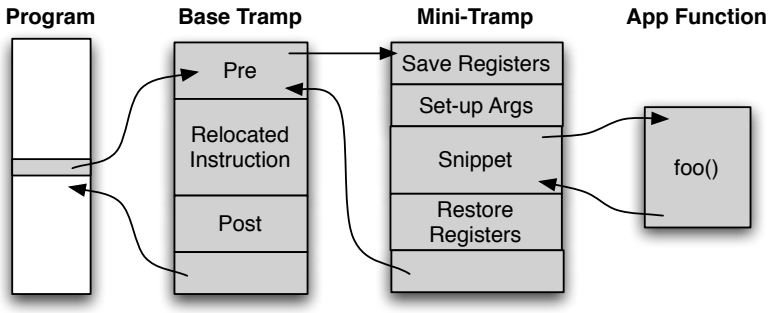

Figure 2.6: Inserting code into a running program using Dyninst [9].

to patching the code on disk, however since the code is actually running additional precautions are required. Dynamic linking can also complicate the instrumentation significantly as memory addresses will change as part of the linking process.

There have been several libraries written to assist in the instrumentation process. One of them is the Dyninst [9] library. Dyninst provides a runtime library for inserting code into a running process and several abstractions to build the code snippets themselves. It also eases much of the complication of dynamic instrumentation as it contains methods to find program points where it can be inserted. Once the point of instrumentation has been found and a code snippet built, Dyninst uses the code trampoline method as shown in Figure 2.6 to insert it into the running program.

2.7 Summary

This chapter has presented an detailed investigation into the area of multi-threading and the debugging of multi-threaded programs. It has discussed the different methods of multi-threading and their advantages and disadvantages. The state-of-the-art methods for the debugging of both single- and multi-threaded programs were also presented and ideas for combining them to produce a more useful solution were given. Finally, the techniques for implementing deterministic replay, uses of execution traces and techniques for program instrumentation were discussed.

Chapter 3

Previous work

This chapter presents some of the previous work that has been done in the area of deterministic replay and the limitations of each work.

3.1 Bugnet (1988)

Bugnet [31] was one of the first ever deterministic record/replay solutions to be created. It was designed to debug distributed applications by monitoring interprocess communication, I/O events, and execution traces for each component process within the distributed application.

Bugnet uses a checkpointing scheme which allows users to rollback to and replay from the position that the checkpoint was taken. Due to the fact that Bugnet is designed to debug distributed software, checkpoints require that the point at which each component process is stopped be synchronised with the others. Bugnet achieves this by stopping each independently on each machine at specified intervals between 15 and 30 seconds long. The short length of of this interval means that clock drift in this period is minimal and hence accurate enough for synchronisation.

To allow for the capturing of IPC messages, Bugnet provides the necessary `send`, `recv` and `replay` primitives that applications must use. These primitives, when used, capture and timestamp the communication so that it can be recreated at the correct time during replay. Each process in the distributed application has its IPC messages captured independently and hence there is no need for synchronised clocks between the distributed hosts.

The Bugnet system itself consists of several support processes working together as illustrated in Figure 3.1:

Global Control Module (GCM) This process is responsible for distributing the application processes to each node and providing network addresses to the Bugnet processes. It is the main co-ordinator of the control processes.

Local Control Module (LCM) This component is responsible for connecting the application processes on each node to their *TRAP* processes, relaying commands from the GCM and to oversee all the processes running on the node.

TRAP During the record stage, TRAP processes are responsible for capturing the IPC and thus for each process within the distributed application, there exists a TRAP process. When the user decides to replay from a specified checkpoint, the TRAP processes arrange the history to match the selected checkpoint and then replay the messages of dead and non-sending application processes.

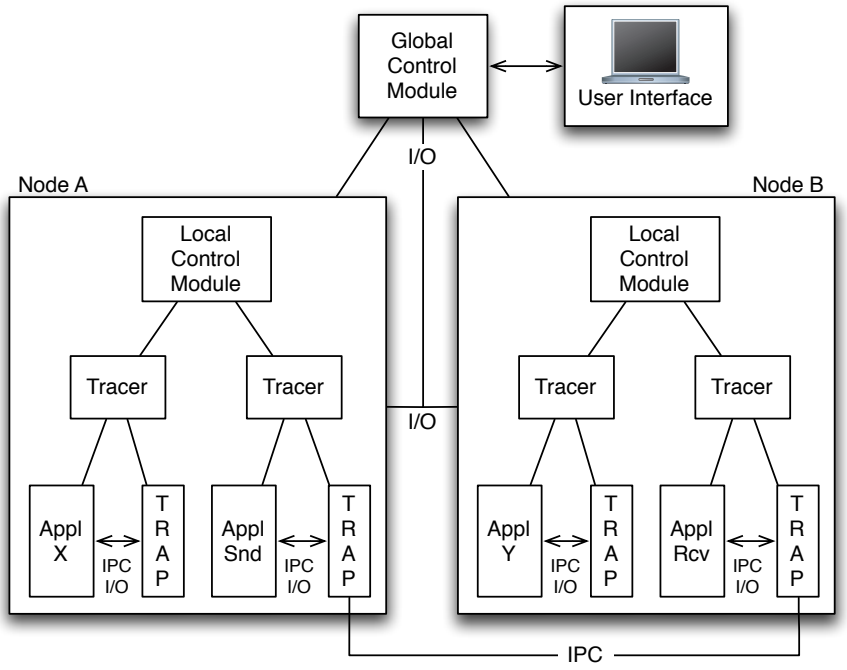

Figure 3.1: Bugnet Architecture [31]

Messages of sending processes are not replayed; however, differences between the newly sent messages and the recorded ones are presented to the user for debugging purposes.

TRACER This is similar to a TRAP process, but monitors the process execution trace during the record stage and controls its execution during both record and replay. There is a TRACER process for each process within the distributed application.

The main limitation posed by the record and replay framework presented in Bugnet is that it can be used to debug only applications written to a specific API (i.e. IPC primitives given and system calls only).

3.2 BugNet (2005)

One of the most recent pieces of work in the area of deterministic replay is BugNet [23]. The focus of this project is very similar to that of *Imitate*, however BugNet concentrates on deterministically replaying the set of instructions that lead up to the program crash; that is, the last set of instructions before the crash, so that they can be sent back to the developer for debugging.

BugNet is based on the Flight Data Recorder (FDR) project [32] which enables deterministic replay on multi-processor systems via dedicated record/replay hardware which records all aspects of the system including low level events such as DMA accesses. BugNet, while still using support from dedicated hardware, uses a different approach. Firstly, since BugNet is designed to debug only application activity, only application and shared library activity is recorded. Secondly, only the first load of a each memory address is recorded. The intuition behind this is that during reply all future states of memory will be recreated, and hence there is no need to record them.

BugNet is implemented using a checkpointing scheme where a new checkpoint is created after executing a specified number of instructions, though they can be terminated prematurely due to context

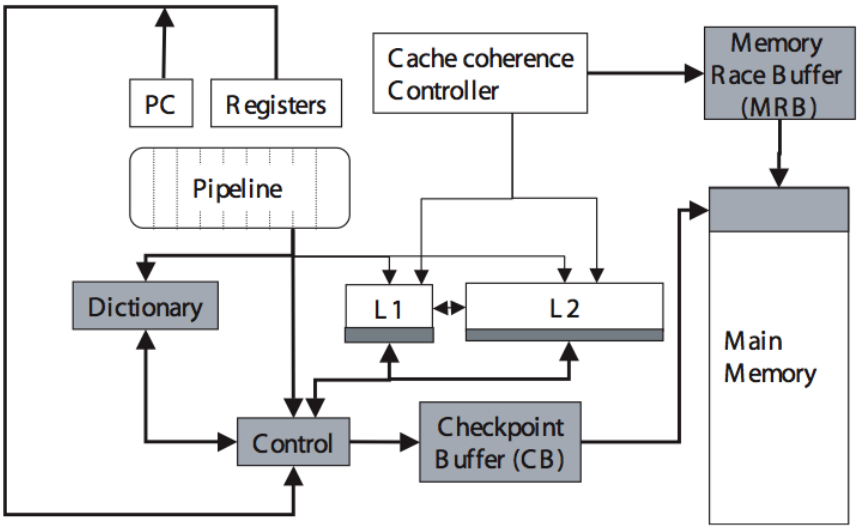

Figure 3.2: BugNet Architecture [23]. The Dictionary is a 64-entry fully associative table which is used to optimise the storage of the loads in FLLs.

switches and interrupts. The data saved during a checkpoint interval is known as an *First Load Log (FLL)* and is stored in a Checkpoint Buffer (CB). One FLL exists for each thread and is initialised with the following information:

- **Process ID and Thread ID** - Used to associate the FLL with the thread for which it was created.
- **Program Counter and Register File** - Used to initialise architectural state when replaying.
- **Checkpoint Interval Identifier** - Used to identify the checkpoint interval corresponding to this FLL.
- **Timestamp** - Needed to order FLLs for a thread according to their time of creation.

Once an FLL has been initialised, all first-loads of a particular memory address have their output logged. This is achieved by using a first-load bit for every word in the L1 and L2 caches and resetting it at the beginning of every new checkpoint. Then, whenever a first load occurs for a memory address within the current checkpoint interval, the output of the load can be logged. On subsequent loads of the memory address within the same checkpoint interval, the first load bit will be set and hence will not be logged. The data contained in the FLL for a particular thread in a checkpoint interval is sufficient to replay that thread for that interval.

To allow for debugging of data races between threads, *Memory Race Logs (MRL)* are kept in the *Memory Race Buffer (MRB)*. These are created and initialised at the beginning of a checkpoint for each thread and are then used to track shared memory access orderings between threads. To achieve this BugNet writes an entry to the MRL whenever a cache coherence reply message is received. These messages arrive whenever the cache detects a write while another thread is attempting to read or write from the cache. The entries in the MRL can then be used to retrieve the memory operation orderings across all threads.

Upon encountering a program crash, BugNet retrieves the latest information (i.e. from the latest checkpoints) stored in both the CB and MRB and saves it to disk. This information can then be sent to the developer to be used for debugging.

Aside from requiring dedicated hardware, the main limitations of BugNet arise from the fact it is based on a checkpointing scheme where by it is only possible to replay only a portion of the execution. Also,

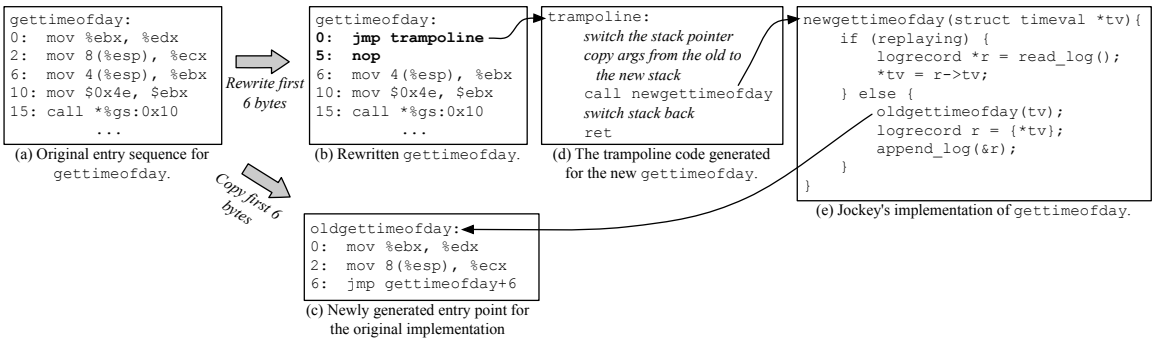

Figure 3.3: System call interception and patching in Jockey for the `gettimeofday` system call [27].

since the checkpoint does not take a snapshot of the virtual memory of the program, it is not possible to analyse the portions of memory not used after the checkpoint was taken. This makes it impossible to analyse errors that may have been caused indirectly by that memory location. Also, since it tracks only application code, it is not possible to find errors resulting from the program's interaction with the operating system.

3.3 Jockey

Jockey [27] is an execution record/replay tool for debugging Linux programs. It is implemented as a user-space shared object file which is linked into the target program dynamically on load, `libjockey.so`. Jockey works by recording timing information, as well as their inputs and outputs, between system calls and CPU instructions that cause non-deterministic behaviour. This allows them to be replayed deterministically in the future. To speed up debugging of long-running processes, Jockey also supports checkpointing.

The implementation of Jockey uses system call interception and instruction patching to implement recording and replay. This is achieved by carrying out the following initialisation steps on start up:

1. All system calls within `libc` with timing- or context- dependent behaviour have their first few instructions patched to intercept the call so that their return values can be logged to be replayed later. This instruction patching occurs as shown in Figure 3.3. It must be noted that in step *(b)* in the `nop`-sled is only added if the fifth byte occurs is in the middle of an instruction. Also, to avoid corrupting the target's memory, the trampoline is dynamically generated and uses a separate stack.

2. Non-deterministic CPU instructions are also patched in a similar way to system calls. This is achieved by looking at virtual memory map in `/proc/N/maps` (where `N` is the PID of the target) and then scans the text sections listed for non-deterministic instructions and patches them.

3. A checkpoint of the process state is taken to make sure that the target sees the same environment and command-line parameters during both record and replay.

4. Control is transferred to the target.

Due to the fact that Jockey is implemented in user-space, there are significant issues with managing memory since the target can, intentionally or accidentally, read or write to memory allocated to Jockey. File handles/descriptors are also affected by similar issues. Jockey handles each of these issues individually:

Heap Jockey cannot use `libc`'s memory management functions to manage its own memory as it risks disturbing the target's execution. To get around this, it uses an `mmap`ed region at virtual address `0x63000000` that is unlikely to be used by the target program.

Stack As seen from Figure 3.3, step *(d)*, the stack is also segregated by switching the stack pointer and copying over the arguments from the old to the new stack. Once the operation is complete, the stack pointer is switched back. Due to the switching of the stack pointer, the target program is isolated from Jockey as no memory above the targets original stack pointer is changed.

File descriptors To avoid altering the file descriptor allocation scheme of the target when Jockey needs to carry out logging or checkpointing, Jockey calls `dup2` after every `open` to move the descriptor to a range unlikely to be used by the target.

