Record and vPlay: Problem Determination with Virtual Replay Across Heterogeneous Systems

Dinesh Subhraveti

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2012

ABSTRACT

Record and vPlay: Problem Determination with Virtual Replay Across Heterogeneous Systems

Dinesh Subhraveti

Application down time is one of the major reasons for revenue loss in the modern enterprise. While aggressive release schedules cause frail software to be released, application failures occurring in the field cost millions to the technical support organizations in personnel time. Since developers usually don't have direct access to the field environment for a variety of privacy and security reasons, problems are reproduced, analyzed and fixed in very different lab environments. However, the complexity and diversity of application environments make it difficult to accurately replicate the production environment. The indiscriminate collection of data provided by the bug reports often overwhelm or even mislead the developer. A typical issue requires time consuming rounds of clarifications and interactions with the end user, even after which the issue may not manifest.

This dissertation introduces vPlay, a software problem determination system which captures software bugs as they occur in the field into small and self-contained recordings, and allows them to be deterministically reproduced across different operating systems and heterogeneous environments. vPlay makes two key advances over the state of the art. First, the recorded bug can be reproduced in a completely different operating system environment without any kind of dependency on the source. vPlay packages up every piece of data necessary to correctly reproduce the bug on any stateless target machine in the developer environment, without the application, its binaries, and other support data. Second, the data captured by vPlay is small, typically amounting to a few megabytes. vPlay achieves this without requiring changes to the applications, base kernel or hardware.

vPlay employs a recording mechanism which provides data level independence between the application and its source environment by adopting a state machine model of the application to capture every piece of state accessed by the application. vPlay minimizes the size of the recording through a new technique called *partial checkpointing*, to efficiently capture the partial intermediate state of the application required to replay just the last few moments of its execution prior to the failure. The recorded state is saved as a *partial checkpoint* along with metadata representing the information specific to the source environment, such as calling convention used for the system calls on the source system, to make it portable across operating systems. A partial checkpoint is loaded by a partial checkpoint loader, which itself is designed to be portable across different operating systems. Partial checkpointing is combined with a logging mechanism, which monitors the application to identify and record relevant accessed state for root cause analysis and to record application's nondeterministic events.

vPlay introduces a new type of virtualization abstraction called vPlay Container, to natively replay an application built for one operating system on another. vPlay Container relies on the self-contained recording produced by vPlay to decouple the application from the target operating system environment in three key areas. The application is decoupled from (1) the address space and its content by transparently fulfilling its memory accesses, (2) the instructions and the processor MMU structures such as segment descriptor tables through a binary translation technique designed specifically for user application code, (3) the operating system interface and its services by abstracting the system call interface through emulation and replay. To facilitate root cause analysis, vPlay Container integrates with a standard debugger to enable the user to set breakpoints and single step the replayed execution of the application to examine the contents of variables and other program state at each source line.

We have implemented a vPlay prototype which can record unmodified Linux applications and natively replay them on different versions of Linux as well as Windows. Experiments with several applications including Apache and MySQL show that vPlay can reproduce real bugs and be used in production with modest recording overhead.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I attribute much of what I have learnt and accomplished over the graduate years to my advisor, Jason Nieh. I cannot overstate his influence on my life and my career. He provided the financial support, which made my higher education possible, and afforded me important opportunities that shaped my career. With patience, he mended my crude ways of early semesters and later accommodated me in the program despite the challenges of working with me remotely. His uncompromising demand for excellence pushed me far beyond I would have ever ventured on my own. His example is inspiring, yet inimitable. Each attempt and failure to emulate him taught me countless lessons and made me a better researcher.

My doctoral journey has been a long and windy one, and in its course, introduced me to many outstanding individuals from Columbia, HP, Meiosys and IBM, who contributed to this thesis in various ways. Many thanks to Nicolas Viennot and Oren Laadan, whose contribution is more than incidental. Their Scribe work forms the basis for a key piece of this dissertation. Oren's generous gesture at the crunch time of Sigmetrics submission will not be soon forgotten. Working with Steven Osman, Shaya Potter and Gong Su was always a pleasure. John Janakiraman, my HP manager, gave key insights about the industry and valuable exposure to my research within and outside HP. Renato Santos and Yoshio Turner provided treasured friendship and mentorship.

I would like to express my gratitude to a number of people at IBM whose support made this work possible. Mark Dean gave me the impetus to resume Ph.D. following the acquisition of Meiosys. Leslie Swanson, my IBM manager, provided unreserved support, encouragement and guidance, and sponsored the degree program semester after semester. Dilip Kandlur inspired me toward a "final push" for graduation and served on my dissertation committee. His counsel guided me both academically and professionally at IBM. I am immensely grateful to Gail Kaiser, Giovanni Pacifici and Junfeng Yang to have taken the time to be on my dissertation committee and provide insightful feedback. Sincere appreciation is due to Linda Duyanovich and Prasenjit Sarkar for their support during the final months of the dissertation. Good friends, Philippe Bergheaud and Marc Vertes made up for the lack of an on-campus support environment by acting as a sounding board during the formative stages of this work. I cherish the insightful discussions we had and the time we spent together at various IBM locations across the world. Giridhar Togarcheti, friend and colleague, shared an invaluable insider perspective of crisis management in a large technical support organization.

I am indebted to my parents, Shyamala and Sridhar Subhraveti, for the numerous sacrifices they made in bringing me up. My wife and best friend, Pallavi, took pride in my work and stood by me through the pressures of full-time work and full-time research. My little daughter Saeesha taught me with her innocence what truly matters in life.

This dissertation is dedicated to Bhagavan Sri Sathya Sai Baba, Whose eternal presence is the source of all inspiration.

Dinesh Subhraveti

COLUMBIA UNIVERSITY

December 2011

In loving surrender to Swami, in all His forms and formlessness.

Chapter 1

Introduction

As transaction volumes raise, even brief periods of application downtime lead to major revenue loss. When core business processes of a customer are suspended due to an application failure, quickly diagnosing the problem and putting the customer back in business is of utmost importance.

Resolving a problem that occurs in the field environment typically starts with reproducing it in the lab. Developers adopt a cyclic debugging technique, where the bug is repeatedly triggered to examine the application's internal state and its behavior from various angles to gain insight into the root cause of the failure. Once the symptom is reproduced, the developer is often able to visualize a solution. However, due to the heterogeneity of the application environments and nondeterministic factors, reproducing a software bug is one of the most time consuming and difficult steps in the resolution of a problem (Figure 1.1.)

Application environments are increasingly heterogeneous due to the large diversity of individual components that make up the software-hardware stack. A variety of operating systems, corresponding libraries and their many versions, application tiers supplied by different ISVs, network hardware with varied configuration settings etc. make application environments complex and bugs hard to reproduce. As open systems are favored, it is rare that a single vendor provides the complete software-hardware infrastructure. Users and platform distributions choose from a rich variety of components to provision their individual deployments. The diversity in the operating systems, middleware, network infrastructure etc. and their subtle differences often lead to unforeseen issues. The source of a problem

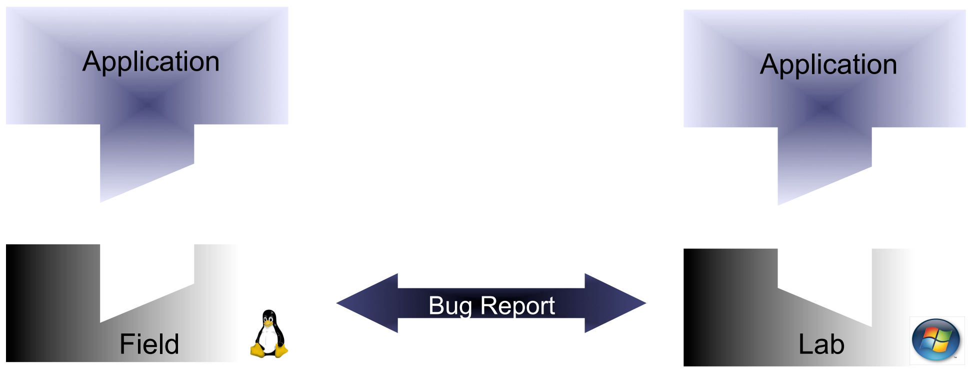


Figure 1.1: Reproducing problems is difficult due to differences in field and lab environments

could be an incorrect assumption implicitly made by the application about the availability or configuration of local services such as DNS, or about co-deployed applications and their components, or it may surface only when a particular library version is used [76]. For example, there may be a difference in the application binaries or other support libraries installed on the system. Dynamically linking with incompatible versions of libraries may cause applications to fail. Problems occur due to discrepancies in the configuration files on disk, differences in the kernel API, or unexpected responses received from other application tiers. Although software components are normally designed to conform to standards and pre-negotiated interfaces, discrepancies arise due to inaccurate implementations and incompatible versions.

