Imperial College London

Department of Computing

Deterministic Replay of Concurrency Bugs

by

Hugo Gibson

Submitted in partial fulfilment of the requirements for the MSc. degree in Computing Science of Imperial College London

September 2012

Abstract

With the onset of multi-core processors, writing software that takes advantage of such processors has become an essential part of modern software development. Unfortunately, writing concurrent programs is a difficult task and often produces bugs which will not be found even after extensive testing. There are many scenarios, such as data races and deadlocks, that may go unnoticed until a system has gone to production, potentially leading to catastrophic results. To overcome this, many tools that trace concurrent execution have been created to debug concurrent software.

An existing tool, KLEETHREADS, builds on the KLEE symbolic execution engine, using symbolic execution to trace concurrency bugs and outputs a thread schedule to show the conditions for a given bug to arise.

Concurrency bugs are notoriously difficult to reproduce, since many bugs are dependent on a particular thread schedule that occurs only very rarely. The aim of this project has been to provide deterministic replay for concurrent C programs.

We introduce a tool, P-Rep, which takes a schedule generated by KLEETHREADS, or a hand-written schedule, and forces that thread schedule to be followed. KLEETHREADS achieves its schedule via symbolic execution. This tool provides a replay feature via function interposition.

P-Rep works with the original Pthreads API and requires no extra-intervention on the part of the programmer. It incurs a minimal overhead and will follow any generated schedule accurately.

The Pthreads API is very large. As a result of this, implementing a replay feature would have been beyond the scope of this project. Instead, we have focused on four key aspects on the API: thread creation, thread termination, mutex locking and unlocking, and condition waiting. These are the most widely used parts of the API and many programs only make use of these features.We have made a higly robust deterministic replay tool for these parts of the Pthreads API.

We have implemented the final tool by using function interposition. P-Rep loads our shared library and executes a program that uses Pthreads while forcing the thread schedule it has been passed to be followed. It operates in four modes. In its first mode, P-Rep follows the schedule and either hangs or finishes execution at the end of the schedule. In its second mode, P-Rep follows the schedule to its end and then stops execution. In its third mode, P-Rep follows the schedule until its end and then resumes normal execution with no further intervention. In its fourth mode, P-Rep provides a detect and record feature. It will detect a deadlock and output the schedule that was followed to reach that deadlock. These four modes of execution are useful since they allow the programmer to observe a multitude of different behaviours and also record deadlocks.

We have found that P-Rep works excellently under a variety of conditions and all of the functions we have chosen to override work as expected. We have tested P-Rep on a wide variety of different concurrent C programs and fed it a range of schedules generated by KLEETHREADS, and found that it provides deterministic replay very well.

Acknowledgements

I would like to thank my supervisor Alastair Donaldson for his continued support and advice. I would also like to thank Paul Thomson for his creation: KLEETHREADS.

Contents

Listings

List of Tables

Chapter 1

Introduction

Multi-core processors have moved from specialist computers and large-scale clusters to desktop and laptop computers and are becoming the norm in all areas of computing. The need to make use of multiple processors in software development has become essential and this has led to many new types of bugs emerging.

1.1 Deterministic Replay

The aim of this project has been to create a tool that provides deterministic replay for concurrent software written using the Pthreads API. The Pthreads API provides a set of functions for concurrent C programming. It is a low-level implementation and that aims to provide concurrency features such as locks and conditions variables that programmers can use to write concurrent software. We chose the Pthreads API since there are few concurrency tools that specifically target it.

Deterministic replay is used to force a program to follow a certain path of execution. By reproducing a particular execution trace, a programmer is able to see an error that might not occur under normal conditions. Many bugs in concurrent software rely on a very specific thread schedule before they will occur. As a result of this, many such bugs may not appear even after thousands of executions. Deterministic replay will force a particular schedule to be followed and reproduce the circumstances that gave rise to the bug, thus allowing the programmer to quickly and accurately debug their software. The programmer can run the schedule over and over, knowing that ther final program state will always be deterministic.

For deterministic replay to be successful, a thread schedule must be captured from a given program. Once the thread schedule has been captured, deterministic replay becomes possible by forcing the schedule to be followed. P-Rep, the tool presented in this paper uses an existing record tool, KLEETHREADS for detecting concurrency bugs in concurrent C programs that use the Pthreads API. P-Rep takes a schedule from KLEETHREADS and then follows the path of execution as determined by the passed schedule.

The Pthreads API is a particularly large API and as a result of its size, we have focused on implementing the most widely used parts of the API. We have focused on creating a robust tool to work with thread creation, thread joining, mutex locking and unlocking, and waiting on conditions.

1.2 Motivation

Software bugs remain a major source of difficulty for software developers and with the advent of multi-core processors and the need to write concurrent software. New, more devastating bugs have arisen. These bugs are often difficult to track down and occur only under very specific conditions which might not arise even after extensive testing.

For example, consider the short program in 1.1. Here we have a shared variable being accessed by multiple threads. Each thread increments the value of the variable. Under such conditions, we cannot guarantee what the final outcome of the shared variable will be and we will find after numerous run-throughs that the value of the variable will be different, resulting in an indeterminate program state and a hard to find bug.

```

int GLOBAL;

Thread A
Thread B

run_global() {
  GLOBAL++;
}

create_thread(A, run_global());
create_thread(B, run_global());
```

Listing 1.1: Shared Variable Accesses

To ease the detecting of such bugs, many tools which provide execution traces using a number of techniques have been created.

One such tool, KLEETHREADS [6], uses dynamic symbolic execution [5] to trace concurrency bugs in concurrent C programs. While KLEETHREADS has proved excellent in detecting bugs, it does not support a replay feature to show programmers the bugs which it detects. The motivation behind this project has been to provide such a tool.

Programmers can often be sceptical about the results of static analysis tools [1]. This is because static analysis tools can sometimes report false positives and detect bugs which do not actually appear in the program. Static analysis tools might also produce output that is complex and at times indecipherable to a programmer who is not familiar with the tool. Some tools might have a steep learning curve and a software developer might simply not have the time to learn the nuances of the tool. They may wish simply to be shown the bug and nothing more. Many statis analysis tools will show the conditions which will cause a program to crash. In a multi-threaded application, however, the conditions are often changing and thus there is a high demand to provide deterministic replay to show a particular route to a given bug.

Tools like gdb and other popular debuggers can often slow down execution and prevent bugs from occurring in the way that they did in the original execution. By adding a deterministic replay feature to the KLEETHREADS tool, we are able to show programmers the conditions under which their bug arises and demonstrate the exact conditions under which the bug occurred.

By showing a programmer a bug quickly and allowing them to recreate the exact conditions under which the bug occurred, we will be able to speed up the debugging process significantly as through a combination of the schedules output by KLEETHREADS and the deterministic replay provided by this tool, it is simple to detect, trace and fix bugs quickly and efficiently.

It is also desirable to create a replay feature for a program without having to alter the original source code. This way the programmer can simply run P-Rep using their program and feed P-Rep a schedule. P-Rep will then follow this schedule faithfully to reproduce the conditions under which the bug occurred. By requiring no modification of the original executable, a user is able to use P-Rep with ease.

Some deterministic replay tools [13,14], however, achieve deterministic execution in non-deterministic programs but incur a large overhead. Ideally, a tool should incur as little overhead as possible. A tool that produces a large time and space overhead can be inconvenient to use and sometimes bug traces may be so large that such tools will run out of memory before finishing the execution replay.

1.3 Objectives

The objectives for the project are as follows:

- To provide a tool that takes schedules produced by KLEETHREADS and accurately forces the thread schedules to be followed.
- To create four different modes of execution for P-Rep.
    1. To follow a schedule and either hang or terminate execution depending on its current state, since this will show the point at which the bug will occur.
    2. To follow a schedule either to its end or an arbitrary point and then resume normal execution so that results can be observed from a certain point. This is useful should a user wish to observe certain scenarios.
    3. To follow a schedule to its end and terminate. This prevents the program from hanging and is useful to examine the exact state of the program at the end of the schedule.
    4. To provide a record feature to show and save the schedule P-Rep followed which can be used to detect deadlocks and show the schedule that led to the deadlock.
- To create P-Rep such that it can be run with the original executables, so that no source-code level modifications are required and such that no re-linking is required.
- To create P-Rep while incurring as minimal runtime overhead as possible so that large schedules can be followed with minimal memory footprints.

1.4 Contributions

The contributions for the project have been as follows:

- A novel way of using function interception on Linux platforms to interpose Pthread functions and force thread schedules to be followed.
- A deterministic replay tool that can be run with the original executable and the original functions in the Pthreads API.
- A deterministic replay tool that incurs a minimal additional overhead by way of function intercepting using the LD_PRELOAD environment variable.

1.5 Results

The results for the project have been promising. P-Rep works successfully with a large number of thread schedules output by the KLEETHREADS tool and successfully executes the original executable with minimal overhead. P-Rep only requires an executable that uses the Pthreads API as well as a thread schedule for it to follow.

P-Rep does not have to be used exclusively with KLEETHREADS as it will force execution on any schedule it is passed. Thus a user can create their own schedules for P-Rep to follow and observe the output. In this way, P-Rep is truly a deterministic replay tool for C programs and does not rely on any specific record feature to function correctly.

The main limitations of the project have been in the size of the Pthreads API and the amount of time allocated to complete the project as well as some portability issues. As a result of the size of the Pthreads API, we have been unable to integrate P-Rep with every aspect of the API. Instead, we have focused on creating a robust tool for thread creation, thread termination, mutex locking and unlocking, and waiting on and signalling condition variables.

There have also been some portability limitations with P-Rep. While P-Rep is guaranteed to work with the Linux kernel, the use of `pthread_tryjoin_np(...)` in thread termination calls limits its portabiliy across other POSIX compliant platforms.

Below is a list of the Pthread functions we have implemented:

Table 1.1: Pthreads API Table

API Function	**Usage**
pthread_create(...)	Thread Creation
pthread_join(...)	Thread termination
pthread_mutex_lock(...)	Mutex locking
pthread_mutex_unlock(...)	Mutex unlocking
pthread_cond_wait(...)	Condition waiting
pthread_cond_broadcast(...)	Condition signalling
pthread_timed_wait(...)	Condition Waiting

1.6 Structure of this Report

The rest of this report is structured as follows: Chapter 2 presents background and related work, as well as some of the inspirations for P-Rep. Chapter 3 discusses the approach that was taken in the creation of P-Rep and the key concepts of the implementation. Chapter 4 discusses the development process and the problems encountered along the way. Chapter 5 discusses the testing process and the evaluation of P-Rep. Finally, Chapter 6 summarizes the document and concludes the achievements made by the project and also discusses possible future work.

Chapter 2

Background and Related Work

Tools already exist that provide both record and replay features to concurrent software on a variety of platforms. This chapter introduces some of the existing record/replay tools and examines their effectiveness and how they have influenced the creation of this tool. We also give a brief introduction to different concurrency problems and their typical solutions.

2.1 Concurrency

Concurrency can best be described as two or more things occurring at once. The lifetime of the two operations may overlap, but individual steps may or may not occur simultanaeously. It differs from parallelism (although the two are often used interchangeably) since concurrency allows thread interleaving while parallelism generally refers to two or more threads running simultaneously.

Concurrency has become a major problem for the modern software developer as many programs are expected to have at least a basic level of concurrency support. The time for developers to simply rely on increased CPU performance to speed up their programs has come to an end [2].

Writing concurrent software produces bugs that would not usually occur in single-threaded applications.

Concurrency bugs are difficult to reason about and there may be many possible paths of execution that can be followed in a given program state. As a result of this, even after extensive debugging and testing there may still be bugs in a concurrent system that go unnoticed for long periods of time and applications may even ship with undetected concurrency bugs which can lead to serious complications.

In recent times the need to have sophisticated debugging tools for concurrent software has increased. It is particularly useful to show a programmer their error rather than simply showing the error trace.

Race Conditions

A race condition occurs when two or more threads attempt to access a shared variable, without first obtaining a mutex lock. The reading or writing of a shared variable or state should be considered a critical section and should thus be mutually exclusive from other threads. In its critical section, a thread should not be interrupted in order to guarantee the integrity of its access to the shared variable.

Data races can come in two forms. A critical data race occurs when the order in which the value of a variable or variables will effect the final outcome of the program.

A non-critical data race occurs when no matter what order the variables are changed in the final state of the program remains the same.

Both types of data race can have equally devastating consequences in application critical environments, such as healthcare [3].

The program in listing 2.1 shows a data race occurring. It is written in a pseudocode not dissimilar to the Pthreads API. Two threads compete to access the global variable at the same time and the final output of the variables cannot be known ahead of time, since if thread A arrives first the value will be 2 and if thread B arrives first, the values will be 1.

```
int GLOB = 0;

void* set(int a){
  GLOB = a;
}

int main() {
  Thread A,B;
  create_thread(A, set, 1);
  create_thread(B, set, 2);

  thread_join(A);
  thread_join(B);

  return 0;
}
```

Listing 2.1: A Basic Data Race

Deadlock

A deadlock is when two or more competing threads are waiting on each other with neither releasing their held resource, resulting in no progress being made in the system. A common cause of deadlock is a thread failing to unlock a mutex after finishing its critical section.

The Coffman conditions [4] listed below, are the required conditions for a deadlock to occur:

1. Mutual Exclusion: At least one resource must be non-shareable. Only one process can use the resource at any given instance of time.
2. Hold and Wait or Resource Holding: A process or thread is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. No Preemption: The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
4. Circular Wait: A process or thread must be waiting for a resource which is being held by another process or thread, which in turn is waiting for the first process or thread to release the resource.

The program in listing 2.2 shows a deadlock occurring. Thread A locks the mutex but does not unlock it so Thread B is left waiting for the lock but will never obtain it. Since we must wait for the threads to finish their operations before terminating, the program will enter a deadlocked state as no further progress can be made.

```
int GLOB = 0;
void* set(int a){
  lock_mutex(mutex);
  GLOB = a;
}

int main() {
  Thread A, B;
  create_thread(A, set, 1);
  create_thread(B, set, 2);

  thread_join(A);
  thread_join(B);

  return 0;
}
```

Listing 2.2: A Basic Deadlock

2.2 Bug Detection

Some tools have been created to detect concurrency bugs using a variety of techniques. We give a brief outline and evaluation of each of these tools below.