Jockey has several limitations which are the result of being a user-space solution. One is the fact that since context-switches cannot be monitored, programs using kernel level multi-threading cannot be potential targets. Similarly, targets that interact with other processes or the OS via shared memory are not supported. Also, since Jockey's data is in the address space of the target, the target can interfere with the operation of Jockey. Due to the way Jockey manages memory, problems also exist with using Jockey when address space randomisation software such as ExecShield is in operation.

3.4 DejaVu

DejaVu [7] is a tool that deterministically record's and replay's the execution of the Jalapeño Java virtual machine and the application it was running on uniprocessor systems for debugging multi-threaded Java applications. Jalapeño is a cross-optimising virtual machine where the application it is running and the virtual machine are optimised together, hence blurring the distinction between the application code and the virtual machine code.

The essence of the approach taken in DejaVu is to replay the Java thread schedule. To ensure thread switches occur at the same point during both record and replay, events causing the switches are first grouped into the deterministic and non-deterministic categories so they can be handled individually.

Deterministic thread switches are caused by `wait` events or by an unsuccessful `monitorenter` as they can cause a thread to block. Synchronisation events, which cause threads to awaken, such as `monitorexit`, `notify`, `notifyAll` and `interrupt` are also deterministic as these are all caused by the application code itself. Non-deterministic thread switches are caused by events as a result of timer functions such as `sleep` and timed `wait` and as a result of thread pre-emption.

The cross-optimisation employed in Jalapeño makes replay of deterministic thread switches simple. This is because during replay, DejaVu recreates the program state of Jalapeño as well, thus recreating the state and data structures within the Jalapeño thread package. This leads to the same thread being scheduled during replay as during record.

For timer based events, DejaVu reproduces the wall-clock time during replay. This is sufficient because timer events are based on a particular application state as well as wall-clock time during replay, and hence reproducing wall-clock time results in deterministic timer events.

Replaying thread switches due to pre-emption is again simplified by the cross-optimisation properties of Jalapeño as the thread package's activity is also replayed in DejaVu. However, the issue of how to identify corresponding events during both record and replay still remains. Wall-clock time is not an accurate basis for these events and using the instruction pointer is not possible as an instruction could be executed many times. To solve this, DejaVu uses the observation that a single point in the program's execution can be uniquely identified by a pair containing the instruction pointer and the number of backward branches executed by the program [20, 26]. Since Jalapeño only pre-empts threads at *yield*

```
// during DejaVu record
// at every yield point
if (liveClock) {
  // only when the clock is running
  liveClock = false;
    // pause the clock
  nyp++;
  if (preemptiveHardwareBit) {
    // preemption required
    // by system clock
    recordThreadSwitch(nyp);
    nyp = 0;
      // reset the counter
    threadSwitchBitSet = true;
      // set the software switch bit
  }
  liveClock = true;
    // resume the clock
}

if (threadSwitchBitSet) {
  threadSwitchBitSet = false;
  performThreadSwitch();
}
```

(A)

```
// during DejaVu replay
// at every yield point
if (liveClock) {
  // only when the clock is running
  liveClock = false;
    // pause the clock
  nyp--;
  if (nyp == 0) {
    // preemption performed
    // during record
    nyp = replayThreadSwitch();
      // initialize the counter
      // for the next thread switch
    threadSwitchBitSet = true;
      // set the software switch bit
  }
  liveClock = true;
    // resume the clock
}

if (threadSwitchBitSet) {
  threadSwitchBitSet = false;
  performThreadSwitch();
}
```

(B)

Figure 3.4: DejaVu instrumentation at yield points for record (A) and replay (B)

points consisting of loop back-edges and method prologues, these can be used in place of backward branches simplifying the implementation.

So far, cross-optimisation in Jalapeño has greatly simplified record and replay. However, this is not an entirely complete picture. Due to the way that cross-optimisation blurs the boundary between the runtime and the application, any instrumentation added to facilitate record and replay can influence the execution of the application. The instrumentation added during record and replay will be different by nature, and as such may result in differences in the executions. To minimise this effect, DejaVu uses *symmetric instrumentation* when complete transparency cannot be achieved. This is a method by which any actions that may modify the JVM or the application execution occur during both record and replay to minimise the difference (Figure 3.4). A simple example of where symmetric instrumentation is needed is in class loading: during record classes that output to files are needed, where as during replay classes that read from files are needed. To make this symmetric, both sets of classes could be loaded for both record and replay.

Cross-optimisation also poses the problem of attaching a debugger since this can also result in a modification of the JVM when carrying out reflection (which must be done in-process) to access data. To solve this issue, DejaVu implements *remote reflection* within the JVM so that reflection can be achieved outside the process. This is implemented in DejaVu by setting up a link between the two JVMs by specifying a list of *mapped* reflection methods which allow access to internal components of objects. These methods will then return *remote objects* to the debugger (local) JVM, that serve as proxy objects for the real objects in the application (remote) JVM. Thus, inter-process reflection is achieved.

The main limitation of DejaVu is the fact that it can only be used to replay Java programs. There is also the problem that DejaVu makes no attempt to replay I/O and thus any files that the target reads must remain exactly the same between record and replay. This also means that it cannot be used to debug software that takes user input or reads from the network as this cannot be reproduced during replay.

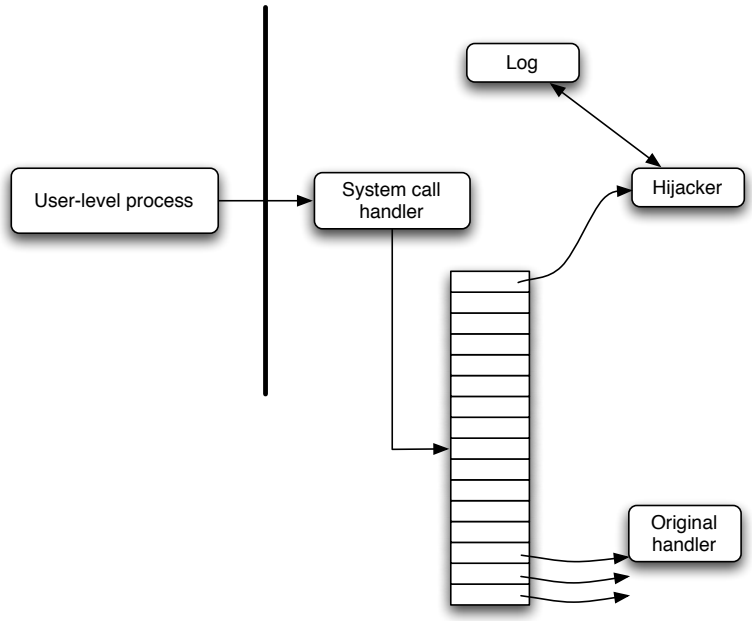

Figure 3.5: Hijacking System Calls for Logging and Replay in Flashback [29].

3.5 Flashback

Flashback [29] is a lightweight OS extension that provides fine-grained rollback and replay to assist in debugging software. It is based around a checkpointing scheme that uses shadow processes to efficiently roll back in-memory state to a previous point in time. To allow for deterministic replay it also logs processes interactions with the system.

The checkpointing facility in Flashback is made available through 3 system calls which can be called by the process itself or from within the debugger:

stateHandle = Checkpoint() Used to take checkpoint that can be used to roll back to the point when the call was made by passing the `stateHandle` to the Replay call.

Discard(stateHandle) Discards the checkpoint associated with the `stateHandle`. Once this call is made, the checkpoint can no longer be used for rolling back.

Replay(stateHandle) Rolls back to the previous execution state determined by `stateHandle` and deterministically replays the execution until the point where `Replay()` is called.

Whenever a checkpoint is taken (there can be many checkpoints), a new shadow process is created. This is achieved by replicating the in-memory state of the running process within the operating system (registers, file descriptors, signal handlers, etc). From this point, the original process' interactions with the environment are monitored so that if a roll back is requested the same external inputs can be reproduced from the log during deterministic replay.

To support multi-threaded programs, Flashback captures the state of all threads within the monitored process so that they can all be rolled back when required. This eases the implementation as there is no need to track thread dependencies and also aids in software debugging as it allows the roll back of all threads which can then be played back interactively step-by-step for the debugging of synchronisation errors and data races.

Deterministic replay from a checkpoint is achieved by recording the interactions of the process with the

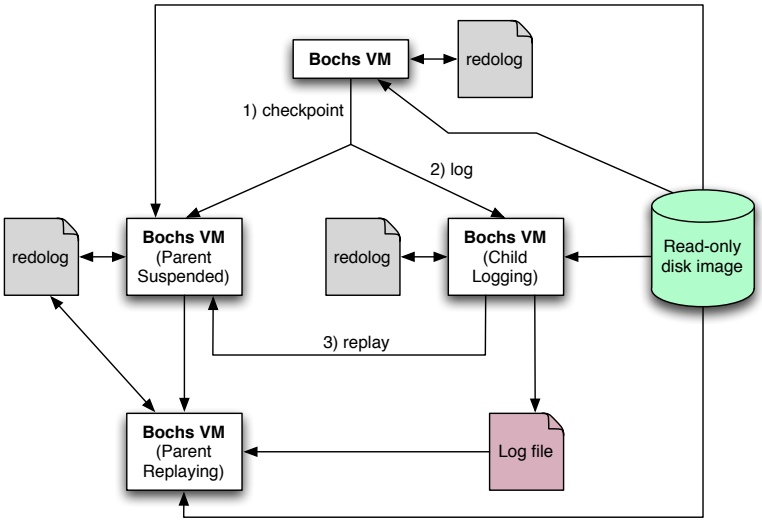

Figure 3.6: ExecRecorder [13]

environment from that point. To facilitate this, Flashback intercepts the system calls so that their outputs are recorded in *log mode* and replayed during *replay mode*. Log mode is entered when the `checkpoint` system call is invoked. Replay mode is entered when `replay` call is made. Intercepting system calls is achieved by replacing the default handler for each system call with a function that does the actual logging and replay as shown in Figure 3.5. In logging mode, the function invokes the original call and then logs the return values and side effects. In replay mode, the function checks that the parameters to the call match those during record and returns the logged values. An exception to replaying calls is made for memory management functions, as the memory must be allocated again during replay, not just replayed as recorded.

Flashback has several limitations. The most serious one is that it cannot be used in production environments without modifications to the program code, something which is not always possible. This arises from the fact that it is based around explicit checkpointing which must be manually done from either or from the debugger. Secondly, Flashback has no support for signals. For most applications this poses no serious issues, however some programs do use signals as a means to be notified of events. For example, sending the `SIGUSR1` user one to reload the configuration.

3.6 ExecRecorder

ExecRecorder [13] is a virtual machine (VM) based full-system log-and-replay system for uniprocessors. It is designed to replay the entire system including the virtual memory, virtual hard disk and memory of all external devices for post-attack analysis and recovery.

The implementation is of ExecRecorder is based on the Bochs IA-32 Emulator and is composed of three components: checkpoint, log and replay. The checkpoint component is responsible for saving system state at a point in time and is implemented by executing the `fork` system call and suspending the *parent process*. To maintain a checkpoint of the hard disk, ExecRecorder uses the undoable disk mode in Bochs. This is a system by which Bochs uses the disk image as read-only and writes updates to a file called the *redolog*. Once execution of the virtual machine has finished, this file can then either be discarded or merged back into the disk image.

The log component's purpose is to log enough information about the non-deterministic events in the

guest system (i.e. hardware interrupts and input events) such that it can be deterministically replayed in the future. To achieve this, ExecRecorder monitors the non-deterministic instructions executed and logs the instruction along with the number of instructions executed since the last non-deterministic instruction was executed. In order to replay input events, enough information about the input instructions is also recorded. For example, the number of bytes transferred and where they were transferred to.

Finally, the replay component is responsible for replaying system execution from a certain checkpoint. In order to do this ExecRecorder uses the `SIGUSR1` signal to wake up the parent (the parent was waiting for this signal) which was used to checkpoint execution, and then disables interrupts and input events so that they can be replayed from the log file.

The main limitation of ExecRecorder arises from the fact that it is a virtual machine based solution. This causes significant performance degradation. Also, the current implementation does not include support for multiprocessors and DMA (Direct Memory Access) resulting in further performance issues.

3.7 Echo

Echo [17] is a deterministic record/replay framework implemented as an OS extension with goals similar to that of this project.

It consists of three main components:

1. The Echo driver
2. The Logger
3. The Re-player

The Echo driver is at the heart of the framework and is provided as a kernel module. This module is responsible for setting up the kernel such that the logger and re-player can achieve their tasks. Upon being loaded, one of the first things it does is to load a new system call table so that the activity of the calls can be captured and replayed. The original system call table is saved so that during the record stage the original calls can be made by the new functions for actually servicing the request. During the replay stage, the original calls are not actually carried out, but simply replayed using the log file.

The other purpose of the Echo driver is to capture and replay the thread schedule. The way in which is done is unique. During capture, it uses the performance counters within the processor to monitor the number of instructions executed in the quantum that the thread was allocated and logs it. When replay is requested, the values logged are used to set up the performance counter in such a way that an interrupt is generated when the counter overflows. Once this occurs, the Echo driver suspends the thread and activates the thread that ran next during the record stage.

The Logger and Re-player are the tools provided to record and replay executions of applications by interacting with the Echo driver.

The Logger works by creating the necessary log files and the shared buffers that will be used to store the log. It then notifies the driver to switch to record mode and starts the process to be recorded. Finally, it continually reads the buffers, which are filled by the driver, and writes them out to disk.

The Re-player works in a similar way to the logger. The first action T carries out is to switch the Echo driver to replay mode. It then sets up the shared buffers and populates them with the initial data recorded during capture. Now that the driver is ready for replay, the application to be replayed is started. From this point on, the Re-player continually refills the shared buffers, which are consumed by the driver to replay the execution, using the data logged during capture.

For debugging purposes, Echo also provides a simple macro to allow the developer to add print statements to the source code, recompile and then replay the execution even though the binary has changed.

Echo provides a unique and elegant solution to deterministic replay, however its dependence on performance counters create a serious limitation in that it is not easily ported to other architectures. Also, the current version is a partial implementation, with only a subset of the system calls supported and no support for signals.