The heterogeneity of application environments places a burden on the application developers to design applications to be portable. Each software component has to be able to work with all possible varieties of other components. Applications have to be portable not only across different hardware architectures and operating systems, but also across subtler differences in the computing environment. In order to correctly function in diverse environments, applications are often written to sense their target environment at runtime and adapt accordingly. However, thoroughly accounting for all possible combinations of target environments is impractical if not impossible. As a result, software often fails mysteriously when encountered with the alien customer environment.

Reproducing a bug can require creating the original field environment in the lab and providing the same set of inputs to the application. However, the large number of interacting software and hardware components and the complexity of their configuration make it difficult to accurately replicate the production environments in the lab. For example, reproducing a bug in the application server may require configuring a backend database and a web front end, which in itself, is a tedious and error prone process. Alternatively, the developer may directly access the production environment, but in practice a variety of privacy and security issues prevent the use of the production environment for debugging.

Nondeterminism adds to the problem. Factors such as timing, scheduling, signals, user inputs, inputs from the operating system may introduce unexpected execution paths by changing the interleavings of the application code and contribute to the difficulty in reproducing a problem. With multicore systems becoming pervasive, applications are turning to higher levels of parallelism to realize higher processing rates. Increased parallelism also increases application nondeterminism and the number of possible execution paths. Anticipating every type of input the application may receive in the field, and verifying each possible control path the application may take is challenging, if not impossible. As a result, even after extensive in-house testing, many crucial bugs are only detected by external users after the software is released.

Even though the complexity of applications and their susceptibility to errors is growing, the debugging methodology has not changed significantly in decades. When a failure occurs, the user typically provides a descriptive account of the problem to the developer through a bug report. The bug report consists of assorted pieces of information such as description of the status of the application at the time of the failure, the action being performed, the workload applied, specific failure inducing operations, any suspicious interactions with other applications etc. In many cases, most of the information presented to the developer may have little or nothing to do with the root cause of the bug. Since the end-user may not know the application internals and what is relevant for debugging, user supplied data may overwhelm or even mislead the developer. Typically a bug report has to be followed up with several rounds of exchange with the user.

Since the common approach of manually conveying a bug report is often inadequate, some application vendors [41; 45] provide built-in support for automatically collecting necessary information about the execution environment when a failure occurs. They conser-

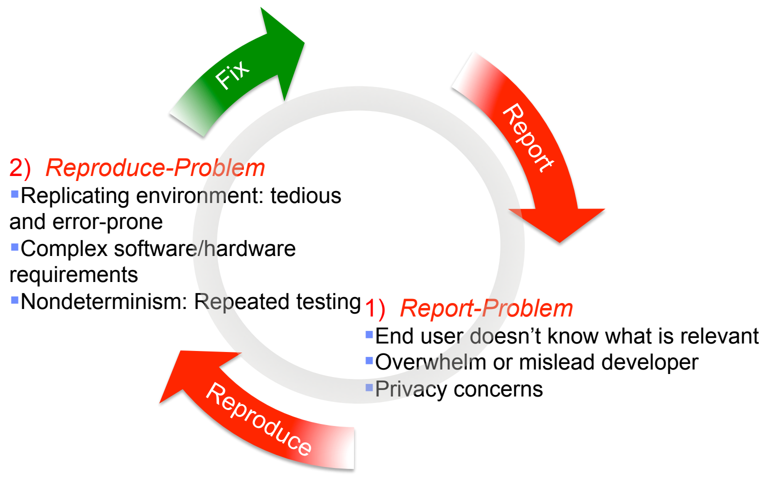

Fix
Report
2) Reproduce-Problem
Replicating environment: tedious and error-prone
Complex software/hardware requirements
Nondeterminism: Repeated testing
1) Report-Problem
End user doesn't know what is relevant
Overwhelm or mislead developer
Privacy concerns
Reproduce

Figure 1.2: Much time is spent in understanding the reported problem and reproducing it. Once reproduced, fixing is relatively easy

vatively record every piece of information that potentially has a bearing on the manifested problem, in an attempt to ensure that sufficient context is recorded to be able to reproduce the behavior offline and possibly fix it. Other sophisticated mechanisms [28] may provide more comprehensive data including traces and internal application state. However, they are limited in their ability to provide insight into the root cause of the problem because they represent the aftermath of the failure, not the steps that led to it. Furthermore, indiscriminate recording and transfer of client data evokes privacy concerns.

1.1 Record and Replay

Record-replay approaches [64; 62; 69; 46; 53; 2; 35] capture and reproduce application bugs by recording them as they occur, then providing the recording to the application vendor to deterministically replay the bug at a later time. By directly recording the application and capturing the bug as it occurs, the burden of repeated testing to reproduce the bug is removed. When combined with a checkpoint-restart mechanism, record-replay techniques could be used to capture an interval of an application's execution and replay the interval at a later time.

Despite the potential for simplifying bug reproduction and debugging, the fundamental limitation of previous record-replay approaches is that they require the record and replay environments to be roughly identical. In particular, all previous approaches require at minimum the availability of all original code executed as a part of the recording, including not just the buggy application binary, but also any other software executed, such as other applications, libraries, utilities, and the operating system. If the application were to map a new shared library, for example, the shared library is expected to be available at the destination. The original code is required to generate the instructions that will be executed on replay.

This is problematic in practice for several reasons. First, customers often disallow application vendors direct access to their production environments, so replaying a bug in the original production environment is simply not possible. They are unwilling to make their actual production environment available to vendors for debugging purposes given that keeping it up and running in production is crucial for business. Production compute environments represent key IT assets of an enterprise which are expected to be run at full utilization. Even small downtime typically requires extensive coordination and reprovisioning. Second, customers are often unwilling to even make replicas available since they may contain custom proprietary software that they do not want to provide in their entirety, or applications from other vendors which they are not allowed to provide to a competitor. An application bug can arise from interactions with a particular mix of application versions, and those other applications may not be available at the time of replay outside of the production environment. Third, even if customers provide detailed information to allow vendors to create replicas, it is quite difficult for them to get all the versions and configurations of all software right to replay a bug that occurred in a complex production environment which depends on complex interactions in the environment. Fourth, even if an exact replica of a production environment could be created for debugging purposes, its creation may be prohibitively expensive in terms of both hardware and software requirements for complex production environments. Finally, bugs can be data dependent and all necessary data is typically not available outside of the original production environment. While it may be possible to record every single instruction executed, along with all data arguments, so that they can be replayed without need for the production environment, such a recording would be prohibitively expensive to do, impose excessive storage requirements, and result in unacceptable recording overhead in production.

1.2 Record and vPlay

This dissertation presents vPlay, a software problem determination system which captures software bugs as they occur in the field into small and self-contained recordings and allows them to be deterministically reproduced across heterogeneous operating system environments in the lab. vPlay achieves this functionality using a novel recording mechanism which provides data level independence from the source environment and a new type of virtualization abstraction, vPlay Container, which enables applications built for one operating system to be *natively* replayed on another. While vPlay recording ensures that all information necessary to reproduce an application bug is available, vPlay Container provides the virtual environment which replays that information to the application by insulating its execution from the underlying environment.

1.2.1 vPlay Recording for Data Independence

vPlay recording mechanism adopts a state machine model of application to capture each piece of state accessed by the application such that its execution can be replayed completely independently of the source environment. Starting from an initial CPU state, vPlay treats all other state that crosses the application boundary, including its memory and the very instructions it executes, as external inputs and captures it on initial access. This model creates a concise but complete recording that can replay the application for the specified time interval.

vPlay recording provides two guarantees by design. First, any state not directly accessed by the application is not included in the recording. Since vPlay only captures state actually accessed by the application, any extraneous state such as unaccessed parts of the application's address space or its binaries are not included, leading to a small per-bug recording. Other automated error reporting tools may collect various assorted pieces of information, but vPlay's data collection is directed by the application's execution itself. While conventional error reporting tools may blindly gather information, vPlay only collects information which is known to be relevant.