2.2.1 KLEETHREADS

KLEETHREADS uses [6] dynamic symbolic execution to detect data races in concurrent C programs. It is built on top of the KLEE symbolic virtual machine. KLEE is capable of automatically generating tests that achieve high coverage of complex programs [5]. KLEE uses the LLVM instruction set to map instructions to constraints without approximation and to execute the program within its framework. KLEE uses symbolic execution on a set of generated states to select and execute a wide variety of execution paths. KLEE handles nondeterminism by modelling the native execution environment symbolically [6]. This enables KLEE to consider multiple paths under a variety of situations, allowing it to model the non-deterministic state.

After KLEETHREADS has found a bug it will produce a thread schedule that led to the discovered bug and continue its execution to find more bugs. A typical thread schedule that KLEETHREADS produces will instrument the main thread as 0 and instruments each created thread as 1,2,3 etc. An example schedule might look as follows: (0,0,0,1,1,2,2,3,2,2,1), with the end of the schedule leading to a bug. KLEETHREADS will typically output a large number of schedules for any given executable. If a program is particularly large, it is possible to pass KLEE arguments to restrict the amount of schedules produced and also to alter what sort of schedules it produces. For example, we can force KLEETHREADS to output longer schedules that might lead to a very specific trace occurring. This can be useful to the user since it allows them to track down specific bugs.

KLEE supports symbolic implementation of many POSIX functions so many programs using POSIX-compliant software can run in KLEE without having to be recompiled or instrumented. This is a nice match to our tool which similarly requires no recompilation or re-linking of the executables it is passed.

Our tool was originally conceived as an extension to KLEETHREADS that would provide deterministic replay using the schedules that KLEETHREADS output. For this reason, the schedules which our tool follows are similar in design to those output by KLEETHREADS that is, the main thread is 0 and each thread created is instrumented as 1,2,3 etc. Nevertheless, our tool can operate independently of KLEETHREADS as long as it is invoked with an appropriately formatted schedule.

Being built on top of the KLEE symbolic virtual machine enables KLEETHREADS to have a very wide search space and produce many more schedules than typical replay tools. This is a major advantage, given the number of concurrency bugs it can find in a given program after just one symbolic execution.

2.2.2 Helgrind

Helgrind is a tool provided as part of the Valgrind tool suite for detecting synchronisation errors in C, C++ and Fortran programs that use the Pthreads API. It focuses on three key areas of error detection: misuses of the Pthreads API, potential deadlocks, and data races.

Helgrind works in conjunction with Valgrind so executes a user's program within the Valgrind framework. It intercepts Pthreads function calls to check for misuses of the Pthreads API and reports the errors back to the user. It detects potential deadlocks by monitoring the order in which threads obtain locks and by building a graph to represent this ordering. If it detects a cycle in the graph it has built it means that no further progress is being made and Helgrind will raise a deadlock warning.

In order to detect data races, Helgrind uses the Eraser algorithm [7]. It relies on the happens-before relation [8] to determine the expected ordering of data accesses. If, through its monitoring of variable accesses, Helgrind detects that the accesses are ordered by the happens-before relation then nothing is raised. If the accesses are not ordered by the happens-before relation, however, then Helgrind will signal a data race detection. The happens-before relation only provides partial ordering not total ordering – that is that two variables are merely unordered with respect to each other.

Helgrind's main limitations are that at times it can output false positives – data races that may not actually be data races. Helgrind does not have advanced knowledge of the execution of the program it is used on. It knows a data race can occur at certain synchronisations points, but a lack of advanced knowledge means that it may detect races that are not data races at all because it cannot see future changes to the variables.

2.2.3 ThreadSanitizer

ThreadSanitizer is part of the Google data-race-test tool suite and is a data race detection tool that seeks to improve on Helgrind and create a more efficient concurrent bug detection tool. The Linux and Mac versions of P-Rep are based on Valgrind [9] and the Windows version is base on PIN [10].

ThreadSanitizer runs as part of the Valgrind tool suite and uses a hybrid algorithm as well as a pure happens-before mode for efficient data race detection [11].

ThreadSanitizer's hybrid algorithm works by observing a program's execution as a sequence of events. The key events are memory accesses and synchronisation events. A state machine is also used to maintain the history of its observed events [11]. ThreadSanitizer's hybrid algorithm will report more false positives but will also detect more data races.

Since ThreadSanitizer can operate in different modes, different results will be observed from each execution instance. In pure happens-before mode less data races will be detected and the results are less predictable but it will only report false positives if the program is custom-lock free. The advantage of ThreadSaniztier's pure happens-before mode is that ThreadSanitizer will report all locks involved in a race while a classical pure happens-before mode is unaware of locks and thus can't include them in its report [11].

ThreadSanitizer's main advantage is that it uses Valgrind, to provide better bug detection features than Helgrind, which also uses Valgrind, as well as being able to be run in multiple modes. When using ThreadSanitizer on our tool to detect a potential data race in the wrapper libary it proved more effective in finding the bug than Helgrind. See section 4.3.3.

2.3 Record/Replay Tools

Record/Replay in concurrent software debuggers is a feature that is provided to record a trace to a bug and then replay the conditions under which the bug occurred with the aim of reproducing the bug.

There are existing tools which provide both a record and replay feature to finding concurrency bugs. Below we discuss some of these tools and evaluate their advantages and disadvantages.

2.3.1 CHESS

CHESS introduces the concept of concurrency scenario testing, in which a system designer considers interesting scenarios to test the system under [12]. In order to replay a given scenario, CHESS uses model checking to systematically generate all interleavings within that scenario. A model checker captures all non-determinism within a system and then attempts to enumerate all of those possible choices. To capture non-determinism, CHESS uses a simple abstraction for each asynchronous execution called a task. Any execution of a task results in a new state condition. To implement this abstraction layer, CHESS provides a set of wrappers for the Win32 API.

CHESS controls the scheduling of tasks by instrumenting all of the functions in the concurrency API of the platform that create tasks and access synchronisation objects. It does this to avoid perturbing the system more than absolutely necessary and in order to prevent performance limitations by having CHESS require its own modified version of the runtime platform. Only one thread is allowed to run at anytime to emulate the test on a single processor machine. The reason for this approach is to prevent threads that are running simultaneously from creating data races that CHESS cannot control.

CHESS is impressive in its thoroughness. For each bug found by CHESS, it is able to consistently reproduce the error, therefore making it signiificantly easier to show the error and thus to debug it. By allowing a system designer to test many scenarios, CHESS allows a certain amount of customisability in its debugging and thus increases its ease of use. The fact the CHESS requires no modified runtime platform to execute is also particularly useful as it minimises the overheads produced by using a record/replay tool.

A problem with CHESS is that it does not try to reproduce all sources of non-determinism resulting in an incomplete search path. CHESS will only search the passed thread schedule so if a bug depends on a particular series of inputs under certain conditions, it will not be found under a normal execution trace in CHESS.

2.3.2 LEAP

LEAP's general approach for tracing execution is to force each shared variable to record each thread access during a given execution [13]. To achieve this, it is required to precisely locate which variables can be considered shared. This is achieved by using a static field-based shared variable scheme to identify which variables are being accessed.

LEAP compiles Java code to an intermediate representation of Java bytecode and then relies on a transformer to provide instrumentation to that intermediate representation. It does this by instrumenting critical areas of the bytecode and inserting calls to the LEAP API so that correct record and replay can be recorded by LEAP [13].

LEAP handles its record feature by invoking the LEAP API on each critical event it has identified to record the ID of the thread performing the access into the access vector. The access vector for each variable is the order in which each thread accessed the shared variable. Once program execution has finished, LEAP will output a replay driver consisting of the thread creation order list and the access vector list [13].

LEAP's replay feature works by associating each thread in the schedule with a semaphore maintained in a global data structure to allow each thread to be suspended and resumed on demand [13] rather than having to rely on Java's built in multi-threading tools which would not be suitable for LEAP's replay implementation.

LEAP is successful in that it provides a high-level of concurrent bug reproducibility while incurring a fairly minimal runtime cost. Its main disadvantage, however, is the need to insert instrumented API calls into an intermediary bytecode, since this means it cannot be used with pre-compiled Java programs. Also neither the record part nor the replay part of P-Rep can be used independently, which limits P-Rep's usability.

Our tool takes a similar approach to LEAP in thread scheduling, since it associates all threads with a global pthread condtion variable so that threads can be put to sleep and woken on demand. This approach is advantageous in its simplicity since it requires no intervention with existing source code.

2.3.3 DejaVu

DejaVu aims to provide a record/replay feature to the Java programming language by identifying a logical thread schedule [14]. Identifying a logical thread shedule aims to make DejaVu more efficient.

A logical thread schedule consists of one or more thread schedules belonging to the same equivalence class. Schedules can be deemed equivalent if the physical thread schedule matches the ordering of shared variable accesses in a given execution [14]. The information captured in a logical thread schedule is enough for DejaVu to reproduce the execution behaviour of a program during execution.

DejaVu is a modified Java Virtual Machine that can provide deterministic replay to multi-threaded Java programs [14]. This is a desirable feature since it means that the replay can be provided on existing Java bytecode with minimal overhead.

The main advantage of DejaVu is that it can be used with programs written in Java with no need for recompilation. The main disadvantage is that users wishing to DejaVu must use the virtual machine provided to debug their programs.

2.4 Other Approaches

2.4.1 Output Deterministic Replay

Output Deterministic Replay (ODR) is a software only tool that targets a variety of software and provides a minimal overhead in its replay execution. It is written in C and x86-Assembler and aims to be a lightweight middleware for Linux programs [15].

The main difference between ODR and other replay tools is its approach to replication of the original file. ODR provides only a low-fidelity replay system, rather than the structured deterministic approach other tools provide [15].

The *output-failure replay problem* is to ensure that any failures which occur in the original run of a program are visible in the replay of the program [15]. This is a different approach to deterministic replay since it provides guarantees about the final program state rather than the execution path, allowing multiple paths to be explored in a given replay provided they lead to the same output. This can expose bugs that might occur under numerous circumstances.

To solve the output-failure replay problem ODR focuses on output determinism – that is the output of the replay will match the output of the original executable. So reads and writes of shared variables can happen in a different interleaved order than in the original execution but their final values, or output, must be the same as at the end of the original program execution.

By relaxing its record model and focusing on output-determinism ODR provides a low-overhead recording system. Its main downside, however, is that it does not provide full deterministic replay for multiprocessor software systems, since it instead focuses only on output-determinism. In all, ODR offers a unique and interesting new approach to providing a record/replay feature that incurs a minimal overhead.

ODR's approach, however, is not suitable with our tool. As the aim of the creation of this tool was to provide accurate replay of a given execution trace, simply guaranteeing that the final output will be the same as in the original execution is not enough to provide formal deterministic replay for concurrent C programs, as KLEETHREADS will output a schedule as far as a bug, thus resulting in schedules that are smaller than a full program execution which means there is no guarantee of the final output of the program's execution.

2.5 Function Interception Techniques

Function interception can be used to intercept calls to functions at runtime and either interpose a different function in place of the one called or modify the behaviour of the original function [16]. This project uses function interception to provide deterministic replay at runtime.

Below, we go over some different techniques for intercepting function calls on a variety of platforms and evaluate their suitability for this project.

2.5.1 Function Interposition in Linux

In Linux, function interception can be achieved at runtime by using shared libraries and the LD_PRELOAD variable. A user can specify any number .so files as the file to be used first for looking up shared objects. Thus, at runtime, each call to libraries will first be looked up in the library specified in the LD_PRELOAD. If the call is found, then the version in the .so provided will be used. If there is no matching function call then the system will look for the function elsewhere as standard.

This technique is particularly useful for this project as it can be applied in Linux with no re-compiling or re-linking of binaries so we can use function interception to load a different library than the one called in their executable and observe different results.

Functions provided by the `dlfcn.h` (which is a header provided by Linux to provide an interface to functions that can be used for dynamic loading) also make it simple to interpose function calls at runtime with functions provided in the .so file that is set in the LD_PRELOAD variable. This project has made extensive use of the `dlsym()` function in particular. The `dlsym(void *restrict handle, const char *restrict name)` function returns a handle to the next object with the same name as specified in the name parameter. This handle can then be used to call the function named. This is especially useful as it is possible to maintain the original functionality of the overridden function through this pointer.

2.5.2 Detouring in Windows

Function detouring intercepts function calls and re-writes target functions at run-time. Detours [17] is a library for instrumenting Win32 functions dynamically at runtime. A function is intercepted and an unconditional call is inserted in its instructions to call the new function. A pointer to the original function is preserved via trampolining.

Detouring is useful because it can be used on pre-compiled existing programs and is incredibly fast in its interception.

We have not used the detouring technique, however, for this project as the Linux kernel provides the LD_PRELOAD environment variable to intercept shared library calls at runtime.

Detouring is more focused towards extending existing binary software and adding additional functionality to such software. For the purposes of this project, however, simple function interception has proved adequate.

Chapter 3

Approach

In our approach to creating P-Rep, we identified three key areas which needed to be considered before implementation of P-Rep. In addition to these, some consideration into the different modes of execution needed to be taken into account.

In its current state P-Rep will run in four modes of execution as specified by a command line flag. In the first mode, P-Rep will simply follow the schedule to its end and either hang or continue execution dependent on the schedule length. In the second mode of execution, P-Rep will follow the schedule to its end and then terminate. In the third mode of execution, P-Rep will force the schedule to be followed to its end but then will continue normal execution. In its fourth mode of execution, P-Rep will run in record mode and will try to detect a deadlock and record the thread schedule it followed in to get to that deadlock. All of these modes are discussed further below.

3.1 Function Overriding

In order to force a program to behave in a certain way through its function calls, it is necessary to provide a mechanism to override a function call.

Imagine we have a function:

```
some_fun_add(int i);
```

that is called by a program. Let's assume that this function adds the integer i to some value. To override the behaviour of this function we can use two methods.

The first would be to rename the function as follows:

```
intercept_some_fun_add(int i);
```

Here we have the program call this newly defined function and then we can implement the behaviour of the function as we wish. This method explicitly redefines the function and would require source code instrumentation to replace every call to the original function with a call to the intercepted function.

To do this dynamically, and without having to redefine the function elsewhere, we would require the program to call the original function but we would override the behaviour of this function. For example, the program might have a series of system calls it wishes to use. Each time the program makes one of those system calls, we can intercept the call and force a different version of the function to be loaded.

Consider the original behaviour to be as follows:

```
some_fun_add(int i) {
i++;
}
```

By using function interception, we can redefine the function to be:

```
some_fun_add(int i) {
i*i;
}
```

and have the program that made the call believe it had called the original function.