3.8 Summary

This chapter has presented an overview into the area of deterministic replay. This has shown it to be a very active field of research. It has also shown that little work has been done on generating execution traces that can used for both deterministic replay and trace analysis.

Chapter 4

Design

This chapter discusses the design of Imitate and provides a high-level overview the implementation. It introduces the main components that form Imitate and how they interact.

4.1 Overview

The main purpose of Imitate is to assist in debugging multi-threaded programs by allowing deterministic replay and trace analysis. This goal is at the centre of Imitate's design.

The main non-determinism in multi-threaded software comes from interaction with the operating system and the interleaving of threads. Therefore to be able to accurately debug and replay a process, both the data generated by system calls and the thread schedule must be recorded. The easiest way to do this is from the operating system itself, and therefore at the heart of Imitate is a kernel module.

In addition to the kernel module, there are two user-space utilities. Together, they are referred to as user-space monitors. One of the monitors is used to during process capture, while the other is used during process replay. The monitor used during recording has the responsibility of reading the data generated by the kernel module and writing it to disk, while the one used during replay reads data from the disk and passes it to the kernel module. The reason the monitors are used rather than simply accessing data directly from the kernel is that I/O in the kernel is complex and much more prone to failure. In addition to this, using separate monitors means that the process being recorded or replayed does not have to be blocked while doing I/O for reading and writing log files, thus improving performance.

4.2 Kernel module

This is the main component of Imitate as it provides the necessary interception of kernel events to facilitate for the recording and replay of processes.

The interception of system calls is carried out by overwriting the system call table, which routes the calls to their specific handling function, with a custom intercept function defined in the module. This means the module can monitor all system calls made on the system and when a process that is being recorded or replayed makes one, it can act on it. During capture, this action is to let the system call continue to be serviced normally and then recording the result. While replaying, it prevents the call from being serviced by the kernel and instead fabricates the result of the call by replaying from the log.

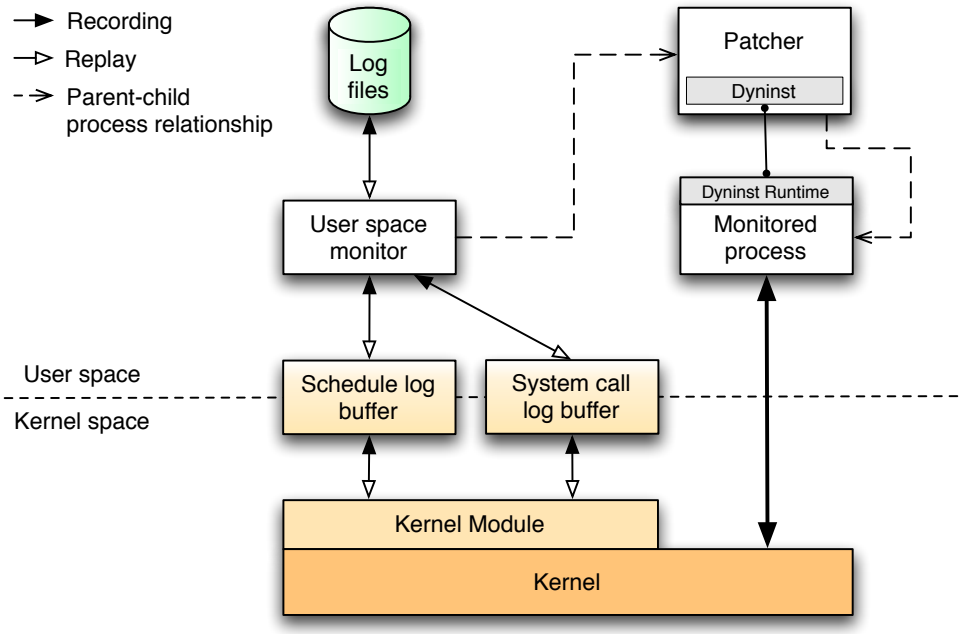

Figure 4.1: Imitate architecture

The interception of the scheduler is more complex. The basic idea of recording and replaying a thread schedule is to identify and record the exact points where threads are pre-empted by the scheduler and then use this data to force the schedule upon the threads during replay. The method used in Imitate to identify these pre-emption points is that discussed by Russinovich and Cogswell in [26]. It involves instrumenting the program to count the number of backward control transfers, and then locating the pre-emption points by the use of a tuple consisting of the number of backward control transfers made up to that point and the instruction pointer at the point. This instrumentation is inserted by the `patcher` component (Figure 4.1) of Imitate. During replay, the recorded tuples are used to force pre-emption by setting the counter value to the negative of that recorded. Then, when the counter reaches 0, a breakpoint is set at the recorded instruction pointer to force pre-emption.

The alternative method to record the schedule is to record the number of instructions executed in each scheduled quantum. This is the method taken by the Echo framework [17]. However, this method was not chosen for Imitate since it does not produce traces that are easily analysed as they contain no reference the code that was running at pre-emption points. Also, they require hardware support to count the number of instructions and thus decrease the portability of the framework.

4.3 User-space monitors

4.3.1 Recorder

The main purpose of this monitor is to start the application to be recorded and flush the buffers used to record the schedule and system call data to disk as necessary. In addition to this, it also has the responsibility of saving the environment variables and arguments of the program so that, during replay, they can be used to accurately recreate the environment in which the program was recorded.

When the record monitor process starts, it opens a connection to the kernel module and prepares it

to begin recording. This involves notifying the kernel module that it is a monitor process, causing the kernel to set up the necessary buffers, and then to memory map those buffers so that they can be read and flushed to disk when required. Once this is complete, it calls the `patcher` to instrument the backward control transfer counter and start the program to be recorded.

Upon the completion of instrumentation, it runs an event loop that is used to save the buffers to disk whenever it is notified to do so. This loop exits when it receives an `EXIT` message from the kernel module.

4.3.2 Replayer

The replay monitor is used to start the application to be replayed and refill the log buffers, as they are consumed in the kernel module, by reading the log files from disk.

When the replay monitor starts, it creates a connection to the kernel module and prepares to replay in the same way as the record monitor. It then proceeds to reading the saved arguments and environment from the log files on disk and starts the `patcher`, which in turn will start the application to be replayed. The `patcher` is passed the saved environment as an argument so it can restore it for the replayed process.

Finally, it starts its event loop and refills the buffer from disk whenever it is notified by the kernel module.

4.4 Summary

This chapter has presented the architecture of Imitate and has given rationale for the decisions made and how they relate to the goals of the project. It has also provided an overview and discussed the three components of the project, the kernel module and the two user space monitors individually as well as their interactions in producing a functional system.

Finally, it has also discussed the process of instrumenting a backward control transfer counter and how it can be used to capture and replay the thread schedule of a multi-threaded program.

Chapter 5

System call capture and replay

This chapter presents a detailed description into how system calls, one of the major causes of non-determinism in a process, are captured and replayed in Imitate. As part of this, there is also an explanation of the method by which Imitate differentiates processes that are being recorded from those that are not.

5.1 System call interception

The process of capturing and replaying system calls in Imitate is achieved by overwriting all entries in the system call table with the address of a custom intercept function, `syscall_intercept`. This function is the single point of entry for all system calls that are being intercepted. It is responsible for calling the pre- and post- system call *callback* functions, before and after the real system call respectively.

The *callback* functions have the responsibility of carrying out basic home-keeping activities, such as saving the saving the system call number, before passing control to the pre- and post- system call *handling functions*. There are two unique handlers for each intercepted system call and these are responsible for the bulk of the work that is carried out during record and replay of the associated system call:

`pre_systemcall` **handler** This is executed from the pre-system call callback function, and thus is invoked prior to the execution of the real system call. In general, this handler does not carry out any action during recording, though there are exceptions such as for the `exit_group` system call. During replay, it used to return the recorded result to the replayed application.

`post_systemcall` **handler** This is executed from the post-system call callback function, and therefore is invoked after the execution of the real system call. The main purpose of this handler is to write out the return value(s) of the system call to the log buffer during recording. It is generally not invoked at all during replay – exceptions to this rule include replaying the `mmap` system call.

To simplify the association of handlers to their corresponding system call, the pointers to the handlers are stored in two arrays indexed on the system call number. For example, the `pre_mmap` handler can be found at index 192 of the pre-system call hander array, as the `mmap` system call has a call number of 192.

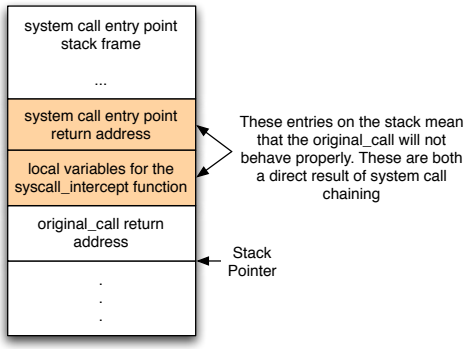

Figure 5.1: The stack just after calling `original_call`, when using the system call chaining method of interception shown in Listing 5.1

5.1.1 Maintaining the stack

Under Linux, system calls depend on the kernel stack having a particular structure, making system call interception difficult. This means one cannot simply intercept calls by chaining the real system call function as this would modify the stack by leaving one more return address on it. I.e. the following will not work as the stack would look like that shown in Figure 5.1 when calling `original_call`:

```
long syscall_intercept(long arg1, void* arg2)
{
  /* Pre-call interception work here */
  long ret = original_call(arg1, arg2);
  /* Post-call interception work here */

  return ret;
}
```

Listing 5.1: System call chaining. This method will place an additional return address entry onto the stack and thus cannot be used in Linux to intercept system calls.

To get around this problem, the `syscall_intercept` function is very careful to remove the return address that is stored on the stack before calling the real system call function. However, doing this alone will lose all reference to where the interception function must return once complete. Therefore, the pre-system call callback is used to save this return address and the post-system call callback is used to restore it. How this is done is best understood by looking at the actual code for the function (Listing 5.2).

Saving the return address is relatively simple as parameters to functions and their return address are both saved on the stack by convention on the X86 architecture. This makes it possible for the pre-system call callback to simply treat the return address as a parameter. The address is then saved to an array `syscall_return_addresses` at entry `PID`, where `PID` is the process identifier of the thread making the system call, by the callback. The reason that one entry exists per thread is that there is actually more than one way to make a system call, all of which may leave different return addresses on the stack. Thus, to make sure we return to the correct point for each individual thread after interception, it must be saved on a per-thread basis.

Once the address is saved, it is safe to remove it from the stack (line 10) prior to invoking the real system call handler (line 11).

```
1  syscall_intercept:
2      pushl   %eax                    /* Save system call no */
3
4      call    pre_syscall_callback    /* Pre-call */
5      cmpl    $0, %eax                /* Did we get a NULL? */
6      jne     replay_return           /* No... Perform replay */
7
8  original_call:
9      popl    %eax                                /* Restore syscall number */
10     add     $4, %esp                            /* Remove return address */
11     call    *original_sys_call_table(,%eax,4)   /* Call original handler */
12
13 original_call_return:
14     pushl   $0      /* We will restore original syscall return address here */
15     pushl   %eax    /* Save system call return value */
16
17     call    post_syscall_callback   /* Post-call */
18     movl    %eax, 4(%esp)           /* Restore return address */
19     popl    %eax                    /* Restore system call return value */
20
21 intercept_return:
22     ret
23
24 replay_return:
25     add     $4, %esp        /* Remove saved system call number from stack */
26     movl    (%eax), %eax    /* Set replayed return value */
27     ret
```

Listing 5.2: The `syscall_intercept` function

When the real system call handler returns, the saved address must be restored. To do this, an empty space is created in the stack where the saved return address will be written (line 14) and then post-system call back is executed. Once the callback completes execution, it `returns` the saved address in the `%eax` register. Finally, before control is returned to the system call entry point, this register is copied to the previously created space (line 18) to complete the restoration of the stack.

5.1.2 Setting up the intercepts

When the kernel module is loaded, some initialisation takes place in order to set up the system call intercept mechanism.

Firstly, the system call table is copied into the `original_sys_call_table` array. This is done so that the real system call functions can still be invoked as part of the interception function even when the system call table has been overwritten. The reason this is required is that system calls must still be serviced during capture and for processes that are not being recorded or replayed.

Secondly, all entries in the system call table are overwritten with the address of the `syscall_intercept` function. This actually sets up the interception and enables the kernel module to track each and every system call made on the system and invoke the pre- and post- system call *handlers* as necessary.

The final step in setting up system call interception is to populate the pre- and post- system call handler array's with their respective handlers. For system calls that do not have handlers, the entries in

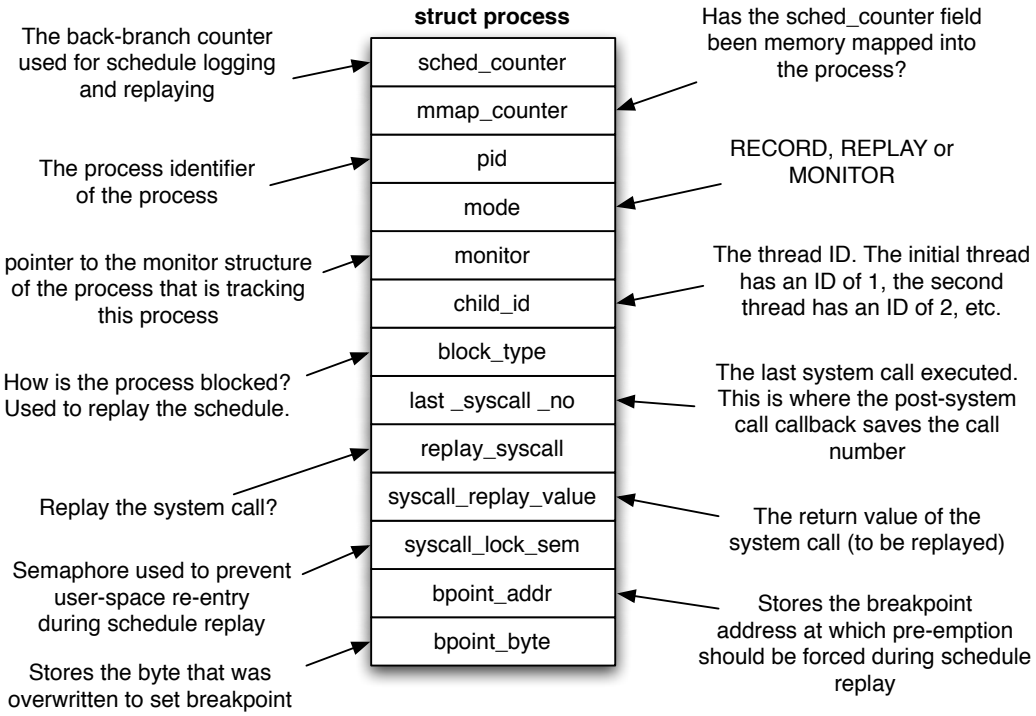

Figure 5.2: The `process` structure

the array's are set to have the address of the special empty function `empty_handler` which does nothing.