Second, all state necessary to replay a specified interval of execution is captured. As vPlay monitors every interface through which the application could access external state, any data required by the application during its deterministic re-execution is guaranteed to be available. The completeness of vPlay's recording allows it to replay the application independent of the target environment by providing necessary state from the self-contained recording. For example, a bug in a Linux application can be natively replayed on Windows. At the target lab, there is no need to install or configure the original application, support libraries, other applications, or the operating system to reproduce the failure. Portions of the application environment, including bits of application and library code necessary to reproduce the failure, are automatically isolated and recorded.

The key feature of vPlay recording is that it allows replay to occur without replicating the original production environment in which the application is recorded. Exclusively relying on a lightweight per-bug recording, vPlay deterministically reproduces the captured bug on a different operating system, without access to any originally executed binaries or support data. The dependencies on the application binaries is addressed by directly capturing the specific code pages within the executable files accessed by the application during the faulty run. Alternatively, it may be possible to install the same set of application binaries and libraries at the target site. However, installing an application usually involves also installing a series of other packages, which can be a tedious task. Furthermore, there is no clear and systematic way to determine the set of packages and their configurations necessary to reproduce the failure. Directly recording the actual binary pages provides the most definitive way to ensure that the same instructions and data are presented to the application during replay.

vPlay introduces *partial checkpointing*, a simple, novel technique to capture the partial intermediate state of the application required to partially reconstruct the application for replay. Based on the premise of short error propagation distances [48; 72], vPlay cap-

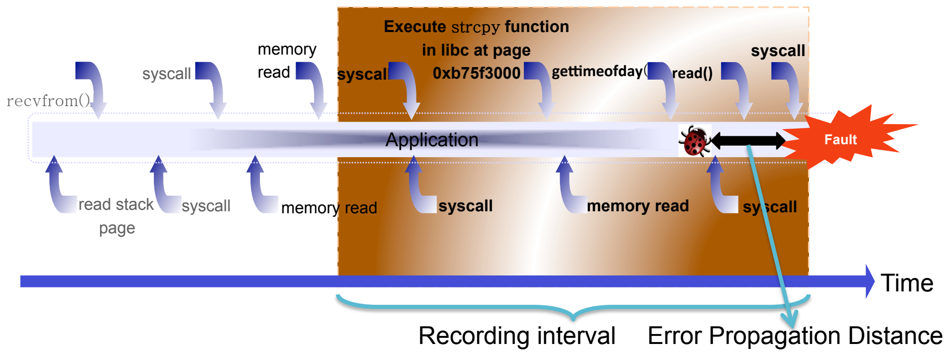


Figure 1.3: Partial checkpointing captures data consumed by the application within the last recording interval

tures the application state necessary for its execution during the last few seconds prior to its failure (Figure 1.3.) Instead of the traditional approach of taking a full application checkpoint representing the application's cumulative state until that point, partial checkpointing completely ignores the application's previous execution and focuses on state accessed by the application within the interval of interest. All application state, including the instructions executed within its binary and other libraries, required to reconstruct the application for replay is captured. While a complete application checkpoint [51; 34; 22] can require high overhead and have an adverse impact on client's privacy, partial checkpointing selectively records discrete pieces of data accessed by the application during a brief time interval immediately preceding a failure.

The resulting *partial checkpoint* is saved in a format which is designed to be portable across different operating systems. In addition to the data directly consumed by the application, a partial checkpoint also contains metadata identifying the specificities of the source environment that allows the data to be correctly interpreted even on a different operating system. vPlay provides a *partial checkpoint loader* to load a partial checkpoint into memory. The partial checkpoint loader itself is designed to be portably implemented on any operating system. It partially reconstructs the application and directly launches it into a vPlay Container where the application resumes execution from an intermediate point prior to its failure and continues through the end of the specified interval.

vPlay combines partial checkpointing with a logging mechanism that serves two purposes. First, it ensures that all data accessed by the application during its execution in the interval is recorded. In particular, changes to the memory region geometry are tracked and any new memory state accessed by the application is logged by trapping accesses to the memory pages. Second, the logging mechanism captures the nondeterministic events encountered by the application so that the same execution trajectory is reproduced at replay. vPlay's logging mechanism provides techniques to address each source of application nondeterminism, including asynchronous signal events, data race conditions caused by concurrent accesses to shared memory and system calls, which may otherwise diverge the execution of the application.

1.2.2 vPlay Virtualization

Conceivably, a variety of issues arise when an application from one operating system is executed on another. It has been shown in the past that applications can be run natively across different application environments, kernels versions [58] and even hardware architectures [67] through various forms of virtualization and emulation. However, natively running an application binary from one operating system on another has been challenging. At the outset, the target operating system may not recognize the binary format. The memory model of the target operating system may prevent the application code to be loaded. The application may depend on specific system configuration data, libraries and their layout on the persistent storage.

Existing virtualization approaches are unsuitable for replaying application bugs across different operating system environments. Hypervisor virtualization [4; 5] allows applications of one operating system to be run on another through the medium of a guest operating system. However, since virtual machines need to encapsulate the guest operating system and its applications in their entirety, the state associated them is typically too large for easy sharing with the developers and may contain sensitive client data. Container virtualization [51; 59] isolates the application from the underlying environment by providing private resources and namespaces. However, they essentially run the virtualized application directly on the underlying operating system and cannot support applications from a different operating system, which may require services that are unavailable on the target or are incompatible with its semantics.

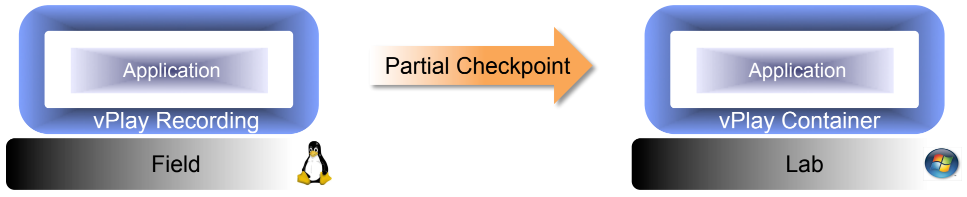


Figure 1.4: vPlay decouples the application from its environment

vPlay introduces vPlay Container, a thin virtual replay environment which can natively replay applications built for one operating system on another (Figure 1.4.) vPlay Container decouples the application in three key areas which enables it to run in a different operating system environment than the one it is built for. First, the application is decoupled from the memory address space and its content. In particular, the application is decoupled from its binaries by trapping accesses to the code pages and presenting the actual pages captured at the source, avoiding dependencies on the target and any version discrepancies. The ABI incompatibility is addressed through a portable design of the partial checkpoint and the partial checkpoint loader. Second, applications are decoupled from the processor MMU structures such as segment descriptor tables through a binary translation technique designed specifically for user application code, which traps and emulates the offending instructions during replay. Third, the application is decoupled from the operating system by abstracting the system call interface through emulation and replay. System call emulation decouples the application from the various operating system specific resources and dependencies. It also decouples individual processes of the application from each other by virtualizing inter-process interactions.

vPlay achieves heterogeneous record-replay functionality while meeting four important goals. First, vPlay does not require source code modifications, relinking, or other assistance from the application. Record-replay is a complex and sensitive operation and requires intimate access to application's internal processing to record the state required for replay. Consequently, it is typically implemented as a loosely coupled library extension [64] of the application itself to monitor its execution. Sometimes record-replay tools require the programmer to directly participate in the process through language extensions [23] which assist in determining the pieces of state that need to be recorded. However, these methods fail to support the large body of commonly used proprietary and legacy applications.

Second, vPlay does not require specialized hardware modifications. It may be possible to bypass all software layers and provide a transparent solution by recording individual instructions at the processor level with additional hardware support. Several processor extensions [78; 48] propose mechanisms to record and replay applications at the instruction level granularity. Unfortunately, they are only tested in software simulations and few of them have ever been implemented in practice due to the high cost barrier. They also impose a high space and runtime overhead due to the fine-grain level of their recording.

Third, the state captured by vPlay is small. Rather than coarsely recording the application along with its operating system and all other potentially irrelevant processes running on the system, state recorded by vPlay is per-bug and small enough that it can be easily shared with the developers. Having to share large amounts of data or heavy virtual machine image files impacts ease of use. Even storing and accessing recordings corresponding to long durations of application execution presents a challenge. Sharing large amounts of data, much of which may not be useful for problem determination, further compounds privacy concerns.