Being able to do this is essential to the correct operation of P-Rep, since it eliminates the need for any source-level instrumentation.

3.2 Thread Scheduling

Thread scheduling is providing an order (or schedule) for threads to execute in. For example, we might have three threads: A,B,C and wish them to run in the order C,A,B. Without some kind of scheduling, the order in which the threads will execute is nondeterministic. In order to add complete determinism to a given program we must be able to wake and sleep threads on demand.

One way to do this would be to provide a global data structure which each thread must obtain mutual exclusion with before it can execute. Each thread that is waiting on the structure can be place in a pool and be forced to wait until it is woken and able to be executed. Doing this, would enable us sleep and wake threads on demand.

Consider the following pseudocode representation of a thread waiting:

```
while not in schedule do
    sleep
end while
if eligible to be run then
    run
end if
    notify other threads
```

As we can see, a thread must wait until it is eligible to be run and then run and then notify the other sleeping threads so that they can check if they are eligible to be run.

Another possible way to provide accurate thread scheduling might me to create some kind of a schedule to explicitly check the schedule and choose which thread to run next. This type of implementation would add a level of fine-grained control to the schedulign system. The problem with this approach, however, is complexity. For the sake of simplicity, we decided it was far easier to have each thread scheduled as above rather than creating a scheduler to run threads.

3.3 Synchronisation Points

Synchronisation points are points in a given schedule when thread synchronisation should occur. Using synchronisation points allows a particular thread schedule to be enforced, since it forces a thread's status to be checked at certain points in its execution.

Such points might be locking a mutex, unlocking a mutex, a thread being created or a thread joining.

At each synchronisation point it is necessary to check if a thread is eligible to be run. A thread should enter the synchronisation point and wait on the schedule. If the thread appears in the schedule then it may continue its execution. When the thread is about to leave the synchronisation point it should notify the other sleeping threads so that they can check their eligibility to be run. If a thread is woken and is not eligible to be run, it should go back to sleep and wait sleep until another thread wakes it after leaving a synchronisation point.

3.4 P-Rep's Thread Scheduling

In order to guarantee a particular thread schedule is followed, it is necessary to provide a thread scheduler to determine which threads should be run next. The role of the thread scheduler should be to put threads to sleep and wake them up on demand and explicitly specifiy which thread is eligible to be run.

This tool approaches thread scheduling in a simple manner. When threads are created they are assigned an instrumented ID which is matched with their pthread ID and placed into a C++ STL map container. Each time a synchronisation point occurs, a thread must call the `wait()` function in order to check if it is eligible to be run. If a thread's ID does not match the current ID shown in the schedule then it will be required to wait on a global pthread condition variable.

So that threads are not indefinitely put to sleep an additional function, `step_and_notify`, is used to move the schedule along to its next step and to broadcast to all of the sleeping threads so that they can check if they are eligible to be run.

The main limitation to this approach is that specific threads cannot be run on demand, since the broadcast will wake all waiting threads rather than one single thread. For example, if threads with instrumented IDs: 1,2,3 are all waiting on the global condition and we wished to wake thread 2 only, this would not be possible in the current implementation of P-Rep. This loss of fine-grained control, however, is made up for by the accurate following of a specific schedule that P-Rep allows. P-Rep will follow the schedule it has been passed faithfully. To observe different behaviour, or to choose a different thread to be run at a certain time, it is enough to simply pass a new schedule to P-Rep. Having to maintain prior knowledge of the schedule and which thread to wake next would incur a significant overhead. Using a broadcast wakes all the threads, which will then read the schedule and only run if they are eligible to be run.

Since P-Rep will follow any schedule it is passed, a user can specify a schedule for it to follow to observe the results. This is particularly useful for scenario checking, since a system designer can write a specific schedule to lead to a certain scenario and then observe the behaviour. In this way, P-Rep can be used independently of KLEETHREADS.

Some thread schedules are simply infeasible. For example, if the passed schedule is handmade, there might be syncrhonisation point that the schedule write does not consider, therefore resulting in schedule which simply cannot be follow with P-Rep. In this case, P-Rep will hang or raise an exception and terminate stating that the schedule it has been passed in invalid. If no schedule is specified, P-Rep will run in record mode.

3.5 Wait

The `wait()` function is crucial to ensuring that threads wait until they are eligible to be run. It is implemented by using a global `pthread_cond_t` condition variable to force all threads to wait on the schedule. By taking this approach, threads can be effectively put to sleep and woken up on demand.

A thread will be required to call the `wait()` function at each synchronisation point, so that correct synchronisation of all active threads can be achieved. Once a thread has entered a wait state, it will wait until its instrumented ID matches the current ID in the schedule. This approach means that only the scheduled thread will be allowed to run next and all other threads will be required to wait and in this way threads can be put to sleep and woken up on demand each time they call a function that requires synchronisation.

Listing 3.1 shows our implementation of the wait function as you can see we also make use of the original Pthreads API, specifically the Pthread condition functions which means no source code instrumentation is required.

```

/*wait*/
void wait()
{
  int thread_id = get_instrumented_id(pthread_self());

  mutex_lock(&step_mutex);
    while(thread_id != schedule[step])
      {
        //cond_wait calls the original pthread_cond_wait function
           through the dlsym() handle
        cond_wait(&step_condition, &step_mutex);
      }

    mutex_unlock(&step_mutex);
  }
}
```

Listing 3.1: The `wait()` function

3.6 Step and Notify

The `step_and_notify` function is used to increment the current schedule step and to wake threads which are currently sleeping in the `wait()` function. `STEP` is a global variable used to monitor the progress of the excutable and to index the schedule. Each time a thread enters and leaves a synchronisation function a call to `step_and_notify()` is made. The call wakes other threads so that if the callee is not eligible to be run, the scheduled thread can begin execution. Each call to the `step_and_notify()` increments the global step variable and moves the schedule along.

Like the `wait()` function, `step_and_notify()` makes use of an existing Pthread API function – specifically `pthread_cond_broadcast()`. The function must call this to ensure that threads waiting in the `wait()` function are woken so that the schedule can be checked and the eligible thread can be run. P-Rep has no current implementation to intercept the `pthread_cond_broadcast()` function so there is no need to call it through a handle returned by `dlsym()` and the original API function can be used without issue.

To ensure the safe protection of the global step variable, each thread that enters `step_and_notify()` must lock a mutex before performing any write access of the step variable. Without this measure, it would be impossible to follow a schedule in its entirety without getting skewed values for step.

Listing 3.2 show our `step_and_notify` function. As you can we use `pthread_cond_broadcast` to wake the other threads so that they can check the schedule and resume execution if they are eligible.

```
/*step and notify()*/
void step_and_notify()
{

//mutex_lock and mutex_unlock call the original pthread_mutex_lock
    and pthread_mutex_unlock functions through
//the dlsym() handle
  mutex_lock(&step_mutex);
  step++;
  pthread_cond_broadcast(&step_condition);
  mutex_unlock(&step_mutex);
}
```

Listing 3.2: The `step_and_notify()` function

3.7 Modes of Execution

When approaching the development of P-Rep, it was appropriate to offer four modes of execution so that users could specify the results they might want to observe

3.7.1 Default Execution Mode

In its standard mode of execution, P-Rep will follow a schedule to its end and then either hang or reach the end of the program as normal if that is where the schedule leads. Used in conjunction with KLEETHREADS, in this mode P-Rep will simply cease execution and hang at the end of the schedule. The purpose of this execution mode is to show the user their bug occurring under the very specific conditions that it might occur in the first place.

As an example execution run, consider the following:

- P-Rep is fed the following schedule which leads to a data race: 0,0,0,1,1,2.
- P-Rep will run thread 0 three times and then switch to thread 1. That is thread 0 will perform three operations that involve synchronisation points and then a context switch will occur.
- Thread 1 will run through two synchronisation point before thread 2 is executed.
- At this point P-Rep will hang.

In this example, the exact conditions under which the data race occurred have been reproduced by P-Rep. Thus, a programmer can observe the output of the thread schedule being followed and recreate the conditions under which the data race occurred. This is useful since the programmer can use this schedule in a tool like `gdb` and then use `gdb's` features to examine the state of their program after it has reached this point.

It is important to note that in a data race scenario as described above, P-Rep will hang as a result of the schedule end being reached not as result of a data race or deadlock occurring. P-Rep has reached the end of the schedule but threads still look at the schedule at each synchronization point. Since the thread numbers are no longer valid now that we are at the end of the schedule, no further progress is made as no threads are eligible to be run.

3.7.2 FOLLOW_AND_TERMINATE

In this mode of execution P-Rep will follow a schedule until its end and then terminate. The aim is to show a user the first bug and reproduce the conditions under which the bug arose.

To achieve this follow and terminate technique. P-Rep sets an environment variable which is read by the .so library at the start of the program. If the FOLLOW_AND_TERMINATE environment variable has been set then this flag will be raised and each time the `wait()` function is invoked, P-Rep will check whether or not the end of the schedule has been reached. If it has then the program will terminate and print the schedule that it followed in this execution run.

The downside to this approach occurs in the checking of a global flag to determine if the schedule has come to an end, since multiple threads will call `wait()` at the same time, thus creating a readers/writers problem. This is trivial to solve using mutexes, however, and maintains the minimal overhead the program incurs.

3.7.3 FOLLOW_TO_END

When running in this execution mode, P-Rep will follow the schedule it is passed until its end and then resume normal execution. This allows a particular path through a program's execution to be followed and then the remaining execution of that program to be observed. `pthread_create()` The user's program will not, however, be executing the original Pthreads functions that have been intercepted by P-Rep. This is because once the LD_PRELOAD variable has been set the loader will only look in the location set in the environment variable for shared library objects. Thus all calls to Pthread functions supported by P-Rep will still be intercepted until the program has finished its execution. To overcome this limitation, the script that launches P-Rep will set an environment variable FOLLOW_TO_END which will be checked by P-Rep on the first call to `pthread_create()`. P-Rep will set the relevant flag and once the end of the schedule has been reached, all further calls to intercepted Pthread functions will skip the `wait()` and `step_and_notify()` functions and instead simply call the orignal API functions through the handles returned by calls to dlsym().

The function which determines if we have reached the end of the schedule and can resume normal execution is shown in listing 3.3:

```
int no_follow()
{
    mutex_lock(&nofollow);
    if(FOLLOW_TO_END && step >= schedule_length()) {
        if(!spoken) {
            fprintf(stderr, "Program has run to the end of the
               schedule and is resuming normal execution ...\n");
            spoken++;
        }
        mutex_unlock(&nofollow);
        return 1;
    }
    mutex_unlock(&nofollow);
    return 0;
}
```

Listing 3.3: `no_follow()` function

This approach means that the original executable can continue to run as normal, while incurring a minimal overhead in its call to the intercepted functions.

One problem we encountered when implementing this mode was determining if threads should call the normal versions of the Pthreads API functions or the overridden one. There were several cases in which we found that after the `no_follow()` function was returning true, threads which had been running in parallel were already being executed with the overridden function because they had already progressed past the points which determined which function to call.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
{
  int res;
  if(!no_follow() && !RECORD_MODE){
    int id = get_instrumented_id(pthread_self());
    step_and_notify();
    wait();
    if(pthread_mutex_trylock(mutex))
      {
        return 0;
      }


     if(no_follow()) {
       return mutex_lock(mutex);
     }


     step_and_notify();
     wait();


     if(pthread_mutex_trylock(mutex))
       {
        return 0;
      }
    assert(false);
  } else {
    step_and_notify();
    res = mutex_lock(mutex);
    return res;
  }
}
```

Listing 3.4: Mutex locking when FOLLOW_TO_END set

As an example of this problem consider the small code snippet shown in listing 3.4.

Consider two threads: A and B.

In this example, we can see that it might be possible for thread A to enter the `pthread_mutex_lock` function before the end of the schedule has been reached so it will enter the overridden branch of the code and call `pthread_trylock`. While it is in this branch, thread B could cause the end of the schedule to be reached and the `no_follow` function would start returning 1. Thus thread A would be calling `pthread_trylock` rather than the original `pthread_mutex_lock` and so not executing as normal. In the case that there was a locked mutex that thread B was trying to obtain, B would exit this function normally as if it had obtained the mutex. This would of course, avoid any deadlocks or data races that might occur at the end of a particular schedule.

To fix this problem, we inserted calls to `no_follow()` before returning from our overridden function calls, so that in an event similar to above, the original pthread functions would still be called. While this added a small extra overhead to the calls to the overridden functions, it elminated the possibility that threads would still be calling our overridden function when they were meant to be calling the normal functions.

3.7.4 RECORD Mode

P-Rep's record mode's purpose is to detect a given deadlock and then output the schedule that was followed leading to this deadlock.

P-Rep's minimalist replay mode was implemented using a simple monitor thread. On the first call to `pthread_create()` in any program, P-Rep will create and initialise a thread to monitor the progress of thread execution. The monitor thread's aim is to check for satisfactory progress and, if satisfactory progress has not been made, to terminate program execution and print the schedule that P-Rep followed to reach the deadlocked state.

In order to build a schedule independent of P-Rep's execution mode, we inserted a call to a C++ function that dynamically builds a schedule with each call to `step_and_notify()`. Since `step_and_notify()` is called at every synchronisation point, we were able to trace the order in which threads were executing in any given execution. After the monitor thread detects a deadlock, it will output the schedule P-Rep followed to a file so that this file can then be fed to P-Rep and the execution replayed.

Our initial approach to the monitor thread is shown in listing 3.5. As we can in line 5, the thread will sleep for two seconds before checking if satisfactory progress has been made and if it hasn't then it will terminate execution and output the schedule that it followed to reach the deadlock.