Upon completion of the above, the kernel routes all system calls to the `syscall_intercept` function as all the entries in the system call table contain its address. From now on, the decision to execute the real system call function rests with the intercept function and the callbacks which will decide based on whether the process making the call is being recorded, replayed or not monitored at all.

5.2 Tracking monitored processes

Being able to differentiate the processes that are being captured/replayed from those that are not is important. Without this ability, it would be impossible to collect the correct data required for debugging and replay.

In Imitate, the core of process tracking is the `processes` array which is used to store pointers to `process` structures (Figure 5.2), which in turn store essential pieces of information about a tracked process. This array has a size of `PID_MAX_LIMIT`, which is the maximum number of processes that may be running on the system at any one time. The rationale for the size of this array is based on efficiency while removing restrictions on the number of processes that can be tracked. Having the array with this size means that entries in the array can be indexed on the PID enabling efficient retrieval. Also, since an entry can exist for each possible process on the system, there is no limit to the number of processes that can be tracked.

The initialisation of the `processes` array is done when the kernel module is loaded by setting all entries to **null**. This makes it easy to differentiate tracked processes from the untracked by checking for non-**null** entries.

There are two possible ways an entry in the processes array can become non-**null**, thus making the process with the PID of that entry tracked by the module:

1. **The application to be recorded/replayed sends an `ioctl` message to notify the kernel module.** This occurs only when the first thread (i.e. the application) is starting. The message is sent by the application by means of a patch to the `main` function of the process' memory image which makes the `ioctl` system call. Process image patching is explained in more detail in section 6.1. Two types of messages may be sent, `APP_RECORD` and `APP_REPLAY`, which are both self explanatory.

2. **The tracked process creates a new thread**. When a tracked process creates a new thread, it does so via the `clone` system call. Therefore, whenever this call is intercepted its parent's `process` structure is copied into the entry of the `processes` array representing the PID for the new thread. This is required to make sure all threads of a process are tracked by the kernel module.

   To make sure that a new thread's `process` structure is set up before it begins execution, they are created in the `STOPPED` state and only allowed to continue after the process structure is completely set up. This is achieved by overriding the flags to the `clone` system call in the pre-system call handler and then sending a `SIGCONT` (continue) signal after the process structure is correctly set up.

5.3 Recording system calls

Recording system calls (Figure 5.3) is achieved by using the system call interception mechanism described previously in section 5.1 and works as follows as follows:

1. When a system call is made, the system call entry point (`system_call`) looks up the function to call in the system call table and executes it. Since all entries point to `syscall_intercept`, it is executed in every case.

2. The pre-system call callback is then invoked from `syscall_intercept`. This saves the system call return address and the call number – the call number has to be saved because all reference to it will be lost as the real system call function will overwrite it with the return value of the system call. If it was not saved, it would not be possible to call the post-system call handler as we would not have an index into the post-system call handlers array.

3. The pre-system call handler is the invoked from the callback. This typically returns immediately during recording, however for calls that do not return, such as `exit_group`, some processing may be carried out and a log entry written.

4. Since we are recording, the original system call is now serviced by looking at the original system call table that was saved during initialisation and invoking the correct function.

5. Once the system call has been serviced, `syscall_intercept` invokes the post-system call callback.

6. The callback then invokes the post-system call handler which records the return value(s) of the system call to the log buffer. This buffer will be eventually be flushed to disk when it is full by notifying the user space process monitor.

7. When the post-system call callback completes, it `returns` the saved system call return address to the `syscall_intercept` function which then places it back on to the stack in the position where it originally was. This enables it complete and restore control back to the system call entry point.

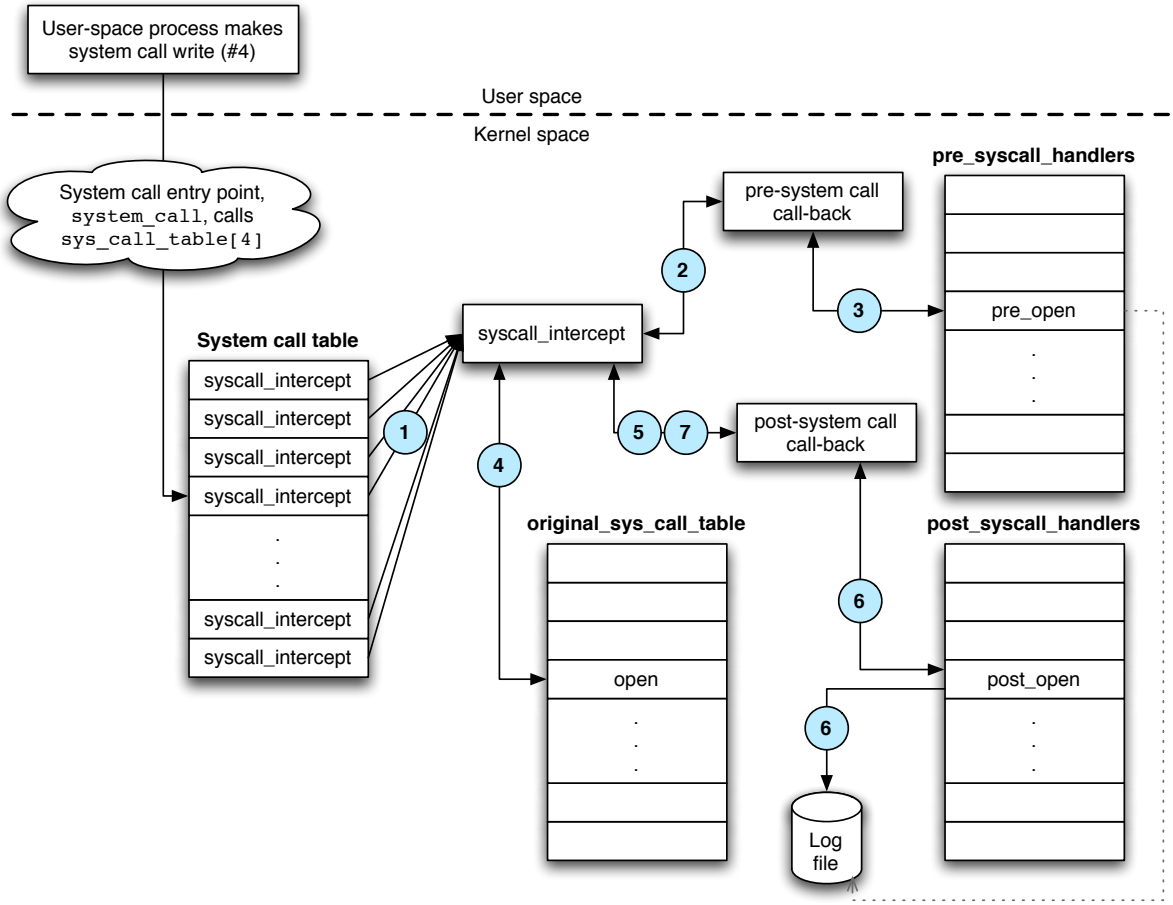

Figure 5.3: Recording system calls

5.4 Replaying system calls

The replay of system calls is carried out in a very similar way to recording them however, since were are replaying data, in most cases the original system call is not executed. The steps to replay calls are given in Figure 5.4 and those that are carried out differently to recording are described below:

3. When the pre-system call handler is called, it usually reads the log file and replays the recorded value(s) by copying any necessary data into the process address space and then calling the `replay_value` macro.

   For some system calls, such as `fork` and `execve`, the real system call function must still be executed to set up kernel data structures. In this case, the `replay_value` macro is not called and instead the pre-system call handler may modify the parameters passed into the handler to recreate the same behaviour as that recorded.

4. If the pre-system call handler invoked the `replay_value` macro, the pre-system call callback will `return` the address of the value that must be replayed to `syscall_intercept` function. Once it has replayed the value (Listing 5.2, line 26), it will return without calling the real system call function or the post-system call callback.

   For the calls that need to execute the real system call (i.e. the pre-system call handler did not call `replay_value`), the `syscall_intercept` function behaves the same way as when recording.

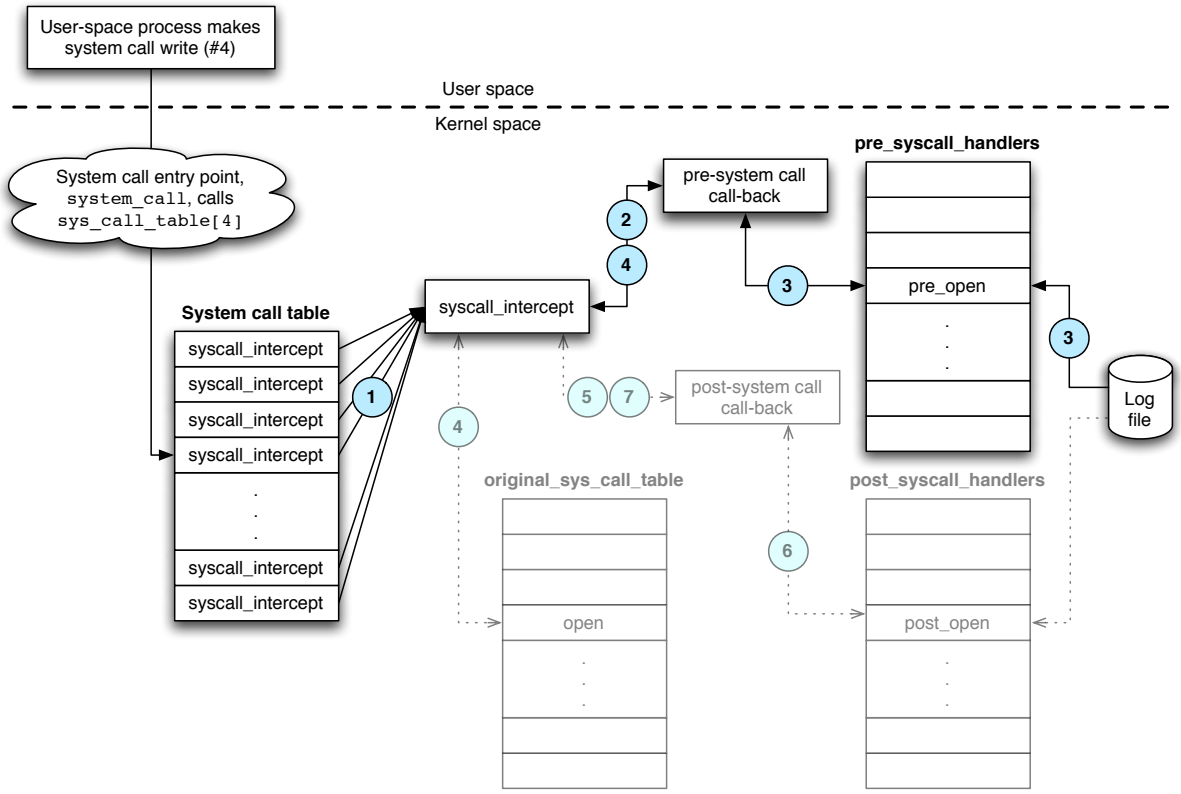

Figure 5.4: Replaying system calls

```
#define replay_value(P,X) seek_to_next_syscall_entry(); \
                          (P)->replay_syscall = 1; \
                          (P)->syscall_replay_value = (X)->return_value
```

Listing 5.3: The `replay_value` macro

6. If the post-system call callback is executed during replay (see step 4 above), the log file may be accessed to replay data. For example, replaying the `mmap` system call requires that the memory contents that existed in memory mapped area during recording be copied into the new memory mapped area during replay.

5.4.1 Replaying system call return value

As discussed previously in this section, replaying the system call return value is done by calling the `replay_value` macro from the pre-system call handler. The macro is shown fully in Listing 5.3. While it does not look particularly special, it plays an essential role in enabling the replay of return values of system calls without having to explicitly jump the exit point of the system call entry function (this is undesirable as it would require hard coding its address, leading to portability issues).

The part of the macro that is significant for replaying the return value of a system call is the last two lines. The second line sets a flag in the `process` structure `P` to tell the pre-system call callback to replay the return value, and the final line stores the value that is to be replayed into the process structure.

After this macro has executed and the pre-system call handler has returned, the pre-system call callback will check the `replay_syscall` flag in the process structure. If set, it will `return` the address of field storing the replay value to the `syscall_intercept` function as follows:

```
if ((process->mode == MODE_REPLAY) && (process->replay_syscall))
{
    VVDLOG("Replaying call return value for call %ld", syscall_no);
    return &(process->syscall_replay_value);
}
```

The `syscall_intercept` function will then replay the value at that address by copying it into the `%eax` register as shown in Listing 5.2 on line 26.

5.5 Summary

This chapter has given a detailed overview of the implementation of system call record and replay in Imitate. It has shown how system call interception is achieved by overwriting the system call table with a custom intercept function. In addition to this, it has shown how the original system call table is saved in order to allow the servicing of these calls during recording. Furthermore, it has discussed the problems associated with interception such as maintaining the stack and how they were solved by removing and restoring the return addresses.

This chapter has also discussed the mechanism by which it keeps track of processes being recorded and replayed by maintaining an array of `process` structures to store state.

Chapter 6

Schedule capture and replay

This chapter details the second major portion of Imitate: schedule capture and replay. It details how a backward control transfer counter is added to an already compiled program and how it is used to record and force the schedule during replay.