Fourth, vPlay's runtime recording overhead is low. Bugs that occur in production, in software released after internal rounds of testing and quality control, are most difficult to reproduce and fix. Bugs which manifest in production may never occur again even after extensive internal testing. It is necessary to capture the bug directly in production. However, enabling record-replay function for production software requires a highly efficient design which does not consume excessive resources in production and impact the common case performance of the application. As a positive side effect, leaving vPlay's instrumentation enabled in production also side steps the probe effect problem. If a bug does not manifest due to the instrumentation, it is not an undesirable consequence for production software. However, if the instrumentation triggers a latent bug, vPlay would capture it and assist the developer to quickly fix it.

1.3 vPlay Usage Model and System Overview

vPlay is a tool for recording and replaying specified intervals of the execution of a group of processes and threads. We refer to a group of processes and threads being recorded or replayed as a *session*. A session can consist of multiple processes that make up an application or a set of applications, where each process may contain threads that share the address space of the process. Once vPlay is installed on the same machine as a production application, it continuously records its execution. When a fault occurs, vPlay outputs a set of partial checkpoints and logs taken before the fault. When recording multiple processes, partial checkpoints and logs are saved separately for each process, along with information identifying the process that had the failure.

vPlay divides the recording of an application into periodic, contiguous time intervals. For each interval, it records a partial checkpoint for each application process, and a log for each application thread that executes during that interval. A recording interval can be configured to be of any length. As the application executes, a series of partial checkpoints and logs are generated and the most recent set of checkpoints and logs are stored in a fixed size memory buffer. Storing a set of partial checkpoints and logs rather than just the most recent one ensures that a certain minimum amount of execution context is available when a failure occurs.

Partial checkpoints and logs are maintained in memory to avoid disk I/O and minimize runtime overhead. Older partial checkpoints and logs are discarded to make room for the new ones. Partial checkpoints and logs in memory can be written to disk at any time by stopping the current recording interval, causing the accumulated partial checkpoints and logs in memory to be written to disk. vPlay has built-in support for detecting explicit faults such as segmentation violation and divide by zero. In addition, vPlay provides an interface to integrate with external fault sensors. Characterizing application behavior as faulty or correct is difficult in general. Interfacing with external fault detection mechanisms would simplify the mechanism and enable it to correctly function across a broad range of application failures.

When a failure occurs, the recording can be made available to the developer in lieu of, or as an attachment to, a bug report. The bug can then be directly replayed on any hardware in developer's environment using vPlay. Although the failure may involve an interaction of multiple tiers of software, the developer does not need access to any of that software to reproduce the failure. This is important since an application developer may have access to only his application software, not other software required to reproduce the failure. Since vPlay captures architecture dependent binary instructions of the application as a part of its partial checkpoint, the target CPU where replay is performed is required to be the same type as the original CPU. Other hardware attributes, such as number of processors, amount of memory, are not required to be the same.

Using a vPlay recording, a developer does not need to replay an entire multi-process application or set of applications. The developer could just select the process where the fault occurs to simplify problem diagnosis, and vPlay would replay just that process. vPlay virtualizes the interactions of the selected process with other application processes to provide a consistent replay. If the selected process uses shared memory, vPlay also simultaneously replays other processes that share memory with the selected process so that the mutual shared memory interactions among the processes and threads can be reproduced for a deterministic replay. This enables replay to be done on modest hardware even if the original production application required extensive and complex hardware resources.

vPlay integrates with a standard debugger to closely monitor and analyze the execution of the application being replayed. Any inputs needed by the replay are provided from the recorded partial checkpoint and any outputs generated by replay are captured in an output file and made available to the user. If the application writes into a socket, for instance, the user would be able to examine the contents of the buffer passed to the `write` system call and also see how the content of the buffer is generated during the steps leading up to the system call. For root cause analysis, vPlay allows the programmer to set breakpoints at arbitrary functions or source lines, single step the instructions, watch the contents of various program variables at each step, and monitor the application's original recorded interactions with the operating system and other processes. Reverse debugging can also be done by resuming the application from an earlier partial checkpoint with a breakpoint set to a desired point of execution in the past.

A partial checkpoint file itself does not contain any symbol information, so the debugger retrieves it from a separately provided symbol file. Typically, application binaries are stripped of their symbol table and debugging sections before they are shipped to the user. However, the symbol and debugging information is preserved in respective formats [1] separately in a symbol file which would be accessible to the developers.

1.4 Contributions

This dissertation presents a novel software problem determination architecture, vPlay, which efficiently and transparently records software bugs as they occur in production applications and allows them to be reproduced across heterogeneous operating system environments, in the presence of nondeterminism. The architecture is based on the following novel contributions:

1. A novel recording mechanism based on the state machine model of the application, which provides data-level independence between the application and its source environment by concisely capturing the complete state required for its replay. In particular, it removes the dependencies on specific versions of application binaries by isolating and recording bits of application and library code necessary to reproduce the failure.

2. A new checkpointing paradigm, partial checkpointing, which captures the partial intermediate state of an application to reproduce its execution for a specified time interval. When applied to debugging, partial checkpointing allows capturing the application's execution during the last few moments of its failure. The resulting partial checkpoint, together with a log, is used to deterministically reproduce the faulty behavior at the developer site. Partial checkpointing minimizes the impact on client's privacy by recording only discrete pieces of data rather than a complete application checkpoint.

3. A portable representation of the partial checkpoint and a corresponding partial checkpoint loader, which together enable an interval of an application's execution to be replayed across different operating system environments.

4. A memory tracking algorithm which efficiently identifies and captures the changes

made to the application's memory region geometry by its constituent processes and threads. Two embodiments of the algorithm, one based on an extension to the operating system's page fault handler and another based on accessed and dirty bits provided by the processor, are presented.

5. Incremental partial checkpointing algorithm, a further optimization to partial checkpointing is discussed and illustrated. It reduces the size of a partial checkpoint by omitting the content which is already included in earlier partial checkpoints in the sequence.

6. An innovative user space instrumentation framework which enables partial checkpointing to be implemented in user space. The framework allows application events such as system calls to be securely and efficiently intercepted in user space based on a simple modification to the `ptrace` subsystem. The mechanism avoids excessive context switches, typical of `ptrace`, by allowing an application to trace its own events. While being a part of the application's address space, the instrumentation framework protects itself against wild stores from faulty application code.

7. A mechanism to record and reproduce race conditions originating in concurrent accesses to shared memory on multiprocessor systems, which makes novel use of existing hardware features.

8. A new form of virtualization and an associated abstraction called vPlay Container which enables applications built for one operating system to be natively replayed on a different operating system. vPlay Container addresses the ABI incompatibility between the application and the target operating system using a lightweight binary translation mechanism.

9. A debugging tool which integrates vPlay Container with a standard interactive debugger to provide functionality such as setting breakpoints at the source code level within a discrete interval of application's execution recorded in a partial checkpoint. The tool allows the programmer to analyze the steps the application has taken to reach the failure state.

10. A practical debugging-as-a-Service framework which applies vPlay mechanisms to the cloud computing model to alleviate the challenges presented by incipient cloud infrastructure to the legacy applications which are yet to adapt to the cloud. In addition, several other innovative usage scenarios such as such as recording important transactions for archival or compliance, efficiently generating fine-grain traces of applications running in production etc. are discussed.

1.5 Organization of this Dissertation

This dissertation is organized as follows. Chapter 2 describes the high level principles behind vPlay recording and the type of data it collects. Chapter 3 presents the partial checkpointing mechanism, associated algorithms and the details of their kernel and user level implementations. Chapter 4 describes the mechanisms used by vPlay to capture the state accessed by the application during a recording interval, and to address various sources of application nondeterminism. Chapter 5 describes the vPlay Container abstraction and the virtual replay mechanism itself. Chapter 6 provides an analysis of vPlay's performance along with investigative reports of its use in debugging real-life software bugs. Chapter 7 discusses related work. Chapter 8 presents various possible extensions and applications of the mechanisms introduced by vPlay. Finally, we present some conclusions and directions for future work in Chapter 9.

Chapter 2

vPlay Recording

The main goal of vPlay is to produce a concise and complete recording of an interval of an application's execution, so that the recording can be used to independently replay the application in a different environment. In order to decouple an application from its source environment, vPlay provides partial checkpointing (Chapter 3) and logging (Chapter 4) techniques to capture each piece of state accessed by the application within the specified interval. Partial checkpointing isolates the partial state of the application into a *partial checkpoint*, which is restored initially at replay, and logging captures a log of state accessed by the application during the interval, which is replayed back to the application at replay.