```
 1 /*monitor thread to check satisfactory progress*/
 2 void* monitor_progress() {
 3   int last_step = step;
 4   while(true) {
 5     sleep(2);
 6     if(last_step == step) {
 7       /*if there has been no change after execution, assume
            satisfactory progress has
 8         not been made and report a deadlock*/
 9       fprintf(stderr, "Potential deadlock detected!\nTool will now
            exit with error code: %d\n\n",
10                5);
11       print_followed();
12       exit(5);
13     } else {
14       last_step = step;
15     }
16   }
17 }
```

Listing 3.5: Modified Monitor Thread

This monitor will work independently of a schedule and will work with P-Rep if no schedule is passed, since the global step is incremented each time `step_and_notify` is called and P-Rep builds the followed schedule each time `step_and_notify` is called.

The record mode can in fact be used to trace any execution that might lead to a deadlock in a concurrent C program. This is in keeping with P-Rep's independence, since it means that a user can obtain a trace of any execution that causes a deadlock should they wish to.

RECORD mode is not completely problem free and we discovered some issues with P-Rep's RECORD mode in our evaluation of it in secion 5.2.3.

3.7.5 Modified `wait()`

The modified version of the `wait()` function to account for the different execution modes is shown in listing 3.6

```
/*wait*/
void wait()
{
  int thread_id = get_instrumented_id(pthread_self());

  if(FOLLOW_TO_END)
    {
      mutex_lock(&follow);
      check_status();
      mutex_unlock(&follow);
    }



  if(!no_follow()) {
    mutex_lock(&step_mutex);
    while(thread_id != schedule[step])
      {
        check_status();
        if(no_follow()) {
          mutex_unlock(&step_mutex);
          return;
        } else if(terminate_set()) { end(); }
        if(VERBOSE) {
          fprintf(stderr, "Thread %d sleeping ... schedule[%d], step
             :%d\n",
                  thread_id, schedule[step], step);
        }
        cond_wait(&step_condition, &step_mutex);
      }

    mutex_unlock(&step_mutex);
  }
}
```

Listing 3.6: The modified `wait()` function

Chapter 4

Development

Below we detail the development process we underwent to create P-Rep. We explain, in detail, how we implemented each overriden function and discuss the problems we encountered and their resolutions.

4.1 Thread ID creation and Assignment

When a thread is created using `pthread_create()` an ID is assigned to the pthread. This ID is usually a hexadecimal number that is determined by the operating system. Since KLEETHREADS aims to work with any concurrent C program and it cannot know ahead of time what thread ID the operating system will assign to threads, it outputs its schedules with thread IDs starting at 0. An example schedule might be (0,1,2,3). Thread 0 will always represent the main thread and each thread from then on will be assigned an ID.

Since P-Rep receives a schedule in this format, it was necessary to add instrumented IDs to the threads as they were created so that a schedule could be followed without any modification.

To achieve this, each time a thread is about to be run its instrumented ID along with its pthread ID is inserted into a C++ STL map which holds a mapping of instrumented IDs to pthread IDs. Each time a thread enters a section in which its ID is required to continue, a call to `pthread_self()` is made to obtain the pthread ID. This pthread ID can then be used to index the map and obtain the instrumented ID of the calling thread. Giving a function the "extern C" qualifier will prevent the C++ compiler from mangling the names, a C program is able to call the C++ function as if it were a C function. Doing this, enabled us to use C++ libraries and tools with the libraries written in C.

An example thread map might look as follows:

Table 4.1: Thread Map Example

Pthread ID	**Instrumented ID**
0x7ffff7fbd740	0
0x7ffff6c76700	1
0x7ffff6477700	2
0x7ffff6478700	3

Using an STL map enables fast look-up of thread IDs and also means that nothing needs to be passed to the `wait()` and `step_and_notfiy()` functions to verify the thread being called.

Two functions achieve the above functionality. The first `put_instrumented_id(pthread_t p, int current_id)` will create a mapping between `p` and `current_id`. In this case, `current_id` is a global variable, incremented on each call to `pthread_create()` to hold the ID of the next thread to be created. To prevent data races, a thread must lock the global `current_id` before incrementing it.

The second function `get_instrumented_id(pthread_t p)` will look up and return the instrumented ID of the pthread `p` in the STL map defined in the thread_map.cpp file. The returned ID can then be used by P-Rep to check the schedule and determine if the calling thread is eligible to be run.

Below is the source code for the Thread Map:

```
#include <map>
#include <iostream>
#include <pthread.h>
#include <assert.h>

#include "thread_map.h"

pthread_mutex_t put_mutex;

std::map<pthread_t, int>* idMap = NULL;
pthread_mutex_t print_mutex;


extern "C"
int get_instrumented_id(pthread_t id) {
  assert (idMap->find(id) != idMap->end());
  int instrumented_id = (*idMap)[id];
  return instrumented_id;
}


extern "C"
void put_instrumented_id(pthread_t id, int instrumented_id)
{
  if(!idMap)
    {
      idMap = new std::map<pthread_t, int>;
    }
  pthread_mutex_lock(&put_mutex);
  (*idMap)[id] = instrumented_id;
  pthread_mutex_unlock(&put_mutex);
}


extern "C"
void print_mappings()
{

  if(!idMap)
    {
      return;
    }
  else {
    pthread_mutex_lock(&print_mutex);
    std::cout << "Map contents:" << std::endl;
    for(std::map<pthread_t, int>::const_iterator iter = idMap->begin
       ();
        iter != idMap->end(); ++iter)
      {
        std::cout << std::hex <<iter->first << '\t' << std::hex
                << iter->second << std::endl;
      }
    pthread_mutex_unlock(&print_mutex);
  }
}
```

Listing 4.1: The Thread Map

4.2 Wrapper Library

Before implementing P-Rep using function intercepting, we decided to write a wrapper library to use with our own Pthread programs. The purpose of creating such a wrapper was to come to terms with the Pthreads API but also to make sure that the key parts of P-Rep functioned in this environment first before moving on to the more complex function interception techniques.

The wrapper library worked by defining a set of functions in a header that could be included in a program and then called to create threads and perform thread operations. In these intercepted functions we implemented the main features of P-Rep. That is on each call to an intercepted function, the thread would be required to enter a synchronisation point and wait until it was eligible to be run.

This part of the development process was purely for experimental purposes and to understand the processes involved in designing P-Rep. The obvious downside to using the wrapper library exclusively would have been the need for the program to be recompiled with the wrapper library included and the program to explicitly call the overridden functions.

To help with reasoning about how schedules should be followed, we wrote programs in which several threads accessed and altered a global variable without attempting to lock a mutex. The aim here was to write schedules that would induce a particular value for the global variable at the end of the program's execution.

In the program in 4.2, the schedule: (0,0,0,0,0,1,3,4,2,0,0,0,0,0) will guarantee that the final value of `GLOB` will be 3. A minor change to this schedule will force the `GLOB` to be 1,2, or 3. The purpose of this exercise was to ensure that P-Rep could faithfully follow certain schedules and enforce a particular value for the global variable at the end. This was then transferable to the version of P-Rep which used function intercepting and passing it the same schedule would yield the exact same results.

By opting to write a wrapper library to see how P-Rep would work before implementing any dynamic interception, we were able to get to grips with how the threads could be put to sleep and woken on demand and also how thread scheduling should work. Once the wrapper was at a workable state and we were satisfied with the way it handled schedules, we could then move on to implementing P-Rep via function intercepting.

```
#include "../wrapper.h"
int schedule[] = {
  #include "../schedules/racesched/schedule#3.h"
};

int GLOB = 0;
void* fn(void*);


int main()
{

  pthread_t a = intercept_pthread_create((void*)fn, (void*)1);
  pthread_t b = intercept_pthread_create((void*)fn, (void*)2);
  pthread_t c = intercept_pthread_create((void*)fn, (void*)3);
  pthread_t d = intercept_pthread_create((void*)fn, (void*)4);


  intercept_pthread_join(a, NULL);
  intercept_pthread_join(b, NULL);
  intercept_pthread_join(c, NULL);
  intercept_pthread_join(d, NULL);


  fprintf(stderr, "GLOB: %d\n", GLOB);

  return 0;
}


void* fn(void* a)
{
  int i = (int) a;
  GLOB = i;
}
```

Listing 4.2: Forcing a schedule

4.3 Function Interposition

4.3.1 Dynamic Interception

Once P-Rep was working robustly with the wrapper library, we moved on to implementing it using dynamic function intercepting.

As described previously, by setting the LD_PRELOAD variable it is possible to intercept calls to shared libaries at runtime. With this approach there is no need for the program to be recompiled or relinked to make use of P-Rep, as all calls to the Pthreads API within the program will be intercepted by the specified shared library. This is a major advantage to the programmer since it requires no additional effort on their part and means that P-Rep can be used with all existing programs that use the Pthreads API.

One of the main challenges was passing P-Rep the shedule to follow, so that P-Rep has access to the schedule. We did not encounter this problem in creating the wrapper library since the schedules were hard-coded into the executable.

When the script is run to set up P-Rep various environment variables must be set which can then be accessed by P-Rep to read the schedule and determine which mode of execution it should be run in. The schedules are held in the custom defined KLEE_SCHED environment variable. P-Rep will look for the file set in this variable and attempt to read a schedule from it. If the environment variable is not set, or no schedule is found or if the schedule is of an invalid type, then P-Rep will exit. Once the schedule has been read into P-Rep, it can be followed as normal.

The obvious problem with the above approach is the setting of excessive environment variables while P-Rep is executing. Failing to unset these environment variables after execution might cause problems if other programs need to use them. This is easily resolved, however, as long as the relevant environment variables are unset after program termination.

4.3.2 Unwanted Recursive Function Calls

Unwanted Recursive function calls was one of the main problems we faced when implementing function interception. The reason for this was that the library which P-Rep uses to intercept function calls makes heavy use of the existing Pthreads API. For example, when a thread makes a call to lock a mutex, eventually that mutex must be locked, or else execution of the program will become indeterminate.

Another problem is that the `wait()` function, which forces thread to wait their turn in the schedule makes use of both mutexes and condition variables. Any thread, therefore, that makes a call to `wait()` would inadvertently call the the overridden `pthread_mutex_lock` and `pthread_mutex_unlock` functions resulting in an unwanted recursive function call and potentially the program crashing or reaching a deadlock.

To overcome this problem, we made use of the `dlsym(void* restrict handle, const char* restrict name)` function, which looks up the next object with the name specified in the name parameter and returns a reference to the function specified.

Using these features, we have essentially hidden the real Pthreads API functions so that we can then call those functions from within the functions that we have overridden. This enables us to run a thread and lock and unlock mutexes without having the problem of calling the overridden function rather than the original function. So when a user's program makes a call to `pthread_create`, our tool will intercept this and then call the real system-implemented version of `pthread_create` to run the functions specified by the user in their function call.

Our renamed function calls are shown in listing 4.3

```
int mutex_lock(pthread_mutex_t* mutex)
{
    static int(*real_lock)(pthread_mutex_t*) = NULL;
    if(!real_lock)
        real_lock = dlsym(RTLD_NEXT, "pthread_mutex_lock");

    int res = real_lock(mutex);
    return res;
}

int mutex_unlock(pthread_mutex_t* mutex)
{
    static int(*real_unlock)(pthread_mutex_t* mutex) = NULL;
    if(!real_unlock)
```

```
        real_unlock = dlsym(RTLD_NEXT, "pthread_mutex_unlock");

    int res = real_unlock(mutex);
    return res;
}

int cond_wait(pthread_cond_t* cond, pthread_mutex_t* mut)
{

    static int(*real_wait)(pthread_cond_t*, pthread_mutex_t*) = NULL
       ;
    if(!real_wait)
    /*use dlvsym() to prevent the cond variable passed back from
       being changed to an outdated version*/
        real_wait = dlvsym(RTLD_NEXT, "pthread_cond_wait", "GLIBC_2
           .3.2");

    int res = real_wait(cond, mut);
    return res;
}
```

Listing 4.3: Renamed Pthreads Functions

4.3.3 Problems with Wrapper Library and Function Interception

The main problem we encountered when implementing the wrapper library was in inserting the correct thread ID into the Thread Map.

The problem occurred in thread creation. The initial intercept_pthread_create is shown in listing 4.4

```
/*intercept pthread_create*/
pthread_t intercept_pthread_create(void* (*fn)(void*), void* o_args)
{

  if(!first_call)
    {
      put_instrumented_id(pthread_self(), 0);
      first_call = 1;
    }

  /*set up the thread parameters*/
  pthread_t a;
  thread *args = malloc(sizeof(thread));
  args->function = fn;
  args->thread_id = current_id;

  args->other_args = o_args;


  current_id++;

  pthread_create(&a, NULL, run_thread, (void*) args);

24   /*put the instrumented id in the map*/
25   put_instrumented_id(a, args->thread_id);
26
27
28
29   step_and_notify();
30   wait();
31
32   return a;
33 }
34
35 /*start new thread*/
36 void* run_thread(void* arg)
37 {
38   thread* args = (thread*) arg;
39   wait();
40
41   /*run the function*/
42   void* result = args->function((void*)args->other_args);
43
44   /*indicate that the thread's function has been run*/
45   step_and_notify();
46
47   return result;
48 }
```

Listing 4.4: `intercept_pthread_create`

The problem in this implementation occurs in line 25. While a thread is inserting the thread ID into the thread map, it may be that another thread calls `intercept_pthread_create` and thus a data race occurs. Initially this problem was hard to track down. In order to find it, we made use of the ThreadSanitizer tool, which showed us the data race.

Using an existing concurrency bug detection tool during the development of the wrapper library was particularly useful to the overall progression of the project because it gave us a clear idea of how concurrency bugs can be traced and alerted us to some of the problems implicit in writing any kind of concurrent software. We also ran Helgrind on P-Rep to try and determine the cause of the problem. While both tools found the bug, Helgrind reported more false positives than ThreadSanitizer.

This bug was easily fixed by placing the call to `put_instrumented_id` in the `run_thread` function to guarantee that the calling thread had its ID placed in the Thread Map rather than another thread accessing the map at the same time. As a result of the semantics of STL maps, the keys were kept unique so concurrent writes would not result in existing threads having their IDs overwritten.

The new `run_thread` function is shown in listing 4.5. After fixing the bug the only race conditions that either tool found were concurrent reads of the thread map, which is acceptable since there might be occasions on execution when more than one thread is required to read the thread map to obtain its instrumented ID in order to determine its eligibility to be run.

The main problem we encountered when using dynamic interception to implement P-Rep was with the LD_PRELOAD environment variable. Unless otherwise specified, setting the LD_PRELOAD variable will be operating system wide. So any calls by any programs to the Pthreads API will result in those calls being intercepted. Needless to say this can be disastrous for a user if not handled correctly, since this can cause the whole operating system to crash. To remedy this problem, we wrote the script to set the LD_PRELOAD variable for only the executable which is passed and to

```c
/*start new thread*/
void* run_thread(void* arg)
{
  thread* args = (thread*) arg;
  put_instrumented_id(pthread_self(), args->thread_id);
  wait();

  /*run the function*/
  void* result = args->function((void*)args->other_args);

  /*indicate that the thread's function has been run*/
  step_and_notify();

  return result;
}
```

Listing 4.5: `run_thread` with data race removed

unset the LD_PRELOAD variable as soon as execution finished. This prevents any unneccessary function interception and contains the intercepting to only the program the user wishes to test.

Another issue we found when using the LD_PRELOAD was an issues with a program's permissions. In order to prevent a malicious user from intercepting system calls, a program must have the relevant permissions to link and load a dynamic library. The GNU loader will check a program's permissions on loading and determine if it is setuid or setgid. The loader will generally assume that a program does not have these permissions and will thus greatly limit its permissions on loading. While this is not problematic when running programs from within the user's own profile, running programs on a remote server via `ssh` or on any other machine where the owner of the program does not have privileged access will result in a segmentation fault at loadtime.

There was also a problem when using `dlsym()` to obtain a handle to function calls. In particular the problem arose when implementing `pthread_cond_wait()`. The `dlsym()` function makes an unversioned lookup and by default this lookup will match the oldest symbol in the dynamic shared library. If any new features are added to a libary in the future then the `dlsym()` will still call the older version. Failing to use a versioned `dlsym()` call resulted in the older version of `pthread_cond_wait` being returned which returned a reference to a different `pthread_cond` variable. If the the initial `pthread_cond` if located at 0x600d80, for example, then using the unversioned `dlsym()` function will result in the `pthread_cond` variable begin returned at 0x7ffff00008c0. In this scenario, however, the main thread will still assume that the `pthread_cond` variable is located at 0x600d80 and so will wait on this variable and receive no signal. To resolve this problem, it was neccessary to use a versioned `dlvsym()` call to return the GLIBC_2.3.2 version of the `pthread_cond_wait()` function.

4.4 Pthreads API

This project intercept calls made to thread creation, thread joining, mutex locking and unlocking and waiting on condition variables. The various implementation details for each of these functions are discussed below.

4.4.1 pthread_join() and run_thread()

The source code for `pthread_create()` using function interception is shown in listing 4.6.