6.1 Backward control transfer counting

At the center of recording and replaying the schedule in Imitate is backward control transfer counting. The intuition behind using the number of backward control transfers to record and replay a thread schedule is that any point in a process' execution can be identified by a tuple, the number of backward control transfers made and the instruction pointer. Thus, this tuple can be used to uniquely identify any point at which a thread is pre-empted [26], and hence can be used to record the schedule.

To make it possible to count the number of backward control transfers, Imitate inserts instrumentation into the process memory image (also referred to as the *process image*). This is facilitated by the Dyninst [9] instrumentation library. The method used in Imitate to patch the process image is to use Dyninst to start the application to be monitored (replayed or recorded) in the `STOPPED` state, and then to apply the instrumentation to increment the counter for every back backward control transfer before letting it continue. The alternative method is to start the application to be monitored manually (by using the `fork` and `execve` system calls), use Dyninst to attach to it, and then add the instrumentation. This way was not chosen as attaching to an already started process adds considerable complexity and also runs the risk of the monitored process executing some code before it has been patched.

6.1.1 The patcher

The need for Dyninst to start the program to be recorded/replayed means that monitor process cannot be used to instrument it, as both Dyninst and the monitor process itself have an event loop that must run in the *main* thread. Therefore, a new mutator program was written, called `patcher`. This is invoked from the monitor process with arguments to start the process to be recorded/replayed, and does so using the Dyninst `createProcess` method.

Now, since we are using Dyninst to start the mutatee (the program to be instrumented), it is not possible for it to inherit the connection to the kernel module from the monitor process when the `fork` system call is made. Hence, a patch is applied to the beginning of the `main` function of the process image to create this connection and notify the module that the process is being recorded/replayed. This patch is

```
int dev = open("/dev/imitate0", O_RDWR);
if (dev < 0)
{
    perror("Unable to open /dev/imitate0");
    exit(2);
}
if (ioctl(dev, IMITATE_APP_RECORD, getppid())) < 0)
{
    perror("Notifying kernel device of RECORD");
    exit(2);
}
```

Listing 6.1: Code equivalent of the patch used to create a connection to the kernel module. During replay this `IMITATE_APP_RECORD` is replaced with `IMITATE_APP_REPLAY`.

equivalent to the code in Listing 6.1 and is built using the Dyninst abstract tree for building instrumentation.

As can be seen, the patch is quite large and uses library functions that may or may not be loaded into the monitored process' image. This means that applying this patch requires that the library that contains these functions be loaded first into the mutatee's memory image. This is done by using the Dyninst `loadLibrary` function available on the `BPatch_image` object (this object represents the process image of the mutatee). Once this is complete, the patcher searches the process image for the required functions, obtaining their addresses, and goes onto building the patch. Finally, the patch is applied using the `insertSnippet` function provided by Dyninst.

6.1.2 Memory mapping the counter

Once the connection to the kernel module has been made, the counter that will hold the number of backward control transfers is ready to be mapped to make it accessible to the monitored process (remember that the counter is stored in the `process` structure within the module and as such, is normally only accessible to kernel-space). To do this the `main` method of the monitored process is further instrumented. Memory mapping from the kernel module is used instead of direct allocation of the counter since both the kernel module and the process itself need to access it. Also, memory mapping removes the need to handle page faults as the counter will always be in memory – since kernel memory is never swapped out. Furthermore, memory mapping is simpler than allocating the counter within the monitored process' address space and having the kernel module read it explicitly.

The patch used to perform this memory mapping is equivalent to the code shown in Listing 6.2. Notice that mapping this counter also requires allocating the memory to store its address and finding the `mmap` method within the process image.

When the kernel module receives this memory map request from the monitored process, the memory map handler builds a page table entry to map the `sched_counter` field of the `process` structure into the monitored process' address space. This is done by using the `remap_pfn_range` function. For a detailed discussion on memory mapping, see section 7.3.

6.1.3 Patching backward control transfer points

Finally, now that the counter is accessible to the monitored process, the backward control transfers can be instrumented to increment it. Counting backward control transfers requires adding instrumentation

to two things: (1) the back-edges of loops and (2) the entry points of functions.

In Imitate, both of these places are instrumented at the same time by iterating through all the functions available in the process image one by one and applying the patch. The steps taken to do this are as follows:

1. A list of all functions available in the mutatee's process image is obtained using the `getProcedures` function of the process image object.
2. The instrumentation to be applied at each control transfer is built. This is done differently based on whether the process is being patched for record or replay. For recording, the patch is used to simply increment the counter. During replay, the patch is used to increment the counter and then test its value for `0` and send a message to the Imitate kernel module if it is. The reason for this is explained in detail in section 6.3.
3. The list of functions is iterated and each function is patched individually using the following steps:
   (a) The entry point of the function is instrumented to increment the counter. This is necessary as a function call is also a backward control transfer.
   (b) The flow graph of the function is obtained and all loops are found. This is done by first using the `getCFG` method of the function object to obtain the control flow graph. Then the `getLoops` method on graph is called to obtain references to all the loops.
   (c) Each loop is patched on its back-edge by finding the back-edge point, using the `getBackEdge` method of the loop object, and calling the `insertSnippet` method on it.

Once the counter has been patched in at all necessary points, the monitored process is allowed to continue. This is done by calling the `continueExecution` method of the process object that was obtained by starting the application.

6.1.4 Uninstrumentable functions

While most functions can be instrumented without problems, some can cause serious errors and even program crashes if they are modified. The main ones to take note of are private I/O related and dynamic linking related functions that exist within the standard C library, `libc`.

The reason for these uninstrumentable functions is unknown at the current time, especially considering that the patch is very small and minimally invasive. It is thought to be a result of the Dyninst runtime using them while patching, however a thorough investigation is needed into the causes of these errors.

```
sched_counter_t* counter = malloc(sizeof(sched_counter_t));
if ((counter = (sched_counter_t*) mmap(0, sizeof(sched_counter_t),
  PROT_READ | PROT_WRITE, MAP_SHARED, dev, 0)) == MAP_FAILED)
{
    perror("Mapping software counter");
    exit(2);
}
```

Listing 6.2: Code equivalent of the patch used to memory map the backward control transfer counter into the process being recorded or replayed.

```
1  void (*context_switch_hook)(struct task_struct *prev,
2                              struct task_struct *next) = NULL;
3  void set_context_switch_hook(void (*csh)(struct task_struct*,
4                                           struct task_struct*))
5  {
6      /* Set hook only if none exists or is being removed */
7      if (!csh || !context_switch_hook)
8          context_switch_hook = csh;
9  }
10 EXPORT_SYMBOL(set_context_switch_hook);
11
12 static inline struct task_struct *
13 context_switch(struct rq *rq, struct task_struct *prev,
14             struct task_struct *next)
15 {
16   /* ... variable declarations ... */
17
18   if (context_switch_hook)
19     context_switch_hook(prev, next);
20
21   /* ... context switch code ... */
22 }
```

Listing 6.3: The scheduler hook

In the current implementation of Imitate, these functions are simply skipped during patching by comparing their names to a pre-defined blacklist. There is a chance that these unpatched functions cause inaccurate schedules to be captured, however the likelihood of this is low as most of these functions do not contain loops and are called very infrequently (usually only at the start of a process if at all).

6.2 Schedule capture

In order to capture the thread schedule of a process, Imitate intercepts the scheduler and whenever a thread is pre-empted for another, its thread ID, backward control transfer counter and instruction pointer are recorded to the schedule log.

6.2.1 Context switch interception

Context switches are intercepted by installing a hook into in the scheduler. This is a small piece of code which is added to the scheduler at the point where the interception should occur that makes a callback from that point. The code for this hook is shown in Listing 6.3. When the Imitate kernel module is loaded, it registers a callback function for this hook by calling `set_context_switch_hook`. It is this callback function's responsibility to record the thread schedule.

The hook is added to the `context_switch` function of the scheduler as shown (lines 18 and 19). This is the point at which the scheduling algorithm has decided the next process to be executed, but has not yet changed over the virtual memory structures or the stack. It is the natural place to insert this hook as it allows the callback function to be passed the kernel task structures for both the current and next process.

6.2.2 Recording the thread schedule

The main portion of recording the thread schedule occurs in the context switch hook callback function. When this function is invoked, it first checks that either the thread being swapped out, or the thread being swapped in is being recorded. If neither are being recorded, then the function immediately returns as the context switch does not concern any threads that we are interested in.

If the context switch concerns a recorded thread, it executes two portions of code.

The first portion checks if the thread being swapped out is being recorded and if so, saves the address of its task structure into the `last_running_thread` field of the `monitor` structure.

The second portion concerns the thread that is being swapped in. If this thread is being recorded, an additional check is performed to see if it was the last running thread of the application (by comparing the `last_running_thread` field of the `monitor` structure to that of the thread being scheduled). If they are not the same, then the running thread of the recorded application has changed and a new entry in the schedule log is written. This entry stores the last running thread's instruction pointer and backward control transfer counter as this identifies the point at which it was swapped out. Also, the backward control transfer counter is reset in preparation for the execution of the new thread.

Using this method of recording the schedule means that if a thread is scheduled twice in a row, no special consideration has to be given. This is because in this case the `last_running_thread` field of the `monitor` structure will match that of the process being swapped in and hence no log entry will be written.

Recording the last quantum

The method of recording the thread schedule described above has one limitation: it does not record the last quantum of the monitored application. This is because a log entry is only written when the next thread to be scheduled is also being recorded. This will never be the case in the last quantum as the process will exit.

Imitate overcomes this issue by saving the final log entry from the intercept of the `exit_group` system call. This is the system call made whenever a process completes its execution and thus, is executed by every process before exiting.

6.2.3 Obtaining the instruction pointer

Whenever a process enters the kernel, its instruction pointer is saved so that it can be restored when returning to user-space. On Linux, it is saved on the stack along with the other registers and has no explicit reference to it. This means that retrieving its value, so that it can be used as part of a schedule log entry, takes several steps (Listing 6.4):

- Firstly the stack pointer is obtained from the task structure (line 3).
- Then, the pointer to the stored registers is calculated from the obtained stack pointer and the size of the structure that is used to store the registers (lines 4 and 5).
- Finally, the instruction pointer is obtained from the registers structure at the calculated address.

```
1  inline long get_user_mode_instruction_pointer(struct task_struct *task)
2  {
3      void *stack_top = (void *) task->thread.esp0;
4      struct pt_regs *regs = (struct pt_regs*) (stack_top -
5                                                sizeof(struct pt_regs));
6      return regs->eip;
7  }
```

Listing 6.4: Obtaining the user-mode instruction pointer

6.3 Schedule replay

At the current time, schedule replay is non-functional in Imitate. This is due to the many unexpected and difficult problems encountered during its implementation. Please see chapter 9 for details on the issues encountered. However, many of the parts required for replay are in place so this section will describe their implementation.

Replaying the schedule in Imitate is designed to work by setting a breakpoint at each individual pre-emption point stored in the schedule log. When this breakpoint is reached, the kernel module traps the interrupt generated and suspends the thread. It then reads the schedule log and uses it to enable the next thread that ran during recording. In this way, the thread schedule that occurred during recording is forced upon the process during replay.

6.3.1 Finding the pre-emption points

Before breakpoints can be set to force pre-emptions of threads, the point at which they are to be made must first be found. This has to be done dynamically as the process executes, because setting a breakpoint at the instruction pointer stored in the schedule log statically could result in the breakpoint being hit many times if it is in a loop or a function.

Imitate achieves the process of dynamically setting the breakpoint in the following way:

1. The kernel module sets the value of the backward control transfer counter to the negative of the value stored in the schedule log.
2. The process runs as normal incrementing the counter as control transfers happen.
3. At each point where the counter is incremented, a check is done to see if the counter has reached `0`.
4. If it has, the kernel module is notified to set the breakpoint at the instruction pointer stored in the schedule log. This is done by sending a `SET_BPOINT` ioctl message to the kernel module.

This is the same method as that described in [26].

Edge cases

In addition to setting a breakpoint when a `SET_BPOINT` message is received, there are two edge cases that must be handled especially. They are the cases when the process to be replayed has an initial schedule log entry where the counter is `0`, and the case when a breakpoint is received to force a pre-emption and the next log entry has a counter value of `0`.

```
1     /* Save byte at breakpoint address */
2     if (get_user(process->bpoint_byte, (char*) log_entry->ip) == 0)
3     {
4         /* Set breakpoint */
5         if (put_user((char) 0xCC, (char*) log_entry->ip) == 0)
6             process->bpoint_addr = log_entry->ip;
7         else
8             ERROR("Couldn't set breakpoint for replaying schedule.");
9     }
10    else
11        ERROR("Couldn't save byte at breakpoint address.");
```

Listing 6.5: Setting a breakpoint

In both of these cases the kernel module immediately sets the breakpoint at the address stored in the schedule log before letting the thread continue.

6.3.2 Setting breakpoints

The x86 architecture has two ways of setting breakpoints:

- **Hardware breakpoints** – These work by setting up control registers so that when a particular address in memory is read, written to or executed, a breakpoint interrupt is generated.
- **Software breakpoints** – These work by overwriting an instruction in the process memory image with the `INT 3` instruction, the opcode of which is the byte `0xCC`. When this instruction is executed a breakpoint interrupt is generated.

Imitate uses the software breakpoints method. This requires that when the instruction at the instruction pointer is overwritten to set the breakpoint, the previous byte at that location is saved. This is so that it can be restored and the original instruction be executed once the thread is continued. Imitate uses the `bpoint_byte` field of the `process` structure to store the original byte. It also stores the address of the location that was overwritten into the `bpoint_addr` field of the `process` structure. This is to enable a check to verify that the breakpoint was set by the kernel module and to ease the restoration of the stored byte from the breakpoint handler. Listing 6.5 shows the code that is used to set breakpoints. The `get_user` and `put_user` functions are used to read and write values from user-space respectively.

6.3.3 Breakpoint interception

To make use of the breakpoints set at pre-emption points, the breakpoint interrupt must be intercepted. This is achieved in Imitate by installing a hook into the interrupt 3 trap handler. This is the function that is called to handle breakpoint interrupts. The hook behaves in the same way as the context switch hook described earlier by calling a registered function in the Imitate kernel module.