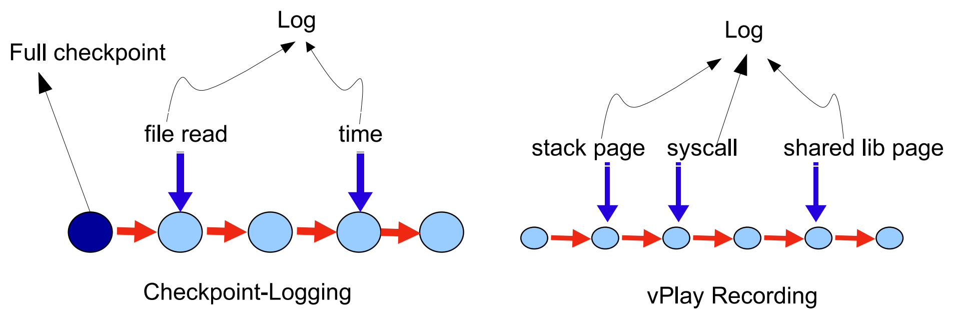


Figure 2.1: Checkpoint-Logging Approaches vs vPlay Recording

An application consumes a variety of data as it executes. For example, the application may query information from the operating system, obtain the value of a configuration option from a file, or receive a response from other application tiers or co-deployed services over a network socket etc. These are clearly different types of data that the application consumes during its execution. Additionally, the application may read from its own program stack, access a global variable within its data segment or execute a particular subroutine in a memory mapped shared library. In order to completely decouple the application from its environment, it is necessary to consider these pieces of data, including the very instructions it executes, to be a form of input to the application.

To exhaustively collect the state accessed by an application, vPlay uses a recording approach based on the state machine model. The application is treated as a state machine which contains certain internal state, and consumes an external input to transition into another state. The model naturally creates an application boundary that separates the state *within* the application from the state *outside* of it, and allows for isolating and recording the "inputs" that cross the application boundary during a specified time interval and independently replaying them back to the application at a different location. In conventional checkpoint-logging schemes (Figure 2.1), the application boundary is defined to include the state of application's resources such as registers, memory, open files etc. In contrast, vPlay shrinks the application boundary to only contain the processor state, with all other state treated as external input. vPlay interposes on key operating system kernel entry points to intercept all interactions of processes and threads with their environment. All state required for replay, including application's own memory, is captured on first access. In addition to recording the system call results and other events required for deterministic replay, vPlay monitors accesses and changes to the address space pages and captures relevant information to create a self-contained recording of an application bug.

vPlay divides recording into periodic, contiguous time intervals called *recording intervals*. A recording interval can be configured to be of any length; it can be few tens of milliseconds or several seconds of execution time. Shorter intervals incur more runtime recording overhead, while longer intervals typically result in larger partial checkpoints. Each recording interval commences with a `start` operation and concludes with a `stop` operation. An external process enforces the recording intervals by issuing `stop` and `start` commands in succession to conclude the previous recording interval and begin a new one. When the application fails or exits, vPlay intercepts the exit event and automatically performs the `stop` operation. As the application executes, a series of partial checkpoints and logs are generated. Within each recording interval, partial checkpoints and logs for each application process and thread that executes in the interval are recorded. The recorded data contains the information necessary to replay the execution of the application during that interval.

The most recent set of partial checkpoints and logs produced during the execution are stored. Storing a set of partial checkpoints rather than just the most recent one ensures that a certain minimum amount of execution context is available when a failure occurs. If the application encounters a failure at the beginning of the recording interval, immediately after `start`, the most recent partial checkpoint may not contain sufficient context for root cause analysis. vPlay seamlessly splices the discrete series of consecutive partial checkpoints into a new partial checkpoint encompassing the total length of the original checkpoints. The user can resume the application from any partial checkpoint in the series and have it continue its execution through the intermediate checkpoints, finally reaching the point of failure. The user can progressively go, as further back as necessary, within the available set of checkpoints to reach the problem source. A particular partial checkpoint in the series marks a well defined point in the execution of the application from where the application can be resumed. An arbitrary point within a recording interval can be reached by rolling forward from the latest checkpoint prior to the desired execution point.

Partial checkpoints and logs generated during the application's execution are maintained in memory to avoid disk IO. Since the goal of vPlay is to capture software failures as they occur in production, runtime performance is an important consideration. The partial checkpoints are stored in a fixed size memory buffer. The number of partial checkpoints, and hence the length of execution history available at any point, depends on the size of the memory buffer dedicated for this purpose. Older partial checkpoints are discarded to make room for the new ones.

Recording can be terminated at any time at the request of the user by calling `stop` to preempt the current recording interval. Partial checkpoints accumulated until that point are written to disk. If the application encounters a failure which results in an explicit exception, such as segmentation violation or divide by zero, vPlay intercepts the exception and writes the recording to the disk. vPlay also provides an interface to enable an external fault detection system to trigger the checkpoint upon detecting a failure. Writing partial checkpoints to disk bears some similarity to the core dumps generated by the operating system. However, a core dump only contains the state of the application at the point of failure, whereas a recording consists of the state of the application a few moments before the failure and the state necessary to deterministically lead its execution to the point of failure.

2.1 State Composition

vPlay applies *copy-on-read* technique at the point of access, to isolate relevant pieces of application state. A shared library page, for instance, is included in the partial checkpoint when the application calls a function located in that page. Similarly, the signal handler state is included when the respective signal is delivered to a thread. A signal handler becomes a part of the application state when it is installed, but it is only needed and used when the signal is actually delivered. Rather than capturing the state of all signal handlers installed by the application in advance, only partial signal handler state corresponding to the signals which are actually delivered within the recording interval is captured. Table 2.1 shows a listing of various types of data captured by vPlay.

For certain resources, vPlay performs full checkpoint rather than capturing partial state. Some pieces of application state such as the runtime register context and the state of the processor MMU resources, are typically small and updated too frequently to efficiently track their state in runtime. An application may never access the contents of a particular register during a recording interval and may not be necessary to be captured. However, the additional complexity and overhead required to track such state is not worth the marginal space savings it may produce.

Much of the internal state of the application resources is ignored by vPlay. Since vPlay focusses on interposing the application boundary, only state internal to the application needs to be checkpointed. All other state, including the intermediate state of various system resources used by the application such as open file descriptors, is not captured. vPlay also does not capture any application outputs produced at recording time. Due to deterministic re-execution, identical outputs will be automatically generated during replay, as determined by the program logic.

In addition to the data directly consumed by the application during replay, vPlay also records certain metadata indicating the provenance of each piece of state included in the recording. The application data itself is required for its execution during replay. The additional metadata helps present the data within the right context for the user to analyze and debug the behavior. For instance, a malformed service request may have caused a server application to fail. In addition to the contents of the offending request, the peer name of the sender would help identify the problem source. Similarly, an application may have failed due to an unexpected interface implemented by an incompatible library version. The failure may be correctly reproduced by simply presenting the same offending subroutine within the incompatible library. However, if the source of the code executed as a part of replay is unknown, it would be of little value for debugging. In addition to the code pages, the .so name and the path of the shared library would point to the root cause of the problem.

Resource	**Checkpoint type**	**Access point**	**Provenance**
Per-session state			
Shared memory	Partial	Page fault	Region name
Per-process state			
File / storage data	Partial	`read()`	File path
Memory mapped file data	Partial	Page fault	File path
Binaries, shared libraries	Partial	Page fault	.so name
Per-thread state			
Anonymous regions (stack, heap etc)	Partial	Page fault	–
CPU/FPU state	Full	`start`	–
Signals	Partial	Signal delivery	Sender
Descriptor table entries	Full	`start`	File path
Thread local state (`clear_tid_ptr` etc)	Full	Respective system call	Thread identifier
External connection requests	Partial	`accept()`	peer address / port
System state (TCP, semaphores, pipes etc)	Not checkpointed	–	–
Other OS state (pid, `uname` etc)	Partial	Respective system call	Respective system call

Table 2.1: Composition of state recorded by vPlay

Chapter 3

Partial Checkpointing

The traditional approach for recording an interval of an application's execution consists of checkpointing the initial state of the application at the beginning of the interval, followed by logging events that guide replay. The initial checkpoint consists of the state of application's resources representing its cumulative execution until the beginning of the interval, and the log consists of events such as system calls, thread interactions and data inputs required to guide replay. Such an approach, however, may include data which is not relevant or miss data which is relevant. For instance, the checkpoint may contain pages in memory address space or state of resources such as open files which may not be used at replay, or in some cases, the application may map a large data file and access a few pages within it, but that data would not be included in the initial checkpoint, even though it is necessary for replay.

vPlay adopts a different approach called partial checkpointing, to capture minimal but complete state required for replay. Instead of taking a cumulative checkpoint of process memory address space at the beginning, only specific memory pages actually used by the application in an interval, and hence necessary for replay are captured. Any previous execution, and state accumulated as a result, is ignored. The partial application state, including a subset of pages mapped at the beginning of a specified recording interval which are accessed by the application within that interval, constitutes a partial checkpoint. A partial checkpoint is used to partially reconstruct the application at the developer site and allow it to deterministically replay from an intermediate point prior to the failure to the point of failure.