```
 1 int pthread_create (pthread_t* thread,
 2                     __const pthread_attr_t* attr,
 3                     void *(*start) (void *),
 4                     void *arg) {
 5
 6   if(!first_call)
 7     {
 8       set_follow();
 9       check_verbose();
10       int got_sched = get_schedule();
11       if(!got_sched)
12         {
13           fprintf(stderr, "Failed to find the schedule! .... tool
              will now exit\n");
14           exit(1);
15         }
16       first_call = 1;
17       output_schedule();
18       mutex_lock(&id_lock);
19       put_instrumented_id(pthread_self(), 0);
20       mutex_unlock(&id_lock);
21     }
22
23   /*set up the thread arguments*/
24   params* args = malloc(sizeof(params));
25   args->thread_id = current_id;
26   args->other_args = arg;
27   args->function = start;
28   current_id++;
29   int rc;
30
31   static int (*real_create)(pthread_t * , pthread_attr_t *, void *
        (*start)(void *), void *) = NULL;
32
33   if (!real_create)
34     real_create = dlsym(RTLD_NEXT, "pthread_create");
35
36   mutex_lock(&id_lock);
37   mutex_unlock(&id_lock);
38   rc = real_create(thread, NULL, run_thread, args);
39
40     step_and_notify();
41     wait();
42   return rc;
43 }
```

Listing 4.6: Intercepting `pthread_create`

On the first call to `pthread_create()` P-Rep will look for the location of the schedule in an environment variable and read the schedule file and use this as the execution schedule to follow.

The function `set_follow()` in line 8 will check what mode of execution the tool should operate in. The function `check_verbose()` will check whether or not the tool should be running in verbose mode. If the VERBOSE flag is set then the tool will output its current status at regular points through its execution.

P-Rep will then set the `first_call` variable as shown in line 16 so that all future calls will ignore this branch of the code.

Each time a thread is created a `params` struct is instantiated to hold the relevant arguments for a thread.

The params struct is shown below:

```c
typedef struct {
  void* (*function)(void*);
  int thread_id;
  void* other_args;
} params;
```

Listing 4.7: The `params` struct

`Params` holds a function pointer to the function to be run in the newly created thread, the instrumented ID of the thread and any arguments that need to be passed to the thread function.

We create a function called `run_thread()` which runs in a separately created pthread to run the function that was passed in the original call to `pthread_create()`. We do this to prevent the thread that called `pthread_create` from having to wait on the function to terminate before it can continue processing. If we simply returned a pointer to `real_create` we would lose the fine-grained control we have over thread creation and the tool would no work as there would be no way of running the thread in such a way that forces it to go through the necessary synchronisation points.

Once a thread's parameters have been set up, a call to the real `pthread_create()` function show in listing 4.8 is made through the function pointer `real_create`. This newly created thread is then told to run the `run_thread()` function and is passed the `params` struct as its argument. At this point in the execution, the params struct is fully instantiated so the original callee's function can be run through the `run_thread()` function.

```c
void* run_thread(void* arg) {

  params* args = (params*) arg;
  mutex_lock(&id_lock);
  put_instrumented_id(pthread_self(), args->thread_id);
  mutex_unlock(&id_lock);
  wait();

  void* result = args->function((void*)args->other_args);
  step_and_notify();


  return result;
}
```

Listing 4.8: The `run_thread` function

The first thing the `run_thread()` function does is insert the calling thread's ID into the Thread Map. The reason this is done in this function rather than on the call to `pthread_create()` is because a call to create a pthread must be done by an existing thread and thus calls to `pthread_self()` will return the pthread ID of the thread which called the create function, not the ID of the thread which we are intending to create.

For example, consider the following simple scenario:

- The main thread with instrumented ID 0 calls `pthread_create()`
- `pthread_self()` is called to obtain the ID of the thread to place in the thread map
- Because the call to `pthread_self()` was made by the main thread its pthread ID will be returned.
- Thus there will be two duplicate entries for the pthread ID for the main thread with different instrumented IDs.

For this reason, it is necessary to insert the ID in the `run_thread()` function.

Before a thread can be run it must wait on the schedule. Once it is eligible to be run the function passed in the `params` struct will be executed. A function might make other calls to Pthread API functions and these are handled accordingly. After a thread has finished its execution. It must increase the schedule, step and notify other waiting threads so that they can run.

By creating another thread to run the function that was passed to the initial call to `pthread_create()`, we are able to force threads to follow the schedule. Without this approach, threads would have to be run within the call to `pthread_create()` and this would cause threads to be forced to wait on each thread creation call, resulting in a lack of proper thread scheduling. The thread created for `run_thread` can be viewed as a worker thread, as it is designated to handle the thread scheduling without any other threads having to wait on its completion.

4.4.2 pthread_join()

Below is the implementation for `pthread_mutex_lock()pthread_mutex_lock()`:

```
int pthread_join(pthread_t tid, void** value_ptr)
{
    step_and_notify();
    wait();


    int res = pthread_tryjoin_np(tid,value_ptr);
    if(res != EBUSY)
      {
        return res;
      }

    step_and_notify();
    wait();


    res = pthread_tryjoin_np(tid,value_ptr);
    if(res != EBUSY)
      {
        return res;
      }
    assert(false && "Invalid Schedule!");
}
```

Listing 4.9: Implementation for `pthread_join`

When a call to `pthread_join(pthread_t p, void** retval)` is made, the calling thread will wait for the thread specified in pthread_t to terminate and any return values are stored in the retval.

When a call to `pthread_join` is made. The calling thread must increase the global step to move the schedule along and notify waiting threads. This is to ensure that other threads can run before the real call to `pthread_join` is made, blocking the calling thread. The calling thread must then wait until it is scheduled to run. After waiting the calling thread calls the `pthread_tryjoin` function on the ID of the thread it is trying to join.

The `pthread_tryjoin_np(pthread_t tid, void** retval)` function performs a non-blocking join with the thread specified in the passed `tid`. If the thread is busy then an error is returned. P-Rep checks for this error and if the thread is busy, it will `step_and_notify` and wait before calling `pthread_tryjoin` again. On the second call to `pthread_tryjoin`, the thread should terminate. If it still returns a busy status then it must be assumed that the schedule is invalid and an assertion failure should be raised.

Of course, one of the main disadvantages to this approach is that it reduces the portability of P-Rep. In its current state, `pthread_tryjoin_np` is a non-standard GNU extension and is thus non-portable.

The positives of using the above approach outweigh the negatives, however, because it prevents the calling thread from blocking. If the calling thread blocked on each call to the `pthread_join` there might arise a situation in which all calling threads were blocked and thus a deadlock would occur.

4.4.3 pthread_mutex_lock() and pthread_mutex_unlock()

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
{
  int res;
  if(!no_follow()){
    int id = get_instrumented_id(pthread_self());
    step_and_notify();
    wait();

    if(pthread_mutex_trylock(mutex))
      {
        return 1;
      }

    if(!no_follow()) {
      step_and_notify();
      wait();
    }


    if(pthread_mutex_trylock(mutex))
      {
        return 1;
      }

    assert(false);
  } else {
    res = mutex_lock(mutex);
    return res;
  }
}


int pthread_mutex_unlock(pthread_mutex_t* mutex)
{
  if(!no_follow()) {
    step_and_notify();
    wait();


    int id = get_instrumented_id(pthread_self());

  }

  int res = mutex_unlock(mutex);

  return res;
}
```

Listing 4.10: Implementation of mutex lock and unlock

Listing 4.10 shows our implementation for locking and unlocking mutexes.

When a thread attempts to lock a mutex it enters a synchronisation point. Therefore, it must first call `step_and_notify` to move the schedule along and then wait to determine whether or not other threads are waiting on the mutex lock.

So that threads are not indefinitely blocked when waiting on a mutex, P-Rep makes use of the `pthread_mutex_trylock` function. When a thread calls this function it tries to lock the mutex. If the mutex if already locked then an error occurs, otherwise the thread successfully locks the mutex. The error which is usually returned is the EBUSY error value, which indicates that the mutex is already locked.

The important feature of `pthread_mutex_trylock` is that it is a non-blocking call to lock the mutex, so the calling thread will continue execution if an EBUSY value is returned. If, instead of using `pthread_mutex_trylock`, P-Rep used the standard mutex lock function, `pthread_mutex_lock`, threads might end up being indefinitely blocked waiting on the schedule to obtain the mutex lock, thus leading to a global deadlock.

Unlocking a mutex requires a thread only to `step_and_notify` and wait. Once ready, a thread simply makes the call to the real `mutex_unlock` through the `dlsym()` handle. The overridden function can then return the result of this unlock.

4.4.4 pthread_cond_wait()

Listing 4.11 shows our implementation for `pthread_cond_wait`.

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)
{
  step_and_notify();
  if(VERBOSE) {
    fprintf(stderr, "Thread %d waiting on condition....\n",
       get_instrumented_id(pthread_self()));
  }
  mutex_unlock(mutex);
  wait();
  mutex_lock(mutex);
  step_and_notify();

  return 0;
}
```

Listing 4.11: Implementation of `pthread_cond_wait`

When a thread makes a call to `pthread_cond_wait` it always blocks, and on return it should have obtained the mutex lock it was originally trying to obtain in the call to `pthread_cond_wait`.

In the intercepted version of `pthread_cond_wait`, a thread first calls `step_and_notify` and then makes a call to unlock the mutex. This is to ensure that it does enter a wait state with the mutex already locked. If the calling thread did not unlock the mutex on entry to the function call and it already had the mutex locked then other threads would be forced to wait, potentially leading to a deadlock. A call to unlock a mutex that a thread does not already have ownership of will result in a failure with the EPERM result. This means that it is viable for a thread to attempt to unlock a mutex it does not already have locked with no adverse repercussions.

Once a thread has obtained the mutex lock it can resume its processing and then `step_and_notify` before exiting the function call.

4.4.5 pthread_cond_timedwait()

The implementation for `pthread_cond_timedwait()` is shown in listing 4.12.

```
int pthread_cond_timedwait(pthread_cond_t *__restrict cond,
                           pthread_mutex_t *__restrict mutex,
                           __const struct timespec *__restrict
                          abstime) {
    pthread_mutex_lock(mutex);
    pthread_mutex_unlock(mutex);
    return ETIMEDOUT;
}
```

Listing 4.12: Implementation of `pthread_cond_timedwait`

We have implemented `pthread_cond_timedwait` by using the already overridden `pthread_mutex_lock` and `pthread_mutex_unlock`. The reason for this is to ensure synchronisation occurs before the thread obtains the lock and also when it releases the lock. We assume that if the thread does not obtain the lock with either of these two calls, then it has timed out.

This approach limits the effectiveness of the timed wait function, since it will always return ETIMEDOUT. For the sake of our implementation, however, it proved enough to lock and unlock the mutex and return ETIMEDOUT, since the programs we tested which made use of `pthread_cond_timedwait` ignored the return value anyway. Of course, any program that should use the return would not work with our tool but for the purposes of getting P-Rep to work in our testing environment this was appropriate.

4.4.6 Problems

One of the main problems we encountered when implementing the Pthread functions was in thread joining. There were numerous occasions when a thread could interrupt another thread before it had finished executing the `run_thread` function and return EBUSY in the call to `pthread_tryjoin_np`. If any call to `pthread_tryjoin_np` returned EBUSY twice, then we would get an assertion failure and P-Rep would cease execution.

Taking this approach guarantees the integrity of the schedule P-Rep has been passed. If the thread has failed to join after two attempts, the schedule can be deemed inaccurate, since the threads should have been forced to finish execution when the join function was called. If the thread is still active, then the schedule is inaccurate and P-Rep will not be able to progress past a certain point. If the schedule is inaccurate then P-Rep will fail and raise an exception.

This problem was resolved by ensuring that the schedules P-Rep was passed were accurate.

Chapter 5

Testing and Evaluation

This section covers the different tests we used to evaluate the overall effectiveness of P-Rep and to assess its performance under a variety of different scenarios.

5.1 Testing Method

In order to test P-Rep, we used a variety of testing methods. Our primary aim in our testing was to ensure that P-Rep followed the schedules it was passed accurately. We wanted to assess how well it performed with schedules that KLEETHREADS output, as well as how it performed with schedules we had written ourselves. We also wanted to ensure that it worked with a variety of complex programs, with threads spawning other threads etc. to see if P-Rep could maintain its integrity. These tests are shown in section 5.2. Finally we wanted to test the speed of P-Rep and evaluate our decision to use a shim library to implement the deterministic replay feature of our tool. This is shown in section 5.3.

5.2 Hand-written Tests

In order to test that P-Rep worked as expected, we wrote a number of simple tests to determine if the P-Rep followed schedules faithfully. In all of the tests P-Rep followed the schedules output by KLEETHREADS. Under almost every scenario, KLEETHREADS output a large amount of different schedules for any given executable.

In order to determine if the passed schedule has been followed faithully by P-Rep, we have modified P-Rep to build an array of thread IDs in the order in which they were executed. We then compare the built schedule with the original schedule and if they match then we can be confident that P-Rep has followed the schedule faithfully. We altered the `step_and_notify` function to log the instrumented ID of the calling thread each time it is called. Since `step_and_notify` is called at every synchronisation point, it seemed suitable to use this approach to build the schedule dynamically.

All of the hand-written tests were run in P-Rep's FOLLOW_AND_TERMINATE mode so that the end of the schedule could be reached and then the generated schedule compared with the passed schedule. Testing the accuracy of P-Rep in the other modes caused problems. Testing in its default mode will result in P-Rep hanging and no further progress being made, thus it would be impossible to determine if P-Rep followed the schedule accurately, since there is no way to extract the followed schedule.

To determine whether or not P-Rep has faithfully followed the schedule or not, at the end of a given execution P-Rep will compare the two schedules (the one it was passed and the one it generated) and throw an error code if the schedules do not match or else it will terminate normally.

To test the basic effectiveness of P-Rep, we tested it with a simple program that makes use of all of the Pthread API functions that we have overridden.

One of the our hand-written programs which we tested under KLEETHREADS is shown below:

```
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>


int GLOB = 0;
int g = 0;
pthread_cond_t cond;
pthread_mutex_t mut, g_mut;

void* f(void* c) {

  int i = (int) c;

  pthread_mutex_lock(&mut);
  while(g != 3)
    pthread_cond_wait(&cond, &mut);
  GLOB = i;
  pthread_mutex_unlock(&mut);
  return 0;
}

void* p()
{
  pthread_mutex_lock(&g_mut);
  g++;
  pthread_mutex_unlock(&g_mut);
  pthread_cond_broadcast(&cond);
}


int main()
{
  pthread_t a, b, c, d;
  pthread_create(&a, NULL, f, (void*)1);
  pthread_create(&b, NULL, f, (void*)2);
  pthread_create(&c, NULL, p, (void*)3);
  pthread_create(&d, NULL, p, (void*)3);


  g++;

  pthread_join(a,NULL);
  pthread_join(b,NULL);
  pthread_join(c,NULL);
  pthread_join(d,NULL);


  printf("\n\nGLOB = %d\n\n", GLOB);

  assert(false);

  return 0;
}
```

Listing 5.1: Simple program using all overridden features

The program creates two threads to wait until a global variable is at 3 and uses the `pthread_cond_wait` function. It creates three other threads to lock and unlock a mutex and increment the value of `g`. The main thread then increments `g` and waits for all of the other threads to join before terminating.

The main thing of note in the program is the `assert(false)` at the end of the program on line 52. The purpose of having this is to force KLEETHREADS to generate a schedule which will find this assertion failure so that we can generate longer schedules to test the program, rather than shorter schedules that only run to the first bug and no further.

KLEETHREADS generated 131 schedules leading to bugs when run on this program. We did not run P-Rep with every schedule but instead opted to test it with a variety of schedules a number of times. In total, we tested 20 of the generated schedules 10 times each and examined the results. We chose to run the tests 10 times each so that we could be more more confident that the tool was working correctly if it consistently followed the passed schedules. To determine which schedules we should use, we used a random number generator [1] to generate random numbers between 1 and 131 and then tested the schedules with the matching numbers. This was to guarantee fairness.

The results are shown in table 5.1. In all of the tests we ran, P-Rep followed the schedule as expected. This is expected and demonstrates that P-Rep works well in a predictable environment. It also gives us confidence that all of the different API calls had been implemented correctly. This was demonstrated by P-Rep's accuracy.

5.2.1 Deadlock Tests

We also tested P-Rep with a program that leads to a deadlock. This test was written in conjunction with KLEETHREADS and aims to force the program to deadlock.

The program is shown in the Appendix B.1. It is more complicated and diverse than the previous one and was written to test the limits of P-Rep by generating longer schedules to see what errors occurred under each condition.

Running KLEETHREADS with the following options: (-emit-all-errors -happens-before-cache -scheduler-preemption-bound=1 -fail-on-race) results in 49 schedules being generated.

Rather than using a random number generator to test the results, we opted to run tests on the first 10 schedules to check for any errors in both our recording process and P-Rep's execution order.

[1] http://www.random.org/

Table 5.1: Simple Scheduling Results

Test Number	**KLEE-THREADS Schedule**	**Schedule Followed**	**Followed Accurately**
test000040	0,0,0,0,2,1,1	0,0,0,0,2,1,1	Yes
test000087	0,0,0,3,0,0,3	0,0,0,3,0,0,3	Yes
test000002	0,1,1	0,1,1	Yes
test000096	0,0,0,0,2,0,2	0,0,0,0,2,0,2	Yes
test000042	0,0,0,0,3,0,1,1	0,0,0,0,3,0,1,1	Yes
test000027	0,0,0,3,3,3,2,2,2	0,0,0,3,3,3,2,2,2	Yes
test000043	0,0,0,0,3,0,1,1	0,0,0,0,3,0,1,1	Yes
test000131	0,0,0,0,0,4,3,2,2	0,0,0,0,0,4,3,2,2	Yes
test000130	0,0,0,0,0,4,3,1,1	0,0,0,0,0,4,3,1,1	Yes
test000021	0,0,0,3,0,0,1,1	0,0,0,3,0,0,1,1	Yes
test000067	0,1,0,0,0,0,2,2	0,1,0,0,0,0,2,2	Yes
test000036	0,0,0,0,1,0,1	0,0,0,0,1,0,1	Yes
test000095	0,0,0,0,1,0,4,4	0,0,0,0,1,0,4,4	Yes
test000063	0,0,0,0,0,1,3,2,2	0,0,0,0,0,1,3,2,2	Yes
test000006	0,0,0,0,3,3	0,0,0,0,3,3	Yes
test000008	0,0,0,0,0,2,2	0,0,0,0,0,2,2	Yes
test000105	0,0,0,0,0,2,3,4,4	0,0,0,0,0,2,3,4,4	Yes
test000077	0,0,0,1,3,3,3,3,3,2,2	0,0,0,1,3,3,3,3,3,2,2	Yes
test000013	0,0,0,0,0,1,4,4	0,0,0,0,0,1,4,4	Yes
test000111	0,0,0,0,0,2,1,2	0,0,0,0,0,2,1,2	Yes

Table 5.2: Deadlock Schedule Results

Test Number	**KLEE-THREADS Schedule**	**Schedule Followed**	**Followed**
test000001	0,0,0,1,1,1,1,2,2,2,2,2,2,3,3, 3,3,3,3,4,4,4,4,4,4,0,0,1,1,1,0	0,0,0,1,1,1,1,2,2,2,2,2,2,3,3,3,3, 3,3,4,4,4,4,4,4,0,0,1,1,1,0	Yes
test000002	0,1,1,1,2,2,2,2,2,2,3,3,3,3, 3,3,0,0,1,1,1,4,4,4,4,4,4,0,0	0,1,1,1,2,2,2,2,2,2,3,3,3,3,3, 3,0,0,1,1,1,4,4,4,4,4,4,0,0	Yes
test000003	0,0,2,2,2,2,2,0,0,1,1,1,3,3, 3,3,3,3,4,4,4,4,4,4,1,1,1,0	0,0,2,2,2,2,2,0,0,1,1,1,3, 3,3,3,3,3,4,4,4,4,4,4,1,1,1,0	Yes
test000004	0,0,0,2,2,2,2,2,2,0,0,1,1,1,1, 3,3,3,3,3,3,4,4,4,4,4,4,1,1,1,0	0,0,0,2,2,2,2,2,2,0,0,1,1,1,1,3,3, 3,3,3,3,4,4,4,4,4,4,1,1,1,0	Yes
test000005	0,0,0,1,1,2,2,2,2,2,2,3,3,3,3, 3,0,0,1,1,1,4,4,4,4,4,4,1,1,0	0,0,0,1,1,2,2,2,2,2,2,3,3,3, 3,3,0,0,1,1,1,4,4,4,4,4,4,1,1,0	Yes
test000006	0,0,0,1,1,3,3,3,3,3,3,1,1,1,2,2, 2,2,2,2,4,4,4,4,4,4,0,0,1,1,0	0,0,0,1,1,3,3,3,3,3,3,1,1,1, 2,2,2,2,2,2,4,4,4,4,4,4,0,0,1,1,0	Yes
test000007	0,0,0,1,1,1,3,3,3,3,3,3,4,4,4, 4,4,4,1,1,1,2,2,2,2,2,2,0,0	0,0,0,1,1,1,3,3,3,3,3,3,4,4,4, 4,4,4,1,1,1,2,2,2,2,2,2,0,0	Yes
test000008	0,0,0,1,1,1,4,4,4,4,4,4,1,2,2,2,2,2, 2,3,3,3,3,3,0,0,1,1,1,0	0,0,0,1,1,1,4,4,4,4,4,4,1,2,2,2,2,2, 2,3,3,3,3,3,0,0,1,1,1,0	Yes
test000009	0,0,0,1,1,1,3,3,3,3,3,3,4,4,4,4,4, 4,1,1,1,2,2,2,2,2,2,0,0	0,0,0,1,1,1,3,3,3,3,3,3,4,4,4,4,4,4, 1,1,1,2,2,2,2,2,2,0,0	Yes
test000010	0,0,0,1,1,1,1,4,4,4,4,4,2,2,2,2,2,2, 3,3,3,3,3,3,0,0,1,1,1,0	0,0,0,1,1,1,1,4,4,4,4,4,2,2,2,2,2, 2,3,3,3,3,3,3,0,0,1,1,1,0	Yes

As can be observed from the results in 5.2, P-Rep performed well under harsher test conditions. With a longer schedule, P-Rep succesfully followed it until the end and performed well with the more complex execution structure and P-Rep was able to maintain the integrity of the thread map, and was able to start and stop threads on demand.

These tests demonstrate that P-Rep is able to continue performing and providing deterministic replay with arbtitrarily long schedules. This is a critical feature of P-Rep, since more complex schedules will inevitably produce much larger execution schedules and we would want P-Rep to perform well under a multitude of conditions.

We can also observe from these results that P-Rep is accurately following the schedules it has been passed, rather than following some other thread ordering. It is essential that P-Rep follows its passed schedule accurately in order to provide deterministic replay. These results above demonstrate P-Rep's effectiveness.

5.2.2 KLEETHREADS Test Suite

We used tests from the KLEETHREADS test suite as part of our test-driven development. These tests were writting primarily to test the effectiveness of KLEETHREADS and mostly consist of simple tests using threads to perform basic operations. We tested on the hbgraphtests, all of which are shown in Appendix B (B.4, B.5,B.6,B.7) to ascertain the effectiveness of the tool and determine what changes we needed to make as we went along.

P-Rep successfully followed all of the schedules that KLEETHREADS output in the various hbgraphtests in its default mode of execution, its FOLLOW_AND_TERMINATE mode and its FOLLOW_TO_END mode. These results were reassuring since they showed us that the basic features of the tool were working before we moved onto the more complex tests. For the complete listing of the programs we tested see Appendix B, (B.4, B.5,B.6,B.7).

5.2.3 Testing RECORD mode

To test the record mode, we wrote a simple program that might lead to a deadlock under certain scenarios. We then ran the program in P-Rep's record mode to record a schedule that led to a deadlock. The program did not always lead to a deadlock, so we had to run P-Rep in record mode several times with the tool to ensure we reached a deadlock. When we reached a deadlock we could then generate a schedule that led to the deadlock with P-Rep. Once we had generated this schedule, we could rerun the program with this schedule and see if it led to a deadlock. If by running P-Rep with the generated schedule we recorded the same output, then we could assume that P-Rep's RECORD mode was working correctly.

To perform this test we used a version of the classic Dining Philosophers problem [18]. We also tested the record mode with the Producer/Consumers problem.

Our test conditions for both were as follows. We first checked to see if P-Rep did detect a deadlock when the deadlock occurred, we then checked to see if P-Rep faithfully followed the schedule that led to a deadlock. Most of the scenarios that we ran created schedules that were up to 200 steps long. This was advantageous for us since it meant that we could simlutaneously test the replay robustness of P-Rep with longer schedules as well as assess P-Rep's performance with longer schedules.

The implementation of the Dining Philosophers problem which we use is shown in Appendix B B.2. We had to run the program several times to ensure that we eventually had a deadlock. We ran five tests with different schedules P-Rep found to ensure that each one did indeed lead to a deadlock.

From the resutls shown in listing 5.3, we can see on the 4th and 5th schedule that we ran, P-Rep did not reach a deadlock. This suggests that the record mode of P-Rep is not as precise in its recording as it could be.

Since the program itself is not data race free, there may have been very specific synchronizations that it missed and thus resulted in the produced schedule not being entirely accurate. A program with data races would cause problems for our tool's record mode, since it would not be able to accurately detect the precise order of thread execution in the situation in which the race occurred. By their very nature, data races are indeterminate, so our tool would only be able to provide a certain level of determinism in such a data race. As a result of this nondeterminism, our tool might output a schedule that was slightly different from the actual order of execution, thus resulting in a different execution path. This is why P-Rep did not deadlock on every occasion.

Table 5.3: Dining Philosophers Record Results

Schedule Found	**Deadlock Reached**
0,1,1,0,2,0,0,0,0,1,1,2,3,3,4,1,1,5,5,3,3,4,2,3,2,3,2,1,1,2,1,1	Yes
0,0,0,1,1,2,0,0,0,3,3,1,1,2,1,1,4,5,5,3,3,4,3, 2,2,3,2,1,1,2,1,5,5,1,4,4,5,5,4,3,3,4,3,2,2,3,2,2	Yes
0,1,1,0,0,2,0,0,0,1,1,1,1,3,3,4,1,1,1,1,3,3,4,5, 1,1,3,2,3,4,4,5,1,4,5,5,3,3,4,1,3,2,2,4,4,3,2,5,5,4,1, 1,2,1,3,3,4,3,5,5,1,2,2,2,2,4,4,2,2,2,2,5,5,4,4,2,2,2,2,3	Yes
0,1,1,0,2,0,0,0,0,1,1,1,1,3,3,4,5,5,3,3,4,1,1,1,1, 3,3,1,1,1,2,5,5,1,4,4,5,5,3,3,4,4,3,2,2,2,2,2,2,2,2,2,2,3,2,2	Yes
0,0,2,1,1,2,0,0,0,0,2,2,4,4,3,3,2,5,1,1,2,1,1,4,4,5,3,3,4,4,3,2,2,3,2,2	Yes

This shows us that while our tool can be effective in its execution traces, it also can produce false positives. The final schedules that P-Rep produces, however, should not be that different from the actual execution that P-Rep followed, since it will have only have been the slightest misordering that would have led to a different schedule.

Where our tool would not be effective would be in a program that does not protect its shared variables with the Pthreads API.

For example consider the following code snippet:

```c
int j = 10;

void* alter(void* p)
{
  j = p;
}
```

Listing 5.2: Unprotected Shared Variable

Assume that one or more threads are running the alter function in this execution. As you can see the global variable `j` is not protected by any mutexes, therefore the threads that alter this variable will not be recorded and this race conditon will be completely ignored by our tool. Since our tool is meant to be primarily a replay tool used for debugging purposes and is primarily intended to aid in debugging programs that make use of the Pthreads API, we do not consider this that much of an issue for P-Rep in its current state. The purpose of this project has not been to implement a robust record tool, but rather a sturdy replay tool and this is something we have achieved.

We also tested P-Rep's record feature with the Producers/Consumers Problem. The implementation we tested is shown in Appendix B, B.3.

The difference between this program and the Dining Philosophers program is that it will always deadlock. Despite this, it will not necessarily be the same execution trace that leads to the deadlock and as can be observed from our reuslts, our tool found a different schedule for each deadlock it encountered.

Table 5.4: Producers/Consumers Record Results

Schedule Found	**Deadlock Reached**
0,1,1,1,0,0,3,3,2,2,2,3,3,0,0,3,3,0,0,0,0,0,0,0, 0,1,1,2,1,2,2,3,3,3,3,3,2,1,2,2,1,1,2,2,3,1,1,3,3,3,3,3	Yes
0,1,1,1,0,2,2,2,0,0,0,3,3,3,3,3,3,0,0,0,0,0,0,0, 0,1,2,1,1,2,2,3,3,3,3,3,1,2,1,1,2,2,1,1,2,2,3,3,3,3,3,3	Yes
0,1,1,1,0,2,2,2,0,3,0,3,0,3,3,3,0,0,3,0,0,0,0,0,0, 1,2,1,1,2,2,3,3,3,3,3,1,2,1,1,2,2,1,1,2,2,3,3,3,3,3,3	Yes
0,1,1,0,2,0,0,0,0,1,1,1,1,3,3,4,5,5,3,3,4,1,1,1,1, 3,3,1,1,1,2,5,5,1,4,4,5,5,3,3,4,4,3,2,2,2,2,2,2,2,2,2,2,3,2,2	Yes
0,1,1,1,0,2,2,2,0,0,0,3,3,0,0,3,3,0,0,3,3,0,0,0,0, 1,1,1,2,2,2,3,3,3,3,3,2,1,2,2,1,1,2,2,1,1,3,3,3,3,3,3	Yes

As we can see from the results shown in table 5.2.3, P-Rep was much more effective in producing schedules that led to a deadlock with this program. The reason for this is that there are no data races in the program and therefore P-Rep is able to produce a much more accurate execution trace and one that always results in a deadlock at the end of the schedule.

5.3 Efficiency Tests

We have conducted efficiency tests to show how P-Rep performs under various situations, but also to demonstrate the minimal overhead that is incurred by using function interception. A system call is typically expensive for a program to make anyway, so intercepting these system calls and forcing certain schedules to be followed incurs a minimal additional overhead to the original system call.

5.3.1 Comparison of Wrapper Library and Function Intercepting Efficiency

There appears to be a small difference in efficiency between using a wrapper library and function intercepting to implement P-Rep. In order to measure a difference between the two implementations of P-Rep we wrote a program that creates four threads each running the same function. In the function, the threads must lock a mutex, modify a global variable, open a file, write to it and close it, and then open, write to and close another file 10000 times within a for loop. After exiting the for loop, the thread must unlock the mutex and return.

We wrote the programs in this way so that their would be a significant amount of computation for each thread to perform. If the programs terminated too quickly without performing some file I/O or some other expensive operation, the program's would be executed too quickly for us to time accurately.

The version using the wrapper library is shown below:

```
#include "wrapper.h"

/*global schedule*/
int schedule[] = {
  #include "schedule#3.h"
};

pthread_mutex_t mut;
int GLOB = 0;
void* fn(void*);
int main()
{

  pthread_t a = intercept_pthread_create((void*)fn, (void*)1);
  pthread_t b = intercept_pthread_create((void*)fn, (void*)2);
  pthread_t c = intercept_pthread_create((void*)fn, (void*)3);
  pthread_t d = intercept_pthread_create((void*)fn, (void*)4);
  intercept_pthread_join(a, NULL);
  intercept_pthread_join(b, NULL);
  intercept_pthread_join(c, NULL);
  intercept_pthread_join(d, NULL);
  fprintf(stderr, "GLOB: %d\n", GLOB);
  return 0;
}
void* fn(void* a)
{
  int i = (int) a;
  intercept_mutex_lock(&mut);
  GLOB = i;
  FILE *f = fopen("something.txt", "r");
  fputc(i, f);
  fclose(f);
  FILE* g = fopen("another.txt", "r");
  int j;
  for(j = 0; j < 10000; ++j)
    {
      fclose(g);
      g = fopen("another.txt", "r");
      fputc(j, g);
    }
  fclose(g);
  intercept_mutex_unlock(&mut);
  return 0;
}
```

Listing 5.3: Wrapper Library Speed Test

The version using function intercepting in shown below:

```
#include <pthread.h>
#include <stdio.h>

int GLOB = 0;
void* fn(void*);

pthread_mutex_t mut;

int main()
{
  pthread_t a,b,c,d;
  pthread_create(&a, NULL, fn, (void*)1);
  pthread_create(&b, NULL, fn, (void*)2);
  pthread_create(&c, NULL, fn, (void*)3);
  pthread_create(&d, NULL, fn, (void*)4);


  pthread_join(a, NULL);
  pthread_join(b, NULL);
  pthread_join(c, NULL);
  pthread_join(d, NULL);


  fprintf(stderr, "GLOB: %d\n", GLOB);
  return 0;
}


/*
 *Make thread lock a mutex, open a file, write to the file and then
    change the
 *the global variable GLOB
 */
void* fn(void* a)
{
  int i = (int) a;
  pthread_mutex_lock(&mut);
  GLOB = i;
  FILE *f = fopen("something.txt", "r");
  fputc(i, f);
  fclose(f);
  FILE* g = fopen("another.txt", "r");
  int j;
  for(j = 0; j < 10000; ++j)
    {
      fclose(g);
      g = fopen("another.txt", "r");
      fputc(j, g);
    }
  fclose(g);
  pthread_mutex_unlock(&mut);
  return 0;
}
```

Listing 5.4: Function Interception Speed Test

Since the version of P-Rep which uses function intercepting is set up using a Python script, we set the environment variables manually for each execution so that there would be no overhead incurred by using the Python script. I/O also requires system calls so we executed both versions with no output to the standard output other than the final value of GLOB.

We ran the tests in P-Rep's normal mode of execution. The schedule we passed P-Rep was handwritten. We wrote the schedule by hand to ensure that the programs finished execution. KLEETHREADS would have generated a lot of possible execution paths for this program and the output might have varied for each version of P-Rep. By writing this schedule by hand we were able to guarantee that the schedule would work with both implementations and also that both versions would be executing in the same order.

We ran the program with the same schedule:

[0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,0,0,0,0,0,1,2,4,3,0,0,0,0,0,0,0,0]

10 times with each implementation of P-Rep to see which version of P-Rep was faster on average.

The average execution speeds are shown below, for the full results see Appendix A, tables A.2 and A.1:

Table 5.5: Speed Test Results

Test	**Real Time**	**User Time**	**Sys Time**
Wrapper Library	12.141	0.118	1.136
Function Intercepting	13.192	0.172	1.648

As we can see the version of P-Rep which uses the wrapper library is marginally faster than the version which uses function interception. We put this difference in speed down to the fact that the program that uses our shared library has to load the library and make additional calls to `dlsym()` to override the functionality of the intercepted Pthreads functions, while the version of P-Rep which uses the wrapper library only needs to lookup statically linked functions.

While the wrapper library is slightly faster, the difference is negligible and we still maintain that the function interception version of P-Rep, we still maintain that implementing P-Rep as a shim is much more effective than implementing it as a wrapper libary. While the wrapper library is faster, it lacks the immediate convenience of the version which uses function intercepting. To use the wrapper library in a production environment, we would have to instrument the user's source code to replace their calls to the Pthreads API with our calls to the wrapper library. We would also be required to to recompile all the programs that used the Pthreads API, so that they made use of our wrapper library instead of the original Pthreads API. For this reason, using function interception is far more desirable than using a wrapper library.

5.4 Portability

Perhaps one of the main problems with P-Rep in its current state is portability. While P-Rep has been designed to work with the Pthreads API and thus theoretically with any POSIX compliant machine. This is not the case unfortunately.

5.4.1 Mac OS X

On a Mac, one of the issues was that the OS X operating system keeps a system implementation of the `wait()` function. This caused our `wait()` function to be ignored since `wait()` has already been defined. We fixed this by simply redefining the `wait()` function to be `my_wait()`.

Another problem we faced on the Mac was in compilation with finding the location of "malloc.h". To fix this all we needed to do alter our Makefile to specify the correct location of "malloc.h"

While these two issues were trivial to fix, we had further problems with compilation in `pthread_tryjoin_np`. As expected, this function is not portable and has no equivalent implementation on the OS X operating system.

The function `dlvsym()` is also not supported on the Mac operating system. Rewriting P-Rep to use `dlsym()` in the `pthread_cond_wait` caused the same problem we had when originally implementing the `pthread_cond_wait` function in section: 4.3.3.

5.4.2 BSD

We used FreeBSD 9.0 runnning in VirtualBox to test our tool's portability to a BSD system. The GNU C Libary (GLIBC) is not supported on FreeBSD so not all of our tool's features are supported on this platform. Specifically `pthread_tryjoin_np` is not supported on the FreeBSD platform, so not all of the functionality of our tool was available on the FreeBSD system we tested with.

After removing our implementations of `pthread_join`, we were able to get P-Rep working on the virtualised FreeBSD system. All of the other overridden Pthread functions worked well on the FreeBSD operating system and this was an encouraging results.

While not every feature of our tool could be implemented on a FreeBSD system, we were nevertheless able to get P-Rep working for the Pthread functions that were portable and P-Rep worked as expected for these functions.

5.5 Overall Evaluation

P-Rep is most effective when used in its replay mode. As we have shown from the results above, it will accurately follow any schedule that it is passed. We believe that the reason for this effectiveness is in its simplicity. By using only one global semaphore to schedule threads, we eliminate the complexity that would be involved in manually interleaving thread execution.

P-Rep is able to replay a full schedule for any concurrent C program that uses the Pthreads API and makes use of the functions we have overridden. Many programmers only use a small fraction of the Pthreads API, making use of mutexes and conidtion variables. Therefore, our tool can be used with many existing concurrent C programs to replay any given schedule it is passed.

While our tool is primarily designed to work with KLEETHREADS, we have shown that it will in fact follow any schedule it is passed accurately. This is a highly desirable feature, since it means it can be used in a number of environments. A programmer might wish to incrementally observe the output of their program by slowly incrementing a schedule and observing the different states the program enters. Our tool can be used in conjunction with `gdb` by manually setting the environment variables to point to the schedule location and the LD_PRELOAD environment variable. Using this technique a programmer can debug their program normally using breakpoints and other tools, while forcing it to follow a particular thread schedule. Once the end of the schedule is reached, the programmer can observe and record the program's state.

Having the ability to maintain such fine-grained control over a concurrent program's execution state is highly desirable, since normally when debugging with a tool like `gdb` the program's thread ordering will be completely different in each execution. Also the debugger might slow down the execution speed, resulting in data races being missed or occurring in a different order or not occurring at all. Our tool removes this nondeterminism and adds a level of complete determinism to concurrent software, which provides a programmer with an invaluable understanding into the execution of their program.

Finally, we have demonstrated our tool to be higly robust in its replay feature. We have shown that it scales well under a number of complex scenarios and is able to follow longer schedules with minimal overhead. The fact that all of this can be achieved with absolutely no modification to the original executables is highly encouraging since it means our tool can be used ‘out of the box,’ as it were, and a programmer can immediately start testing schedules and observing results on any of their existing programs that use the Pthreads API.

Our tool provide determinism to nondeterministic programs and does so in a robust, scaleable and simple fashion.

Chapter 6

Conclusion

6.1 Achievements

We have achieved all of the initial goals for the project as well as adding a record mode to detect a deadlock and record the schedule that led to that deadlock.

What is unique about our tool is that it can be run either with tools that provide execution traces to programs or independently with any schedule the programmer chooses to provide. Using P-Rep solely for its replay feature it particularly useful when attempting to find concurrent software bugs. The programmer can slowly increment their schedule and keep re-running their program with our tool to observe how different schedules will effect the overall outcome of their program.

Combined with its FOLLOW_AND_RESUME mode (in which P-Rep runs to the end of a schedule and resumes normal concurrent execution) this makes P-Rep even more useful since a programmer can force a certain schedule to be followed up to a certain point and then observe how initial deterministic thread scheduling can effect the final non-deterministic state of a program.

Even multi-threaded programs that are data-race free can have non-deterministic behaviour, since we cannot be sure which threads are executing when. Our tool provides an excellent way to provide determinism to these non-deterministic programs and thus can be used to trace all sorts of bugs, not just traditional concurrency bugs. For example, a programmer may have trouble tracing a segmentation fault that only occurs under certain conditions. By using our tool and gradually building a schedule to experiment with different execution routes, the programmer will be able to track down and fix the segmentation fault, or any other bug.

We believe that this the most unique property of our tool. The ability to add deterministic replay to concurrent C programs independent of any record tools, is something which is of great use to the developer. It provides them with a toolbox to find and fix bugs without having to recompile or relink their executables. This is something that will prove more useful for larger programs or programs with longer compile times.

6.2 Future Work

The most obvious extension to P-Rep is to finish implementing the rest of the Pthreads API. While it is a large API, we think implementing the whole API is feasible and would result in a robust and reliable tool that could be used with all concurrent C programs that use the Pthreads API to provide deterministic replay. In combination with KLEETHREADS, this would a create a full record/replay suite for concurrent C programs and would prove an invaluable tool to developers developing concurrent software in C.

Another possible extension to P-Rep would be to give a more thorough execution trace to the bug. In its current state, P-Rep does not really provide too in-depth a trace after termination to the bug. So as an added, extension it would be desirable to provide the user with a stack trace or clear path to the fault after the program's termination.

In addition, when P-Rep runs in its default execution state, it simply hangs once it reaches the end of a schedule and this is somewhat unhelpful for users of P-Rep who might want to print a stack trace to help determine the state of variables at the program's termination. To this end, we think it would desirable to work towards integrating P-Rep with an existing debugger such as `gdb`. This would be highly desirable since it would provide the programmer with the ability to trace concurrency bugs but also to use all of the features P-Reps `gdb` provides as well.

The final extension which we considered but did not have time to implement was extending P-Rep's record feature to make it more robust in programs that have data races. In its current state the record mode will detect a deadlock and output the schedule that led to the deadlock. The schedule it produces will not always be accurate if the program has data races, since by their very nature the specific thread ordering of data races is difficult to trace. P-Rep will only offer a trace to data races if the variable is protected by a mutex or a condition variable. Variables that are not protected by the Pthreads API will be ignored, since P-Rep will only log the thread ordering if a thread calls the `step_and_notify()` function. If a data race occurs with no synchronisation then P-Rep will not notice it and the trace it produced might not accurately represent the actual schedule of execution. Fixing this would be a desirable extension to P-Rep, as it would add a full record feature. We cannot, however, intercept a data race, so the only possible way to add this feature to the tool would be to use traditional static analysis techniques to detect the data races. For this reason, we believe it would be better to implement a solid replay feature for the rest of the Pthreads API before moving on to creating a robust record feature.

Chapter 7

Bibliography

[1] T. D. LaToza and B. A. Myers, "Designing Useful Tools for Developers," in *PLATEAU*, pp. 45–50, 2011.

[2] H. Sutter and J. R. Larus, "Software and the Concurrency Revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[3] N. G. Leveson and C. S. Turner, "Investigation of the Therac-25 Accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[4] E. G. C. Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, pp. 209–224, 2008.

[6] P. Thomson, "Kleerace: Algorithms and Heuristics for Scalable Data-dependent Race Detection using Symbolic Execution," Master's thesis, University of Oxford, Oxford, United Kingdom, 2011.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," in *SOSP*, pp. 27–37, 1997.

[8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[9] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI*, pp. 89–100, 2007.

[10] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, pp. 190–200, 2005.

[11] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer - data race detection in practice," 2009.

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *OSDI* (R. Draves and R. van Renesse, eds.), pp. 267–280, USENIX Association, 2008.

[13] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs," in *SIGSOFT FSE*, pp. 207–216, 2010.

[14] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 48–59, 1998.

[15] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," in *SOSP*, pp. 193–206, 2009.

[16] D. S. Myers and A. L. Bazinet, "Intercepting Arbitrary Functions on Windows, UNIX and Macintosh OS X Platforms," 2004.

[17] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," 1999.

[18] C. A. R. Hoare, "Process Algebra: A Unifying Approach," in *25 Years Communicating Sequential Processes*, pp. 36–60, 2004.

Appendix A

Full Test Results

Table A.1: Functions Intercepting Speed Test Results

Test Number	**Real Time**	**User Time**	**Sys Time**
1.	12.44	0.17	1.81
2.	13.59	0.22	1.84
3.	12.99	0.22	1.79
4.	14.03	0.21	1.82
5.	12.76	0.26	1.73
6.	13.29	0.18	1.84
7.	13.48	0.14	1.90
8.	13.04	0.15	1.85
9.	13.21	0.19	1.83
10.	13.09	0.15	1.88

Table A.2: Wrapper Library Speed Test Results

Test Number	**Real Time**	**User Time**	**Sys Time**
1.	12.02	0.11	1.02
2.	12.53	0.06	1.29
3.	12.04	0.10	1.01
4.	11.90	0.07	1.02
5.	12.20	0.12	1.04
6.	12.49	0.13	0.92
7.	12.12	0.12	1.06
8.	11.80	0.18	0.98
9.	12.02	0.14	1.34
10.	12.29	0.15	1.68

Appendix B

Full Program Listings

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

#define NUM_THREADS1 (2)
#define NUM_THREADS2 (1)

pthread_mutex_t mutexA;
pthread_mutex_t mutexB;

void* locker1(void* args)
{
  pthread_mutex_lock(&mutexA);
  pthread_mutex_lock(&mutexB);
  pthread_mutex_unlock(&mutexB);
  pthread_mutex_unlock(&mutexA);
  return 0;
}

void* locker2(void* args)
{
  pthread_mutex_lock(&mutexB);
  pthread_mutex_lock(&mutexA);
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_unlock(&mutexB);
  return 0;
}

void* t1(void* args)
{
  pthread_t otherHandles[NUM_THREADS1];
  size_t i;

  for(i=0; i < NUM_THREADS1; ++i)
  {
    pthread_create(&otherHandles[i], NULL, locker2, NULL);
  }

  for(i=0; i < NUM_THREADS1; ++i)
  {
    pthread_join(otherHandles[i], NULL);
  }

  return 0;
}

int main(int argc, char** argv)
{
  pthread_mutex_init(&mutexA, NULL);
  pthread_mutex_init(&mutexB, NULL);

  pthread_t handle;

  pthread_create(&handle, NULL, t1, NULL);

  pthread_t handles[NUM_THREADS2];
  size_t i;
  for(i=0; i < NUM_THREADS2; ++i)
  {
    pthread_create(&handles[i], NULL, locker1, NULL);
  }

  for(i=0; i < NUM_THREADS2; ++i)
  {
    pthread_join(handles[i], NULL);
  }

  pthread_join(handle, NULL);

  return 0;
}
```