When the breakpoint interrupt handler is called by the hook, it first checks that the breakpoint came from a replayed thread. If it did not, it simply `returns`.

In the case that the breakpoint was from a thread we are interested in, it carries out an additional check to verify that the breakpoint is one set by the kernel module. This is done by comparing the saved address in the `bpoint_addr` field of the process structure to the current value of the instruction pointer minus one. One needs to be taken away from the instruction pointer as it points to the location after the

INT 3 instruction was executed. If the breakpoint was not set by the kernel module, then the breakpoint handler returns. Otherwise, the following steps are carried out:

1. The saved byte in the bpoint_byte field of the process structure is restored to the address in the bpoint_addr field of the same structure. This is done to restore the original instruction that existed at that address before it was overwritten with the INT 3 instruction.

2. The instruction pointer is decremented so that it points to the beginning of the instruction at which the thread is to continue execution. This is the same point at which the thread resumed execution during recording when it was rescheduled.

3. The thread is removed from the scheduler run queue by setting its state to TASK_INTERRUPTIBLE, and the next thread in the schedule log is woken up.

4. The scheduler is invoked to select a new thread to run.

6.3.4 Context switches in the kernel

Breakpoints are an elegant way of finding the points to block a thread when replaying context switches that occurred in user-space. However, a different way of blocking the running thread is required when the context switch point occurred in the kernel (during a system call request).

When a thread is swapped out while in the kernel, the schedule log entry will store the instruction pointer to the next instruction to be executed when the system call returns. These context switch points can be identified by the stored instruction pointer in the schedule log having a value of SYSCALL_RETURN_POINT. The actual definition for this value is 0xffffe410, and is the point where all system calls return to in user-space after completion. A breakpoint cannot be set at this address because the area of memory in which this address resides is not writable. The address actually exists within a page of memory that is memory mapped into every process from a protected page in the kernel. This page contains code that is used to invoke system calls from processes. For a detailed explanation of this memory area, see [6].

The method used to replay context switches in the kernel is to prevent the return of the thread to user-space after the system call has been serviced, and then to wake the next thread that is to be run according to the schedule log.

The prevention of reentry into user-space is achieved using the syscall_lock_sem semaphore defined in the process structure. It is normally initialised to 1 on entry to the pre-system call callback whenever a system call is made. Then, whenever a thread is about to leave user-space from the pre- or post- system call call backs, a down operation is performed on the semaphore. This will succeed (as the semaphore's value is 1) and the thread would be allowed to return to user-space.

When a process is to be blocked in the kernel, instead of setting a breakpoint (which is impossible due the reason discussed above), the block_syscall flag of the process structure is set to 1. Then, when the next system call is made by the process, the block_syscall field of the process structure is checked. It will be found to be equal to 1, and so instead of initialising the semaphore to 1 on entry into the pre-system call callback, it is initialised to 0. In addition, the next thread to run according to the schedule log is woken.

Now, when the thread tries to leave the kernel, the down operation will block (as the semaphore's value is 0) forcing a context-switch. Later, when this thread is to be resumed according to the schedule log, an up operation will be performed on the semaphore to unblock it and the block_syscall flag will be set to 0.

6.4 Summary

This chapter has presented a detailed description of the method by which Imitate intercepts context switches in order to capture the thread schedule. It has shown how the number of backward control transfers made by a program in a scheduled quantum can be used to locate pre-emption points of threads to facilitate this. It has also discussed the use of the Dyninst instrumentation library to add a backward control transfer counter to process images in memory.

This chapter has also discussed the current partial implementation of schedule replay by setting and intercepting dynamic breakpoints in order to force pre-emption at the points stored in the schedule log.

Chapter 7

Log buffers

Imitate makes extensive use of buffers for logging and replaying processes. The main purpose of the buffers is to simplify the interaction between the kernel and the user-space monitors, however they also benefit performance as they avoid unnecessary copying between kernel and user space. This chapter discusses the implementation of these buffers and how they are used to interact with the user-space monitors.

7.1 Overview

There are two log buffers in Imitate, the system call log buffer which is used to store and retrieve system call data and the schedule log buffer which is used to store and retrieve thread schedule data. Both of these buffers exist for each monitored process and are both shared between all threads of that process.

The content of the both of the log buffers are organised into entries (records). During recording, a new system call log entry is created every time a system call completes and is ready to have its data recorded. New schedule log entries are created whenever a thread is swapped out for another from the same process. The log entries are of the following format:

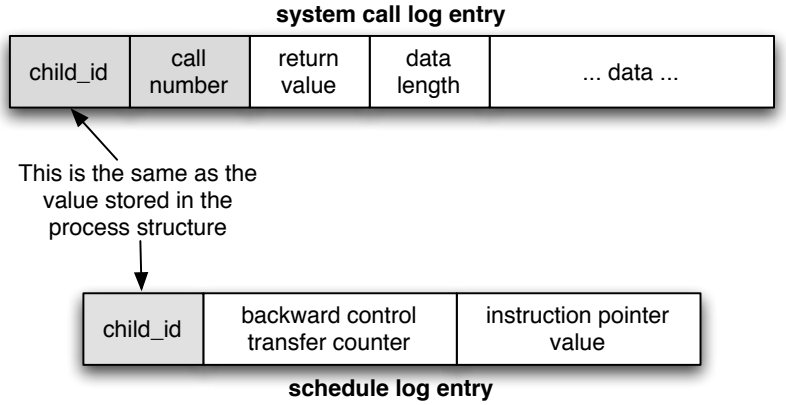

Figure 7.1: The structure of the system call and schedule log entries

When the buffers become full, the user-space monitor process is notified and they are flushed to disk.

During replay, the monitor process is used to populate the buffers, using the log files generated during record, which are then consumed by the kernel module. When the buffers become empty, the user-space monitor is notified to refill them.

7.2 Initialising the buffers

When a monitor process starts, it creates a connection to the kernel module by opening the device that the module presents itself as. It then notifies the module that it is a monitor by sending a `MONITOR` message using the `ioctl` system call. This causes the module to allocate a `monitor` structure and the buffers in preparation for recording or replaying. The `monitor` structure is used to store information pertaining to the monitor, such as the pointers to the buffers, and also the process it is monitoring (i.e. the recorded or replayed application).

The two buffers are allocated using two different allocation methods within the kernel. The system call buffer is allocated into kernel virtual memory using the `vmalloc` method as it needs to be large and does not have any requirements of being contiguous in real memory. The schedule buffer on the other hand does not have such large memory requirements and hence is allocated into real memory by the `kmalloc` method. This provides the advantage that schedule log is faster to access since the paging unit is not required to do any translation. For an explanation on why large allocations cannot occur in real memory see [8, 12].

7.3 Memory mapping into user-space

Once the buffers have been allocated, they are still only accessible to the kernel module itself. To make them accessible to the monitor process they must be mapped into its address space.

This memory mapping is made by issuing the memory map system call, `mmap`, from the monitor process to the kernel module using the handle obtained when the connection to the module was opened. When this call is received by the module, it invokes the memory map handler whose responsibility it is to perform the mapping by creating page tables and page table entries.

The page tables are created differently depending for the two buffers. The simpler of the two methods is to build the entire table at once, and is used to perform the mapping for the schedule buffer. This is a simple case of calling the `remap_pfn_range` function supplied by the kernel on the virtual memory area to be mapped:

```
remap_pfn_range(vma,
    vma->vm_start,
    virt_to_phys((void*)((unsigned long) monitor->sched_data)) >> PAGE_SHIFT,
    SCHED_BUFFER_SIZE,
    PAGE_SHARED)
```

Listing 7.1: Using the `remap_pfn_range` function to memory map the schedule buffer.

Unfortunately, this method can only work for pages that are allocated contiguously in real memory by using `kmalloc`. Since the system call log buffer is large and is allocated in virtual memory, the pages for it are not contiguous in real memory and the more complex method of mapping pages on page faults is used.

Linux provides a relatively simple way of enabling this by allowing the module to register a page fault handler for the virtual memory area (VMA) when an memory map request is received. Imitate

```
1  static struct page *vma_syscall_nopage(struct vm_area_struct *vma,
2      unsigned long address, int *type)
3  {
4      unsigned long offset;
5      struct page *page = NOPAGE_SIGBUS;
6      char *syscall_data = processes[current->pid]->monitor->syscall_data;
7
8      offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
9      if (offset > SYSCALL_BUFFER_SIZE)
10         goto out;
11
12     page = vmalloc_to_page(syscall_data + offset);
13     get_page(page);
14     if (type)
15         *type = VM_FAULT_MINOR;
16
17     out:
18         return page;
19 }
```

Listing 7.2: The system call buffer page fault handler `vma_syscall_nopage`

registers the `vma_syscall_nopage` (Listing 7.2) method by supplying its address to the page fault hander operation of the VMA.

Now, whenever a page fault occurs (as the page has not been mapped), the `vma_syscall_nopage` method will be called to create the page table entry and perform the mapping. This requires several steps:

- The offset into the buffer for the address requested must be calculated (line 8).
- The appropriate page structure must be retrieved for that offset (line 12).
- The reference counter for the page must be incremented to make sure the page is not removed while the mapping exists (line 13). This is necessary because whenever a virtual memory area is unmapped, either by a request or a process exiting, the this counter is decremented, and when it reaches 0, the page will be marked as free. This will make it available for other processes or the kernel to use and could result in the it being overwritten even though it is still in use by the module.
- Finally, the page table entry for the mapping must be created by returning the page structure to the higher-levels of the virtual memory subsystem (line 18).

7.3.1 Selecting the buffer

Since there are two buffers that are memory mapped, a method by which to decide the buffer is being mapped must be implemented as there is no way select the buffer via a parameter in the `mmap` system call.

In Imitate, the method used is to keep track of how many times the memory map system call has been made on the module by each monitor process and decide the buffer based on its value whenever a request is made. The number of memory map requests is stored in the `mmap_select` field of the `monitor` structure and initialised to `0` when the `MONITOR` ioctl message is received. In, the current implementation

the first mmap call by a monitor process will always map the system call log buffer, while the second will map the schedule log buffer. All subsequent requests will be returned an error.

7.4 Flushing and refilling

When a buffer becomes full during recording or empty during replay, a message is sent to the user-space monitor process by replying to the message request sent by the monitor. Understanding how this message is sent requires first understanding the message loop of the monitor process.

The message loop is actually a `while` loop that continuously makes a message request call to the kernel module using the `ioctl` system call. When this call is received by the kernel module, an `up` operation is performed on the `data_write_complete_sem` which is used to store the fact that the monitor is waiting for data (i.e. the previous message has been serviced). Then, a `down` operation on the `data_available_sem` semaphore is carried out. When the kernel module does not have any messages to send to the monitor, this operation will block until a message becomes available. Both of these semaphores exist in the `monitor` structure.

Now, when a message needs to be sent to the monitor process, in this case a refill or a flush message, the process is unblocked in the following way:

- A `down` operation is carried out on the `data_write_complete_sem` semaphore to store the fact that a message is about to be generated and that the monitor has not yet serviced it.
- The message to be sent is copied into the monitor's address space using the `copy_to_user` function provided by the kernel.
- The monitor unblocked by carrying out an `up` operation on the `data_available_sem` semaphore.

The monitor process then carries out the action requested by copying data from the buffer to the log file on disk (in the case of a flush message), or copying the data from the log file into the buffer (in the case of a refill message). Once this is done, it again makes the message request call letting the kernel module to know that it is ready to service another message.

7.5 Concurrent access

The buffers are shared per monitored process, and so they must be protected from concurrent accesses from the different threads of the process to avoid corruption. This is not an issue for the schedule log buffer as it will only be accessed from the scheduler and thus will be protected by the process queue locks. However, the system call log must be protected explicitly by the module.

Imitate uses a semaphore, `buffer_sem`, defined in the `monitor` structure to protect the system call log buffer. During recording, this lock is all that is needed. However during replay, there is a need for something more as several threads could be trying to access the system call log buffer and may do so in the wrong order. This would lead to them obtaining incorrect data and thus incorrectly replaying the system call.

To prevent this erroneous behaviour, Imitate suspends any threads that are trying to read the system call log buffer before their turn. It achieves this by checking the call number and the thread ID (`child_id`) retrieved from the log before a system call is replayed. If the thread ID of the thread making the system call does not match that which is expected in the log, the thread is blocked and placed on a blocked threads queue. The method of blocking used is to set the task state to `UNINTERRUPTIBLE` to remove it from the run queue and then invoking the scheduler to select a new process.

Later, when the next system call log entry has a thread ID that matches that of the blocked thread, it is woken up (by calling `wake_up_process` function) and removed from the blocked threads queue. This enables it to continue replaying by reading the log. The process of waking up the thread is carried out in the `seek_to_next_syscall_entry` function. This is called from the `replay_value` macro and therefore, blocked threads are only woken after the correct thread has replayed the current system call.

In the case that a system call is made that was not saved in the log, the process making the call is sent the `KILL` signal to terminate it. This is because it is most likely that the program has changed between recording and replaying as otherwise the call would not be made since the program behaviour is deterministic during replay.

7.6 Summary

This chapter has discussed on the use of buffers within Imitate to ease sharing data between kernel and user space. It has discussed the two types of buffers used in Imitate and how the memory map method has been used in order to share them with the user space monitors. The mechanism by which messages are sent to flush these buffers to disk has also been shown. Finally, the method by which concurrent access to these buffers is prevented has been presented.

Chapter 8

Testing

During development, continuous testing was used. Whenever a new feature was added or a portion of code that could be tested was written, a test was done to check for correctness.

8.1 System call record and replay

System call record and replay was to be implemented in Imitate, despite the fact that schedule replay was the highest risk endeavour. This was due the dependence of system call interception for recording and replay of the schedule.

8.1.1 The `ls` and `date` commands

The `ls` and `date` commands where the main utilities that were used to test system call record and playback for single-threaded processes. The utilities were chosen as they both return non-deterministic data every time, and thus could be easily checked for accurate replay.