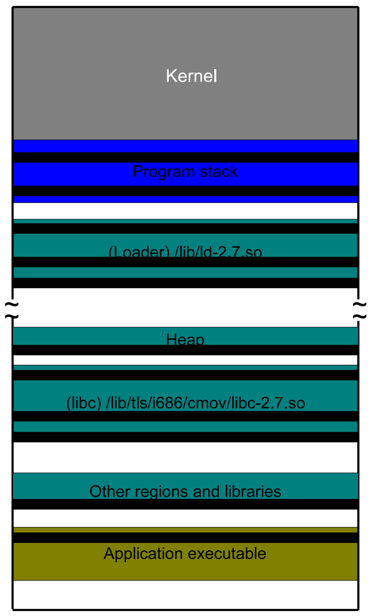


Figure 3.1: Address space layout showing partially checkpointed memory

A partial checkpoint only consists of sparse set of pages from the total mapped address space of an application. Figure 3.1 illustrates the 4 GB address space of a typical Linux/x86 process. The top 1 GB of address space is dedicated for the kernel use and its contents are never included in a partial checkpoint. Application stack resides near the top of the 3 GB user address space, followed by system libraries and various other libraries used by the application. The main application binary is loaded near the bottom of the address space. The total virtual memory usage of even simple applications is usually quite large with all the shared libraries mapped within the process address space. However, as depicted by the black strips of captured pages in the figure, a partial checkpoint only represents a fraction of the total process address space.

3.1 Properties

A partial checkpoint has four key characteristics. First, the state captured is completely decoupled from the underlying application binaries and the operating system. Partial checkpointing creates a self-contained recording which can be used to reproduce the execution in a completely different environment where none of the state from the source environment is available. Traditional checkpoint-logging approaches record the current state of the application's address at the time of the checkpoint. Any other shared libraries or file data mapped into the application address space after the checkpoint is taken, are usually required to be available at the destination as well. Alternatively, those additional libraries and files are packaged along with the checkpoint, increasing the size of the recorded state.

Second, it is defined only for a specific interval of an application's execution and contains only the portion of state accessed by the application in that interval. The space needed to store a partial checkpoint can be small since it is used only for recording the execution for a brief interval of time. Even though an application itself may be large in its memory footprint and processing large quantities of data, it only accesses a fraction of itself during a brief interval of time.

Third, it is only useful for deterministically replaying the specific time interval, not for running the application normally. When the application is replayed, it does not perform any useful work, except that its execution is available to be analyzed using tools such as debuggers and profilers. Once the application reaches the end of the recording, it would have to be stopped or terminated because the subroutines that the application may invoke after that point may not exist, or any further requests it may make cannot be satisfied. Finally, it is captured over a specified time interval, not at a single point in time. A particular piece of state is included in the partial checkpoint when it is first accessed within the interval.

3.1.1 Related Techniques

Regular checkpointing. Partial checkpointing is a new concept which is substantially different from standard checkpointing techniques [55; 51]. In particular, the system state of the application maintained internally by the operating system, such as the state of file descriptors and the state of various operating system resources, is not included in a partial checkpoint. It allows partial checkpointing to be implemented without significant kernel changes and enables a partial checkpoint to be resumed even on a different operating system. Regular checkpointing allows normal execution to be resumed for an arbitrary amount of time. Because the state needed by an application in its future execution can be arbitrarily large, regular checkpoint implementations typically impose dependencies on the underlying system to reduce storage requirements, such as requiring that files in persistent storage be available to the resumed application. In contrast, partial checkpointing does not impose such a requirement because, the specific portions of data on disk, including portions of application binaries themselves needed by the application during replay, are included in the partial checkpoint. Partial checkpoints are also small since they do not need to support normal execution; only deterministic replay over a fixed time interval.

Incremental checkpointing. Partial checkpointing is substantially different from incremental checkpointing [57]. Incremental checkpointing assumes the existence of an earlier full checkpoint, and saves only the execution state that has changed since the last checkpoint. To resume execution from an incremental checkpoint, the state from a full checkpoint must be restored, as well as the state from the subsequent incremental checkpoints. Partial checkpointing differs in at least three ways. First, partial checkpointing does not require saving or restoring any full checkpoint. All state necessary to use a partial checkpoint is completely contained within the partial checkpoint. Second, a partial checkpoint is completed after the recording interval to enable deterministic replay over only the previous interval. In contrast, an incremental checkpoint occurs after a time interval to enable normal execution to be resumed after that time interval going forward. Third, a partial checkpoint contains state that has been read during a time interval, while an incremental checkpoint contains state that has been modified.

Virtual machine snapshots. Checkpointing mechanisms based on virtual machine snapshots [32] typically rely on the availability of complete virtual machine image, including all software code, its entire file system and additional file snapshots, to resume the execution. In contrast, partial checkpoints are not only small and lightweight, but designed to be used without access to any of the code or data originally used during the production execution. Decoupling provided by partial checkpointing is also different from that of virtual machines [10; 15], which may decouple replay of an application in a guest operating system from dependencies on the hosting environment, but still require during replay the complete virtual machine image with all of the installed binaries used at the time of recording.

3.2 Partial Checkpointing Mechanism

Recording memory pages at the operating system level presents new challenges. Operating system memory model is more complex than, for example, the hardware level memory model, where the application reduces to a steady stream of instructions after having gone through appropriate processing at the higher layers. The complex memory model and the dynamic nature of application's execution with memory regions being mapped, unmapped, remapped and shared across multiple threads and processes makes efficiently capturing these events difficult.

We will use Linux and x86 semantics to describe how partial checkpointing is done in further detail. A partial checkpoint broadly consists of session state accessed by processes and threads in the session, per-process state, and per-thread state. Per-session state consists of global shared memory objects accessed during the interval and not tied to any process, such as shared mapped files and System V shared memory. Per-process state consists of the initial set of memory pages needed to enable replay, and mappings that correspond to the global shared memory objects used by that process. Per-thread state consists of CPU, FPU, and MMU state.

To start recording a partial checkpoint, vPlay forces all threads in the session to reach a synchronization barrier. The barrier is required to produce a globally consistent partial checkpoint across all threads. The last thread to reach the barrier records the CPU, FPU and MMU state of each thread, including the processor register state and the user created entries in the global and local descriptor tables. The processor context marks the initial point of execution during replay. It consists of the state of the CPU and FPU registers, and the per-thread state of the processor MMU such as descriptor entries in the segment descriptor tables. The state of segment registers, not directly accessible by user applications such as `cs`, `ds`, is excluded from the register state. These registers typically represent the memory segments covering the entire available address space on Linux and Windows and do not have to be changed on the target system. Appropriate values already setup by the target system are used. A status flag indicating that the session is in recording mode is set and all threads waiting at the barrier are woken up.

For both per-process and per-session memory state, only pages that were read during

Algorithm 1: Partial checkpointing mechanism implemented within the page fault handler

```
1  if partial flag in the PTE is set then
2      if page is shared then
3          add (page address, page content) to the corresponding
           shared_memory_object;
4          add (region's start address, corresponding shared_memory_object) to the
           process shared_maps;
5      else
6          if page is mapped within current recording interval then
7              add page and page content to the list of saved pages in the respective
               system call event_record;
8          else
9              add page to the initial_page_set;
10         end
11     end
12 end
```

a recording interval need to be saved in a partial checkpoint. If a process only writes to a page, but does not read from it, the partial checkpoint does not have to provide such a page during replay. However, page table status flags provided by most processors are not sufficient to determine if a written page has also been read. We conservatively include all pages accessed during the interval in the partial checkpoint even though the application may not have read from some of them. This approximation works well in most cases as most pages that are written by an application are also read.