Listing B.1: A program leading to deadlock

```
#include <sys/time.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

#define NUMP 5

pthread_mutex_t fork_mutex[NUMP];

int main()
{
  int i;
  pthread_t diner_thread[NUMP];
  int dn[NUMP];
  void *diner();
  for (i=0;i<NUMP;i++)
    pthread_mutex_init(&fork_mutex[i], NULL);

   for (i=0;i<NUMP;i++){
      dn[i] = i;
      pthread_create(&diner_thread[i],NULL,diner,&dn[i]);
   }
   for (i=0;i<NUMP;i++)
      pthread_join(diner_thread[i],NULL);

   return 0;
}

void *diner(int *i)
{
int v;
int eating = 0;
printf("I'm diner %d\n",*i);
v = *i;
while (eating < 5) {
   printf("%d is thinking\n", v);
   sleep( v/2);
   printf("%d is hungry\n", v);
   pthread_mutex_lock(&fork_mutex[v]);
   pthread_mutex_lock(&fork_mutex[(v+1)%NUMP]);
   printf("%d is eating\n", v);
   eating++;
   sleep(1);
   printf("%d is done eating\n", v);
   pthread_mutex_unlock(&fork_mutex[v]);
   pthread_mutex_unlock(&fork_mutex[(v+1)%NUMP]);
}
}
```

Listing B.2: Dining Philosophers Problem

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>
#include<sys/time.h>

int     q[5];
int     first  = 0;
int last   = 0;
int     numInQ = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t empty = PTHREAD_MUTEX_INITIALIZER;

void putOnQ(int x) {
        pthread_mutex_lock(&mutex);
        q[first] = 1;
        first = (first+1) % 5;
        numInQ++;
        pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&empty);
}

int getOffQ(void) {
        int thing;
        while (numInQ == 0) pthread_mutex_lock(&empty);
    pthread_mutex_lock(&mutex);
    thing = q[last];
    last = (last+1) % 5;
    numInQ--;
    pthread_mutex_unlock(&mutex);
        return thing;
}

void * prod(void *arg) {
        int     mynum;
        int     i;
        mynum = *(int *)arg;
        for (i = 0; i < 3; i++) {
                sleep(i);
                puts("putting");
                putOnQ(i+100*mynum);
                printf("put %d on queue\n", i+100*mynum);
        }
 }

void * con(void *arg) {
        int     i;
        int     stuff;
        for (i = 0; i < 6; i++) {
                puts("getting");
                stuff = getOffQ();
                printf("got %d from queue\n", stuff);
        }
 }

int main()
{
 pthread_t threadp1, threadp2, threadc1;
 int zero = 0;  int one = 1;

 pthread_create(&threadp1, NULL, prod, &zero);
 pthread_create(&threadp2, NULL, prod, &one);
 pthread_create(&threadc1, NULL, con,  NULL);

 return 0;
}
```

Listing B.3: Producers/Consumers Problem

```
// --scheduler-delay-bound=100
//KLEE: done: completed paths = 3
//KLEE: done: num paths considering hb graph cache = 1

#include <pthread.h>
#include <assert.h>
#include <stdio.h>

```

```
int klee_fork(int reason);

//pthread_mutex_t mutex;

void* slave(void* args)
{

}

void* slave2(void* args)
{
}

int main(int argc, char** argv)
{
  pthread_t handle;
  pthread_t handle2;

  pthread_create(&handle, NULL, slave, NULL);
  //pthread_create(&handle2, NULL, slave2, NULL);

  pthread_join(handle, NULL);
  //pthread_join(handle2, NULL);

  return 0;
}
```

Listing B.4: HB Graph Test 1

```
// --scheduler-preemption-bound=0
//KLEE: done: completed paths = 3
//KLEE: done: num paths considering hb graph cache = 1

#include <pthread.h>
#include <assert.h>
#include <stdio.h>

int klee_fork(int reason);

//pthread_mutex_t mutex;

void* slave(void* args)
{

}

void* slave2(void* args)
{
}

int main(int argc, char** argv)
{
  pthread_t handle;
  pthread_t handle2;

  pthread_create(&handle, NULL, slave, NULL);
  pthread_create(&handle2, NULL, slave2, NULL);

  pthread_join(handle, NULL);
  pthread_join(handle2, NULL);

  return 0;
}
```

Listing B.5: HB Graph Test 2

```
// --scheduler-preemption-bound=1
//KLEE: done: completed paths = 4
//KLEE: done: num paths considering hb graph cache = 2

#include <pthread.h>
#include <assert.h>
#include <stdio.h>

int klee_fork(int reason);

pthread_mutex_t m;
int g;
void* slave(void* args)
{
  pthread_mutex_lock(&m);
  pthread_mutex_unlock(&m);
}

void* slave2(void* args)
{
}

int main(int argc, char** argv)
{
  pthread_t handle;
  pthread_t handle2;
  pthread_mutex_init(&m, NULL);

  pthread_create(&handle, NULL, slave, NULL);
  //pthread_create(&handle2, NULL, slave2, NULL);
  pthread_mutex_lock(&m);
  pthread_mutex_unlock(&m);

  pthread_join(handle, NULL);
  //pthread_join(handle2, NULL);

  pthread_mutex_destroy(&m);
  return 0;
}
```

Listing B.6: HB Graph Test 3

```
// --scheduler-preemption-bound=-1
//KLEE: done: completed paths = 12
//KLEE: done: num paths considering hb graph cache = 4

// 1 non-det choice in each thread.

#include <pthread.h>
#include <assert.h>
#include <stdio.h>

void klee_make_symbolic(void *addr, size_t nbytes, const char *name)
  ;

int klee_fork(int reason);

//pthread_mutex_t mutex;

void* slave(void* args)
{
  volatile int b;
  klee_make_symbolic((void*)&b, sizeof(b), 0);

  if(b == 0)
  {
    b = 1;
  }
  else
  {
    b = 2;
  }
}

int main(int argc, char** argv)
{
  pthread_t handle;

  pthread_create(&handle, NULL, slave, NULL);

  volatile int a;
  klee_make_symbolic((void*)&a, sizeof(a), 0);

  if(a == 0)
  {
    a = 1;
  }
  else
  {
    a = 2;
  }



  pthread_join(handle, NULL);

  return 0;
}
```

Listing B.7: HB Graph Test 4

```
import os
import subprocess
import sys

def parseExecutionMode(N):
    if N == "-e":
        os.environ['FOLLOW_TO_END'] = "YES"
        print "Running in FOLLOW_TO_END mode ....\n"
        return N
    elif N == "-t":
        os.environ['FOLLOW_AND_TERMINATE'] = "yes"
        print "Running in FOLLOW_AND_TERMINATE mode ....\n"
        return N
    elif N == "-r":
        os.environ['RECORD_MODE'] = "yes"
        print "Running in RECORD_MODE ...\n"
        return N
    elif N == "-n":
        print "Running is default execution mode ...\n"
        return N
    elif N == "-m":
        print "Running in monitor mode ...\n"
        os.environ['MONITOR_MODE'] = "yes"
        return N
    else:
        print "No such mode of execution exists"
        return N

record = False
commarg1 = False

if len(sys.argv) < 2:
    print "Please specify an executable"
    sys.exit(1)
elif len(sys.argv) == 2:
    os.environ['RECORD_MODE'] = "yes"
    executable = sys.argv[1]
    record = True
elif len(sys.argv) == 3:
    schedule = sys.argv[2]
    executable = sys.argv[1]
elif len(sys.argv) == 4:
    schedule = sys.argv[2]
    executable = sys.argv[1]
    parseExecutionMode(sys.argv[3])
elif len(sys.argv) == 5:
    schedule = sys.argv[2]
    executable = sys.argv[1]
    parseExecutionMode(sys.argv[3])
    os.environ['KLEE_VERBOSE_MODE'] = "yes"
    print "Running in verbose mode....\n"
elif len(sys.argv) > 5:
    commarg1 = sys.argv[2]
    schedule = sys.argv[3]
    executable = sys.argv[1]
    parseExecutionMode(sys.argv[4])
    os.environ['KLEE_VERBOSE_MODE'] = "yes"
    print "Running in verbose mode....\n"

#find out what shell the user is using
shell = os.environ['SHELL']
res = shell.split("/")
sh = res[-1]
absp = os.path.abspath('../bin')

#Set environment variable so that schedule can be accessed
   dynamically
if not record:
    schedloc = os.path.abspath('schedules');
    schedp = schedloc + "/" + schedule
    print schedp
    os.environ['KLEE_SCHED'] = schedp

#Set up the LD_PRELOAD variable to enable dynamic function
   interposition
ld_prel = absp + "/libintercept.so"

print "\nCalling executable....\n"
t = sh + " -c env "
path = "LD_PRELOAD=\"" + ld_prel + "\" " + "./" + executable
os.system(path)
```

Listing B.8: Python Script to Launch Tool