Two tests were performed during development using these utilities:

- **Verifying that recording does record system call output and replaying a log does indeed replay the recorded result.** The `date` command was recorded by Imitate and its output was noted. Then, it was invoked again without being recorded by Imitate and its output was compared to the recorded run. They are expected to be different as the system state had changed and result was not being replayed. Finally, the log recorded was used to replay. This output is expected to be the same as the output when recording.
- **Verifying that recording a process does not interfere the process' behaviour.** The `ls` command was run on a directory and its output was noted. Then, the same command was recorded under Imitate without changing the directory contents and the output was checked to make sure they matched. This test was repeated again after adding a file to the directory.

8.1.2 The `mtio` program

To test the record and replay of system calls in a multi-threaded context, a simple program was written and tested with Imitate. This program starts several threads that read a single file and behaves as follows:

1. It takes a file name as an argument.
2. On starting, it creates 5 threads. Each thread is assigned an ID between 1 and 5. These threads run concurrently and each carrying out the following operations:
   (a) It opens the file supplied as the argument to the process. Each thread gets its own file descriptor for the file.
   (b) It seeks (10 $\times$ ThreadID) into the file.
   (c) It reads ThreadID number of bytes from the file.
   (d) It prints the bytes read and closes the file.

Due to the fact that each thread makes several system calls on the same file, the order in which the threads are scheduled is completely unpredictable and is usually different for each run. This made it a very good test to verify that the relative ordering of system calls made by each thread can be recreated during replay and in fact revealed a subtle bug that was later fixed. This bug caused newly threads to be able to execute some instructions or system calls prior to having their `process` structure set up and installed during both record and replay. It was fixed by starting new threads in the `STOPPED` state by altering the parameters to the `clone` system call, and then letting it continue only after the process structure was set up.

8.2 Schedule record and replay

While schedule replay is not complete, it is still possible to test schedule recording by hand. The program used to test schedule recording is called `threads`. It works by starting two threads that each increment a counter 50,000,000 times and print its value along with the thread ID whenever it reaches a multiple of 5,000,000. The fact that each thread increments the counter so many times makes it CPU bound, reducing the chance that it will be swapped out due to a system call, and makes sure that threads will be pre-empted in between the `for` loop that does the incrementing.

Once the program completed recording, it was used to compare the output of the program to the recorded schedule to by comparing the order of the printed thread IDs to the order of the child IDs stored in the schedule log. They were found to be the same and hence the schedule recorded was correct. To make sure that the stored instruction pointers were correct, they were checked against a disassembly of the program generated by the `objdump` command.

8.3 Operating system stability

The nature of Imitate means that it makes some very invasive changes to the operating system. This runs the risk of corrupting kernel data structures and destablising the system. To avoid this several tests were done to verify that the system remained stable with Imitate loaded.

- **Verify that Imitate does not affect unmonitored processes**. This was tested by leaving Imitate installed on the system and using the system for standard daily tasks without invoking recording or replay.
- **Verify that a process with a PID that was previously assigned to a recorded or replayed process was not affected**. Linux assigns process identifiers in a sequential fashion an then loops around once the `PID_MAX_LIMIT` value is reached. This was exploited for this test by first recording and replaying a process and noting the PIDs that were assigned to it. Then, short running processes were started in a loop and when one was assigned the same PID as either the recorded or replayed process, its output was checked to see that it behaved correctly.

- **Verifying no kernel memory is leaked**. Kernel memory is a precious resource, and so to check that any memory allocated to Imitate was subsequently freed, debug statements were added to each allocation and deallocation point within the module. Then, the `ls` command was record and replayed. The debug log was then checked using a small Perl script to make sure each allocation had a matching deallocation.

8.4 Summary

This chapter has presented the methods used to test the implementation of Imitate by testing the correctness of system call record/replay and the schedule capture individually. It has also shown how operating system stability was tested in face of the invasive changes made by Imitate.

Chapter 9

Evaluation

This chapter discusses the successes and failures of this project. It presents the challenges that were faced during implementation, gives experimental performance measurements and states the limitations of the project in its current form.

9.1 Overview

Imitate was a very ambitious project. To the best of my knowledge, there have been no attempts to create a generic multi-threaded software debugging platform that has generated traces that can be used for both analysis and deterministic replay on a modern operating system. Moreover, none have attempted to provide one while requiring no additions to the binary of a program on disk. The implementation of Imitate shows that such a framework is possible to implement, albeit difficult, due to the required attention to minute details.

In relation to the initial goals set out for this project, Imitate can be considered a relative success. Although the goal of schedule replay was not met, the implementation of Imitate has resulted in the foundation for future work. It has also carried out a detailed investigation of the issues with implementing a record and replay framework with goals similar to that of Imitate, many of which will not be revealed by a feasibility analysis, thus allowing future projects to supply more complete implementations.

9.2 System call recording and replay

Imitate achieves the goal of providing accurate system call recording and replay for both single and multi-threaded applications in an elegant way without the need to patch the source of the kernel.

The mechanism by which system calls are intercepted (see section 5.1) has enabled the development of a flexible and easy to use interface for the interception of calls currently not implemented in Imitate. This implementation can also be used to provide interception capability to other projects as it is a general solution that is not tightly coupled to Imitate. When a system call is to be intercepted using this interface, only two functions need to be created: the pre-system call handler and the post-system call handler. The implementation of these functions is greatly simplified by the interception interface as they do not need to do any home-keeping. As an example, the implementation for these two handlers for the open system call used in Imitate is given in Listing 9.1.

```
void pre_open(syscall_args_t *args) {
    process_t *process = processes[current->pid];
    syscall_log_entry_t *entry;

    if (replaying(process)) {
        entry = get_next_syscall_log_entry(__NR_open);
        replay_value(process, entry);
    }
}

void post_open(long *return_value, syscall_args_t *args) {
    process_t *process = processes[current->pid];

    if (recording(process))
        write_syscall_log_entry(__NR_open, *return_value, NULL, 0);
}
```

Listing 9.1: Intercepting the `open` system call.

Considering the implementation details discussed above, it is reasonable to conclude that the system call interception mechanism is optimal, as it enables support for new system calls to be added in the future in a modular fashion. Clearly, support for all 312 calls provided by Linux would not have been possible in the time allocated for this project and so this interface is necessary.

9.3 Thread schedule recording

While replaying of the thread schedule is incomplete, the thread schedule recording is functional. Unfortunately, Imitate's implementation is currently very heavyweight due to the use of Dyninst for instrumentation and could be significantly optimised by use of a lighter-weight mechanism. Such a mechanism could use a kernel module that can be sent messages by a user-space process to add instrumentation to the process image of another process. This would be significantly faster as there would be no need to attach to the process using the `ptrace` system call. Instead, the kernel module could instrument the process image directly. This would also remove the issues caused by Dyninst (see below) as a result of the `ptrace` system call and allow debuggers to attach.

This is not to over-criticise the implementation that Imitate provides as it still generates accurate traces which can still serve a useful purpose in trace analysis and a replay once the schedule replay portion is complete. One method by which they can be used for analysis is to obtain thread schedules for both successful and failure runs of a process an then to do a comparison. The differences in the instruction pointers of at pre-emptions could be used to gauge an approximate location of the error and number of backward control transfers can be used to show the execution chain that caused it.

9.4 Performance

This section presents some experimental performance readings that were taken during recording and evaluates the overhead created by Imitate. The programs used for measuring the performance were chosen strategically to test the overhead of I/O bounded and CPU bounded programs individually. Then finally tests were done on real software to measure overhead in a real world context.

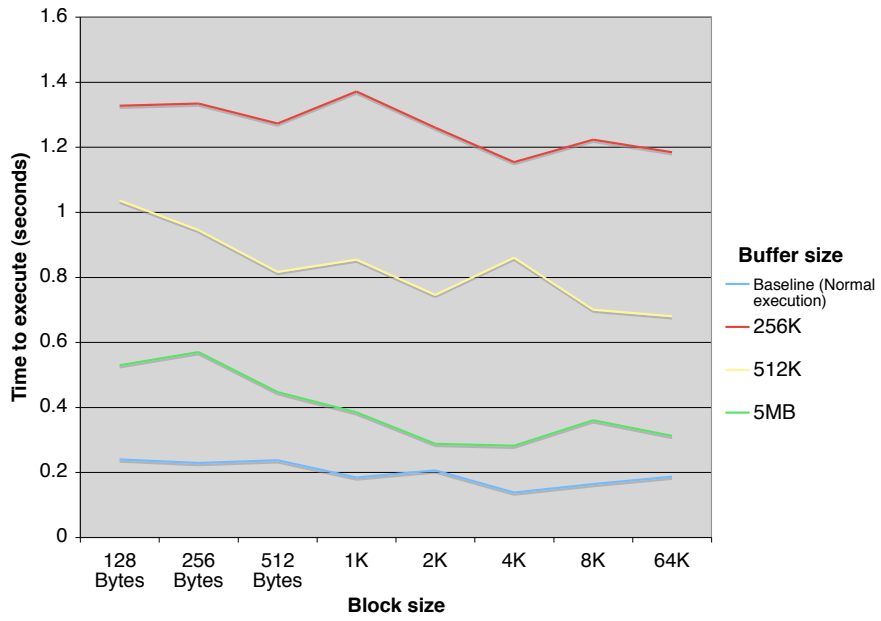

Figure 9.1: The performance of recording the `dd` command under Imitate with different system call log buffer sizes.

9.4.1 The `dd` command (I/O Bound)

The `dd` command was used to measure the performance overhead of Imitate on I/O bound programs. This command is used to convert and copy files and is flexible in the way it does so as it can be supplied the block size in which the file is copied – so for example giving a block size of 512 means that the file will be copied in 512 byte chunks. This gives a good way to control the number of system calls made and thus provides a simple way to measure the effect of system call recording.

In this test, this command was tested under 3 different buffer sizes for the system call log buffer while performing a copy of 512KB from the random number generator device `/dev/urandom` to a file to a file on disk. The results are given in Figure 9.1. To give an accurate picture of the actual overhead, the results are given without the time required to instrument the program (which usually takes between 12 and 15 seconds on average for the `dd` command) as this will be negligible for long running processes as it will only ever be carried out once.

As expected, this test has shown that system calls that generate lots of data (in this case, `read`) result in a significant loss in performance due to increased I/O when flushing the buffers to disk. They have also demonstrated the performance to be gained from increasing buffer sizes to have them be flushed less often. In fact having the system call buffer size increased from 256KB, which results in the buffer being flushed 8 times in this test, to 5MB, which causes it to only be flushed once at the end of execution, reduced overhead from a 5x loss in speed to an average of 2x loss.

This test has also shown reduction in performance caused by increasing the number of system calls made. This is due to the kernel spending more time checking arguments, setting up data structures and saving registers whenever a system call is made. Imitate adds to this loss in performance due by having to call nested functions in the intercept, which requires setting up stack frames and local variables.

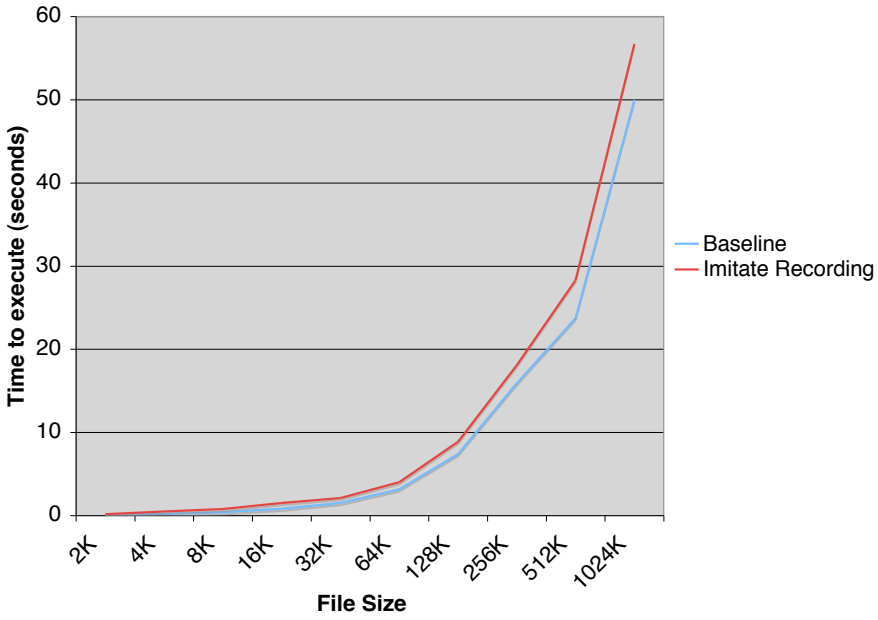

Figure 9.2: The performance of recording GZip while compressing different sized files.

9.4.2 GZip (CPU Bound)

GZip is a popular compression algorithm commonly used on the internet to compress files to speed up transferring files between computers and is usually I/O bound, however a CPU bound version has been developed as part of the SPEC2000 benchmark suite [4] that does all compression to memory instead of disk. This is the version used for this test.

The test was performed by using GZip to compress different sized files. These files were portions of web server logs. As with the previous test, instrumentation time is not included in the measurements (which usually takes between 13 and 14 seconds on average for the version of GZip used). The results are given in Figure 9.2.

As can be seen in the graph, the performance reduction during recording for CPU bounded applications is very small. In the case of GZip, it was only between 1.1 and 1.2 times slower, and most of this performance loss was due to the writing the system call log after the test was complete. This is a very positive result showing that Imitate can be used to record the executions of CPU intensive applications without a prohibitive performance loss. However, one thing to note is the significant slowdown caused as the file size increases, due to more I/O caused by larger files.

9.4.3 GCC (Average application)

The previous tests have identified that Imitate significantly degrades the performance of programs that carry out many system calls. In order to get an idea of how this would affect an average application which has a more balanced split between CPU and I/O time, a test was performed on recording GCC (GNU C Compiler).