To save per-process memory state in a partial checkpoint, vPlay must determine the memory pages that are read by the threads associated with that process during the recording interval. Similarly, vPlay must also account for per-session state corresponding to memory objects that are shared across multiple processes and not necessarily associated with any individual process. To save per-session state in a partial checkpoint, vPlay must determine the memory pages of global shared memory objects that are read by the threads during the recording interval. Algorithm 1 illustrates the partial checkpointing mechanism and Table 3.1 describes the data structures involved.

vPlay uses two types of objects to store the contents of accessed pages during the recording interval. A per-process `initial_page_set` is allocated for memory regions private to a process. Each record in the set contains a page address and content. A per-session `shared_memory_object` is allocated for each shared memory region accessed within a recording interval and contains the subset of pages accessed by any process or thread in the session within that recording interval. Each record in the set contains the offset of the page within the region and its content as of the first access to that page. The pages in the `shared_memory_object` may be mapped as a whole at different addresses by different processes.

To track which pages are accessed, vPlay utilizes the present-bit available in the page table entry. vPlay cooperatively shares its use with the kernel by keeping track of kernel use of and changes to the bit by using one of the unused bits available in the page table entry as a *partial flag*. At the beginning of the recording interval, vPlay clears the present-bit for each page in the process address space that is present, and uses the partial flag to store the original value of the present-bit. When a memory region is newly mapped, the present-bits in the page table entries of the corresponding pages will be initially cleared indicating that the physical pages have not been allocated yet. When a thread accesses a page which does not have its present-bit set, a page fault is generated. As a part of the page fault handler, vPlay checks the partial flag to see if it is set. If it is set, the page was originally present and needs to be recorded. In architectures, such as ARM, which do not provide a page-present bit, the emulated version of the bit maintained by the Linux kernel can be used in place of the hardware bit for the purposes of page tracking.

If the page corresponds to a shared memory region, vPlay adds a record containing the offset of the page within the shared memory region and the page content to the `shared_memory_object` that represents the shared memory region. In addition, vPlay also updates a `shared_maps` set, which is a per-process set of shared memory regions representing the mapped instances of the `shared_memory_object`s for that process. Each record of the `shared_maps` set contains the starting address at which the shared memory region is mapped within the process address space and a pointer to the `shared_memory_object` representing the region. Otherwise, vPlay copies the page address and contents to the process's `initial_page_set`.

Each accessed page is copied just once when it is first accessed during the interval. Memory shared among threads associated with a process is automatically taken care of as a part of this simple mechanism. If a process is created via `fork` during the recording interval, its initially mapped pages at the time of creation which are accessed during the recording interval are also included in the partial checkpoint. This is done by performing the same operations to the process at creation time as were done to other processes already created at the beginning of the recording interval, namely clearing the present-bit for each page in the process address space that is present, and using the partial flag to store the original value of the present-bit. Note that for pages not corresponding to a shared memory region, vPlay only includes pages in the partial checkpoint that are already mapped at the beginning of the recording interval or at process creation.

Changes in Memory Region Geometry. The threads of an application may map, remap or unmap memory regions within a recording interval. vPlay must capture sufficient state to reproduce these events at replay. vPlay keeps track of the system calls made by each thread in a per-thread queue of `event_record` structures. In addition, vPlay keeps track of the system calls that map memory in the current recording interval in a per-process stack called `recent_maps`, including a reference to the respective system call `event_record`. When a page is first accessed that was mapped during the recording interval, a page fault occurs and vPlay starts at the top of the `recent_maps` stack and scans it to find the most recently mapped memory region that corresponds to the accessed page, which is the current mapping being used by the thread. The page is then added to the respective system call `event_record` or to the respective `shared_memory_object` depending on whether the page belongs to a shared memory region. In addition, if the page happens to be a global shared page, a record containing a pointer to its `shared_memory_object`, and the starting address where the shared memory region is mapped in the process address space is added to the `event_record` of the system call event that mapped the shared memory region. If the page was not mapped within the current recording interval, the record is added to the `shared_maps` set of the process. When a failure is detected and a partial checkpoint is emitted, the pages associated with the system call are saved along with the `event_record`.

Data structure	**Function**
`event_record`	Entry describing system call state in per-thread system call queue
`recent_maps`	Stack of system call `event_record`s that map memory regions within current recording interval
`initial_page_set`	Per-process set of pages initially restored at replay
`shared_memory_object`	Set of (page offset, page content) records describing a sparse shared memory region
`shared_maps`	Set of (page offset, `shared_memory_object`) records indicating shared memory regions mapped within a process

Table 3.1: Partial checkpointing data structures

3.2.1 Incremental Partial Checkpointing

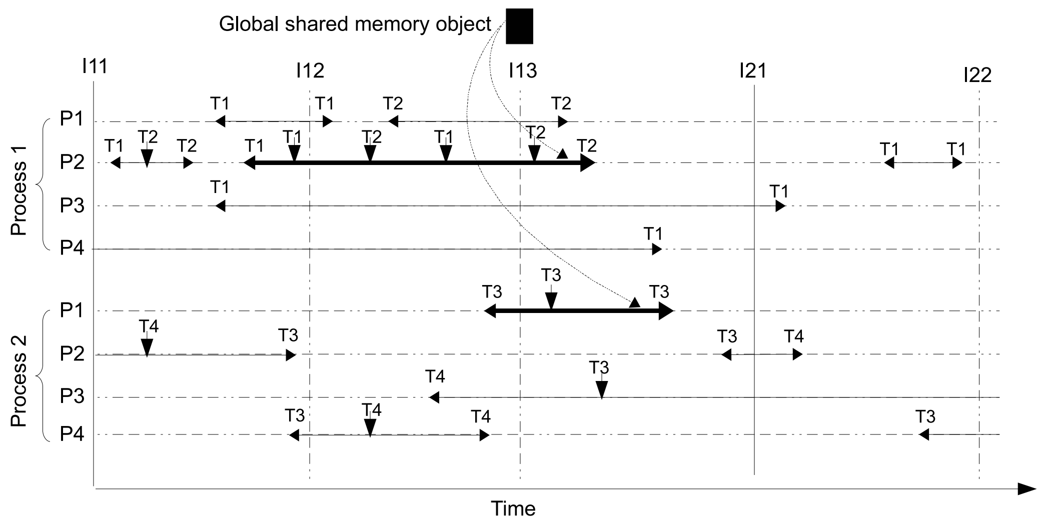
To reduce copying overhead, an incremental partial checkpointing mechanism can be used. Pages already copied as a part of previous partial checkpoints that are still stored in memory do not need to be copied again for the current partial checkpoint if the contents remain the same. This optimization would be most useful for code pages which are generally only read and never modified. This is done by creating regular partial checkpoints periodically but less often, and using incremental partial checkpoints in between regular partial checkpoints. An incremental partial checkpoint can only refer to the previous regular partial checkpoint and subsequent incremental partial checkpoints before the current incremental partial checkpoint. For simplicity, when an older partial checkpoint is removed from memory, all associated incremental partial checkpoints are also removed.

To create an incremental partial checkpoint, vPlay creates a per-process `incremental_page_set` for each process and a per-session `incremental_shared_object` for each shared memory object to track which pages are stored as a part of which regular and incremental partial checkpoints. They are maintained starting with a regular partial checkpoint through its successive incremental partial checkpoints. vPlay uses an extra unused bit in the page table entry as a dirty flag, clears it when the page is copied as a part of a partial checkpoint, and marks the page copy-on-write. Using the copy-on-write mechanism, any modification to the page will be detected and the dirty flag will be set. The per-process `incremental_page_set` mirrors the process address space and associates a pointer to the latest saved page content of the respective page address, stored as a part of the `initial_page_set` or the `event_record` of the memory map system call, as the case may be. Similarly, the `incremental_shared_object` associates the latest saved contents of a page with the respective page address associated with the respective shared memory region.

The next time the page is first accessed in a different interval for recording a partial checkpoint, if the dirty flag is clear and the page does not correspond to a shared memory object, the page address is copied to the `initial_page_set` of the current partial checkpoint, but the contents are not copied. Instead, a reference to the page content stored in the `incremental_page_set` is associated with the page address in the `initial_page_set`, avoiding the cost of copying the page contents. Similarly, if the dirty flag is clear and the page corresponds to a shared memory object, a record containing the offset of the page within the shared memory region, and a reference to the page content stored in the `incremental_shared_object`, is added, again avoiding the cost of copying the page contents. If the page was written to and the dirty flag is set, the page address and content are added to the appropriate system call `event_record` in the current recording interval, and the entry for the page address in `incremental_page_set` is updated to point to the new page data. While incremental partial checkpointing can reduce storage requirements and copying overhead, experimental results in Section 6 indicate that the additional complexity required is not needed as the storage requirements and copy overhead of regular partial checkpoints is modest.