The test was performed by compiling files of different sizes. All of these files were obtained from the latest "vanilla" kernel sources. Again, the instrumentation time was removed from the results (which

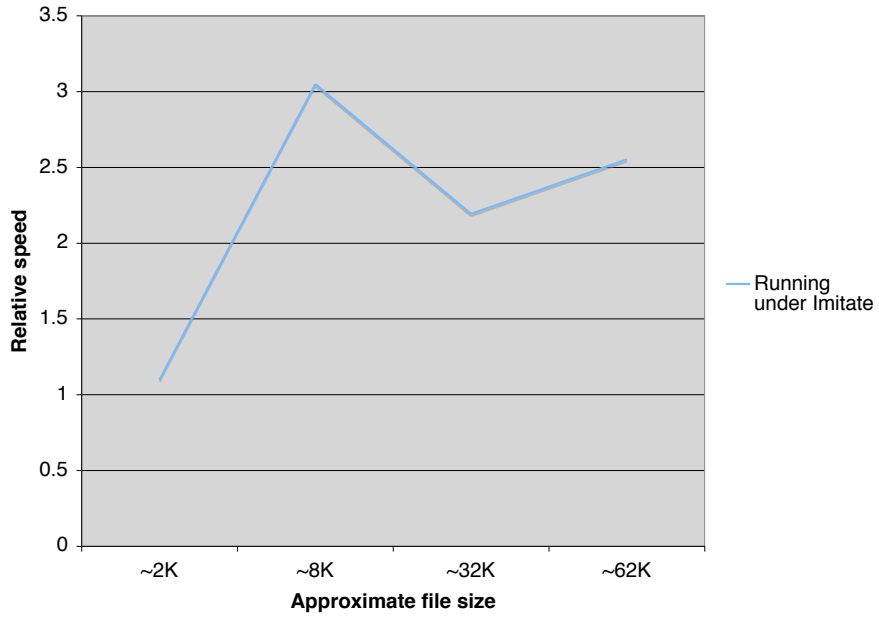

Figure 9.3: The overhead of Imitate on GCC when compiling files of different sizes

usually takes between 13 and 14 seconds for GCC). Figure 9.3 shows the results.

As can be seen from the graph, the results are inconclusive and further investigation is needed.

9.5 Challenges

This project was an extremely challenging undertaking. All throughout the implementation many problems were faced and more than 50% of the implementation time was used to solve them. The most difficult issues that were eventually solved are discussed below.

- **System call interception without stack modification**. Initially, it was not known that the stack could not be modified and so the system call chaining method was used for interception. Problems with the stack were only discovered after the interception of the `clone` system call which caused the system to crash with the chaining method. By this time, most of the system call record/replay code was complete, but had to be rewritten after this discovery.

  Writing the `syscall_intercept` function was very difficult as in addition to maintaining the stack, it had to provide a good interface for system call handlers to be implemented. This is to make sure that currently unintercepted system calls could have their intercept functions implemented easily. It required extensive testing to make sure it was correct and would work for all system calls.

- **Flushing buffers to disk and refilling buffers from disk**. While this is a simple case of sending a message to the user-space monitor process, the implementation is not trivial as it involves a two semaphore locking scheme to make the messaging process synchronised. Due to this, extensive verification was required to ensure it not cause deadlocks within the kernel module or the monitor process.

- **Instrumenting the backward control transfer counter**. Although Dyninst significantly eased the

instrumentation of a process, it came with its own set of issues. Dyninst uses the process trace system call `ptrace` to attach to and control the process to be instrumented. This prevented debuggers from attaching, as they also attach to processes in the same way, and so debugging the instrumentation was very difficult since there was no way to check if an instrumented process was executing the number of backward control transfers it was reporting.

In addition, some functions cannot be patched by Dyninst (as discussed in subsection 6.1.4). To skip instrumenting these functions required finding the library that contained these functions and narrowing the set of functions that could not be patched. This was done by testing each loaded library in turn and then guessing the functions that may be causing problems based on the functions that were used in the program. To obtain this list of functions, the `ltrace` tool was used.

- **Memory mapping the backward control transfer counter**. The instrumentation portion of this was relatively simple, however the kernel portion was more difficult since it is only possible to map on page boundaries. Since the memory required for the counter is so small, this meant that it could be allocated anywhere in the memory that was mapped into the user process. Debugging this required watching the memory that was mapped using specially crafted program. To work around this issue, the counter was added as the first entry in the `process` structure which was then made to be page aligned by allocating it an entire page to it rather than the size of structure. Having the counter as the first entry means that it is always the first thing in the page when mapped, thus removing the need to calculate an offset.

9.5.1 Schedule replay

The most difficult portion of this project was, by far, the implementation of schedule replay. There were very many unanticipated problems that caused development to progress very slowly and is the reason that the implementation is not yet complete. These problems are outlined below.

- **Dynamic libraries are mapped into different addresses on every run**. This problem makes it impossible to set breakpoints at the instruction pointer stored in the schedule log file because, if the recorded pre-emption point was within a library, it would not be valid during replay as the library could be mapped into a different area in memory.

  Currently, an attempt to reduce this effect has been made by overriding the parameters to the `mmap` system call (which is used to map dynamic libraries into memory) during replay by giving a suggested start address for the memory mapped region. The address supplied is that which was assigned to the region during recording. Unfortunately, this is only a suggestion and the kernel is allowed to ignore it and memory map into any address it decides. Therefore, breakpointing still fails in most cases.

- **Dyninst receives `SIGCONT` signal when continuing new threads**. When new threads are created, they are created in the `STOPPED` state in order to allow the kernel module to set up the necessary `process` structures. In order to let the threads continue after this is done, a `SIGCONT` signal has to be sent to them. However, Dyninst catches this signal and acts upon it. This causes the program to crash with an error from the Dyninst runtime. A solution is yet to be found for this problem.

- **Dyninst crashes after instrumenting some programs**. This point also applies to instrumenting during recording. When certain multi-threaded programs are instrumented Dyninst crashes immediately after inserting the instrumentation. Furthermore, it crashes in a non-deterministic manner ranging from not being able to send a signal to the process, to generating a segmentation fault. It is thought this is related to the way in which Dyninst attaches to the process in order to instrument it, however further investigation is needed.

9.6 Limitations

In addition to non-functional schedule replay, the current implementation of Imitate suffers from a number of limitations that prevent it from being useful for debugging large real world applications. Fortunately, most of these limitations can be overcome with time and as such, do not degrade the achievements made. Below is a list of the main limitations of Imitate in its current form:

- Only a subset of the system calls available are recorded and replayed. However, as described above, the way in which the interception of calls is structured enables additional calls to be supported very simply and in a modular fashion.
- Recording the schedule for an application that makes the `execve` call is not supported. This is due to the need to re-instrument the application whenever an it replaces its process image. This problem can be overcome by using the Dyninst `ExecCallback` in the `patcher` program so that it instruments the new process image.
- Programs that apply just-in-time compilation such as Java and Mono cannot be recorded or replayed. This is due to the fact that instrumentation cannot be added until the code of the program is compiled in memory. Since there is no way to get notified when a piece of code is compiled using the jitter (just-in-time compiler), it is not possible to instrument it to record the schedule.
- Long running applications generate massive traces. This is due to the large amount of data that system calls are able to generate. It is possible to reduce trace sizes by implementing compression in the monitor process. Another way this could be made possible is to allow the system to ignore data generated by certain calls. For example, if it is known that a file will always contain the same data, any `read` call's on that file do not have to have the data read stored in the log.

9.7 Summary

In this chapter, we have discussed the effect of Imitate on the performance of processes that are being recorded and replayed. As expected, Imitate reduces the performance of applications that generate a lot of data as a result of system calls. It has also shown that CPU bounded applications can be recorded successfully with as little as a 1.1x degradation of performance.

In addition, this chapter has discussed the most difficult challenges faced during implementation and how they were overcome. Also, it has presented the reasons for the incomplete implementation of thread schedule replay.

Chapter 10

Conclusions

This report has presented a description of a new framework that has laid the foundation for a tool that can be added to a software developers arsenal which can assist in debugging multi-threaded software.

This project has provided an implementation that is able to record execution traces of programs, regardless of the programming language they were written in or the API's they use, which can subsequently be used to debug a program by deterministic replay. In addition, the execution traces generated have been generated in such a way as to allow for automated debugging via trace analysis. Finally a partial, but almost complete, implementation of a replay system has also been provided.

10.1 Achievements

The main achievements of this project are:

- **An investigation into the use of instrumentation for the recording and replay of the execution of a process in a modern operating system**. Modern operating systems are extremely complex pieces of software and modifying them to include new functionality such as recording and replay is no trivial task. They support features such as dynamic linking and memory mapping which makes adding instrumentation to support recording and replay of a thread schedule a non-trivial process. This project has shown that using instrumentation for recording and replay is possible, but much more difficult than a preliminary feasibility analysis would reveal.
- **A solution for capturing the thread schedule using instrumentation has been developed**. Imitate has used Dyninst to insert instrumentation into a process image in memory and provided a kernel module and a monitor process that is able to record the thread schedule with the assistance of the instrumentation. A system by which the schedule can then be replayed has been devised and partially implemented. It is only incomplete due to unforeseen problems caused by a modern operating system.
- **An implementation of a record and replay system for I/O bound programs has been created**. Despite the incomplete implementation of schedule replay, it is possible to deterministically replay I/O bound multi-threaded programs. Being I/O bound means the pre-emptions are most likely to occur during the servicing of an I/O system call. Combining this fact with the fact that Imitate makes sure that system calls are executed in the same order as stored in the system call log during replay, means that different threads making these calls will have them serviced in the same order. This results in a deterministic replay of the implied thread schedule.

- **A simple and intuitive method of system call interception has been developed**. As part of the requirement of this project required system call interception, a simple, reusable method by which to do this has been developed. Any future work that requires this capability will be able to reuse the code provided in this project and no longer have to deal with the intricacies involved such as maintaining the stack.

10.2 Future work

Imitate has a lot of scope for future work. In addition to completing the implementation of schedule replay, there are many additions and extensions that could be made to it more useful or increase performance. The list below suggests some ideas.

- **Implementing support for more system calls**. Currently, only 15 of the 300 or so system calls are recorded and replayed by the system. Increasing the number of supported calls would greatly enhance the usefulness of Imitate by allowing more applications to be recorded and replayed.
- **Double buffering**. Adding double buffering to both the system call log buffer and the schedule log buffer would greatly increase performance and make the system more suitable for use in production for debugging already released applications.
- **Remove dependency on Dyninst**. Dyninst was used in this project primarily for speed of development. It is a very heavyweight solution to a relatively simple instrumentation problem. Removing the dependency on it would increase performance and simplify the implementation. A solution not requiring the use of the `ptrace` system call would also fix all of the problems associated with instrumentation that are currently present.
- **Add support for caching of instrumented programs and libraries**. Caching the instrumented program would enable a quick startup time during replay and future recordings as there would be no need to add instrumentation after the first recording.
- **Increment backward control transfer counter without locking**. To make sure that the backward control transfer counter is incremented atomically, a lock is used in the current implementation. The need for this lock could be removed by using an atomic increment instruction, and would greatly benefit performance. This would also mean that it would be unnecessary to load in the `pthread` library into the instrumented process and that the instrumentation could be applied to the locking functions as there would be no risk of infinite recursion.
- **Implement checkpointing**. Checkpointing would reduce the time required to debug by replay as it would no longer be necessary to replay the entire execution trace from the beginning. However, checkpointing is a costly operation as care would need to be taken to avoid degrading performance to an unacceptable level.
- **Addition of support for signals**. Signals are currently unsupported by Imitate. Adding this support would increase the number of applications that could be recorded and replayed as many daemons (background processes) use them for interacting with the user on a basic level.
- **Use KProbes to install hooks**. Imitate currently uses a source code patch to install the necessary hooks into the scheduler and breakpoint handler. This could be by using the KProbes kernel debugging system thus negating the need for a source code patch and making Imitate easier to install.

Bibliography

[1] Kernel newbies. http://www.kernelnewbies.org.

[2] Linux cross-reference. http://lxr.linux.no.

[3] Linux loadable kernel module howto. http://tldp.org/HOWTO/Module-HOWTO.

[4] Standard performance evaluation corporation. http://www.spec.org. 58

[5] Undo software. http://undo-software.com. 7

[6] What is linux-gate.so.1? http://www.trilithium.com/johan/2005/08/linux-gate. 42

[7] A perturbation-free replay platform for cross-optimized multithreaded applications. In *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, page 10023.1, Washington, DC, USA, 2001. IEEE Computer Society. 17

[8] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005. 46

[9] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000. 12, 35

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press. 9

[11] Jonathon Cooper. Detailed program tracing and replaying. Master's thesis, Imperial College London, 2003. 7

[12] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. 46

[13] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, New York, NY, USA, 2006. ACM Press. 20

[14] Alessandro Giusti. Delta debugging. http://programming.newsforge.com/article.pl?sid=05/06/30/1549248. 9

[15] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 146–155, Lisbon, Portugal, September 2005. 11

[16] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, London, UK, 2000. Springer-Verlag. 8

[17] Ekaterina Itskova. Echo: A deterministic record/replay framework for debugging multithreaded applications. Master's thesis, Imperial College London, June 2006. 2, 21, 24

[18] Guy Keven. Multi-threaded programming with posix threads. http://www.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html.

[19] David Kline. Debugging multi-threaded applications. http://blogs.msdn.com/davidklinems/archive/2005/09/27/474653.aspx. 7

[20] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987. 17

[21] Jeff Magee, Jeff Kramer, Robert Chatley, and Sebastian Uchitel. Ltsa - labelled transition system analyser. http://www.doc.ic.ac.uk/ltsa. 5

[22] Sun Microsystems. Debugging a program with dbx. chapter 11. http://docs.sun.com/source/819-3683/MT.html. 7

[23] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005. 10, 14, 15

[24] Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005. 10

[25] John K. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX '96: Technical Conference*, 1996. http://home.pacbell.net/ouster/threads.pdf.

[26] 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, New York, NY, USA, 1996. ACM Press. 2, 10, 17, 24, 35, 40

[27] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, New York, NY, USA, 2005. ACM Press. 16

[28] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997. 8

[29] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004. 19

[30] Wikipedia the free encyclopedia. Debugging. http://en.wikipedia.org/wiki/Debugging. 6

[31] Larry D. Wittie. Debugging distributed c programs by real time reply. *SIGPLAN Not.*, 24(1):57–67, 1989. 13, 14

[32] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135, New York, NY, USA, 2003. ACM Press. 14

[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. 9

[34] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press. 9