3.2.2 Partial Checkpointing Examples

We illustrate the above mechanism using a hypothetical application with two processes each with an address space spanning four pages (P1-4). Threads T1 and T2 belong to process 1



Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Event	I11	M	R	R	W	U	SM	R	I12	R	W	I13	W	R
Thread #		1	2	1	2	2	1	1		2	1		1	2
Copied flag	0	0	1	1	1		0	1	0	1	1	0	0	1
Dirty flag	0	0	0	0	1		0	0	0	0	1	1	1	0
Page data pointer	Nil	Nil	D(2)	D(2)	D(2)		Nil	D(7)	D(7)	D(7)	D(7)	D(7)	D(7)	D(13)

I##: Checkpoint interval	D(t): Page data as of time t	M: Regular map	SU: Shared memory unmap
SM: Shared memory map	R: First read by a thread	U: Regular unmap	W: First write by a thread

Figure 3.2: Partial checkpointing illustration

and threads T3 and T4 belong to process 2. An interval of execution of the two processes within respective address spaces is represented in Figure 3.2. Its execution is divided into recording intervals, with solid vertical lines representing regular partial checkpoint intervals and dotted vertical lines representing incremental partial checkpoint intervals. Intervals I11, I12 and I13 belong to I1 regular partial checkpoint. Horizontal arrows represent the time interval during which the region of address space page has an associated mapping. The threads which mapped and unmapped each region are also indicated on the left and right sides of each arrow respectively. Absence of a horizontal arrow at a particular time indicates that the address space region is not mapped at that time. First read or write access to a mapping within an interval is indicated by a small downward arrow, with the thread number making that access indicated on top. Shared memory mappings are highlighted as thick horizontal lines.

3.2.2.1 Regular Partial Checkpointing

We describe the processing performed to take a regular partial checkpoint of process 2 for the recording interval I1. At the beginning of the interval, page-present bits for all four pages are reset and only page address P2 has an active mapping. Thread T4 makes the first read access in the interval to page P2. Since the page-present bit is reset, a page fault is generated. Algorithm 1 implemented as a part of the page fault handler checks if the page belongs to a shared memory mapping. Since it is not a shared memory mapping, it checks if the page was mapped within the current recording interval by consulting `recent_maps`. The page was indeed mapped prior to the current recording interval and has to be mapped at the beginning of replay. So the page is added to the `initial_page_set` of the process. The region is eventually unmapped by thread T3, which maps a new region at page address P4. Mapping a new region causes respective system call to be added to T3's log queue as an `event_record` and to the `recent_maps` stack of the process. When thread T4 accesses page P4, Algorithm 1 is invoked once again. Since the page does not belong to a shared memory region and the region was mapped within the current recording interval earlier by thread T3, the page is added to the list of saved pages in the respective system call's `event_record`.

Next read access occurs to a shared memory region mapped by thread T3 at page address P1. As a part of the page fault triggered when thread T3 accesses the shared page, the page offset within the region and its content are added to a global shared memory object that represents the shared memory region. In addition, mapping address of the region within the process address space and a pointer to the global shared memory object are added to the `shared_maps` set of the process.

3.2.2.2 Incremental Partial Checkpointing

The table in Figure 3.2 shows the processing performed to take an incremental partial checkpoint during the intervals I11, I12 and I13 for page P2 of process 1. At time 0, copied and dirty flags are cleared and the page data pointer is set to 0. At time 1, thread T1 maps a regular memory region and later at time 2, thread T2 accesses the region for the first time causing a page fault to occur. Since the copied flag is clear, vPlay needs to capture the page data, and since dirty flag and the page data pointer are both 0, the content of the page is saved in the page set associated with the system call that mapped this page earlier at time 1. Later when thread T1 accesses the same page for the first time, it sees that the copied flag is set and hence does not save the page. Thread T2 writes to the page at time 4 causing the dirty flag to be set and eventually unmaps it at time 5.

Thread T1 then maps a new shared memory region at time 6, reinitializing the copied and dirty flags and page data pointer to 0. First access to the page is performed by thread T1 at time 7. Since the page now belongs to a shared memory object, the page data is saved in the corresponding `shared_memory_object` and the starting address of the region and the `shared_ memory_object` identifier are added to the `shared_maps` set of the process.

The first incremental checkpoint interval is closed at time 8, causing all page ownerships to be preempted, all present and copied flags to be cleared and new shared maps and shared memory object sets to be created. Thread T2 accesses the page for the first time at time 9 and since the copied flag indicates that the page was not yet included in the current interval, it checks the dirty flag to determine whether the page data was already saved in a previous interval. Since its a shared page, the page data needs to be added to the respective `shared_memory_object` in the shared memory object set of the current recording interval. The data need not be copied because the dirty flag is 0. Instead the value of the current page data pointer is added. Appropriate record is also added to the shared maps set of the process for the current recording interval. Thread 1 eventually dirties the page at time 10 and the second incremental checkpoint interval ends while the shared memory region is still mapped. The copied flag is cleared at time 11 as a part of the initialization for the third incremental checkpoint interval but the value of dirty flag carries forward from the previous interval. When thread T2 reads the page at time 13, it adds the current page data to the new shared memory object of the third interval.

3.3 Partial Checkpointing in User Space

Partial checkpointing is a novel and valuable concept, and the ability to implement partial checkpointing in user space enables several additional advantages. A key use-case of partial checkpointing is to reproduce application execution across heterogeneous platforms. Executing applications across different platforms requires that the mechanism used for the purpose itself be portable. A user implementation would decouple the mechanism from the services of the host, and make it portable by minimizing the dependence on the kernel. Further, a user space design makes the mechanism simple and robust by remaining outside the sensitive, high-incidence control paths of paging and virtual memory within the kernel. However, without kernel support, a user space mechanism would be limited to applications that do not use shared memory, and frequent context switching required to trap events at the user level may impact the performance of the solution. We quantify the performance of the user implementation of partial checkpointing in Chapter 6.

An important design feature of partial checkpointing is that it does not require recording internal kernel state of the application which is not directly available to the user, and hence it is amenable for a user space implementation. By segregating the user space counter parts of the application and only recording its input-output interface, partial checkpointing remains untainted by platform-specific internal state. Regular checkpointing methods, on the other hand, typically require extensive kernel support or are limited in the scope of applications they support.

A user space partial checkpointing mechanism must provide two functions. First, a mechanism to transparently intercept system calls made by the applications (Section 3.3.1). Second, an efficient interface to determine the memory pages accessed by the application in a given interval and their contents (Section 3.3.2). The functions are implemented based on two simple and generic kernel extensions. Rest of this section presents the techniques which allow partial checkpointing to be implemented in user space based on these kernel extensions. The techniques are described within the context of an alternative user space implementation of partial checkpointing called vPlay-*user*, which leverages existing kernel and hardware features and provides a new instrumentation technique to enable a user space implementation. The mechanisms implemented by vPlay-user are conceptually similar to the kernel implementation described earlier. We highlight the pieces which are different.

3.3.1 Instrumentation Framework

vPlay-user uses an instrumentation architecture based on a small change to ptrace functionality which allows a process to set it as a ptrace parent to itself. As per normal ptrace semantics, a process attempting to attach to itself as a debugger results in an error. It may be possible to attach to the application through an external process. However, ptrace is originally designed for debugging and its runtime performance is unacceptable for common-case use. The simple extension provided by vPlay-user allows a process to be notified of events generated by itself in addition to any external process, such as a debugger, which may have registered to receive those notifications. In the conventional debugging paradigm, a separate debugger process controls the execution of the debugged process. The kernel notifies the debugger through a `SIGSYS` signal whenever the debugged application encounters events such as system calls, receipt of a signal, process completion etc. The debugger is allowed to take necessary actions before the event is processed. In such a model, each application event generates several context switches between the debugger and the debugged application resulting in high overhead. The ptrace extension implemented by vPlay-user avoids this context switching overhead by allowing the application itself to be the debugger. The challenge, however, is to transparently embed code into the process address space to handle the debugging events posted by the kernel. The rest of this section describes the architecture that permits this operation.

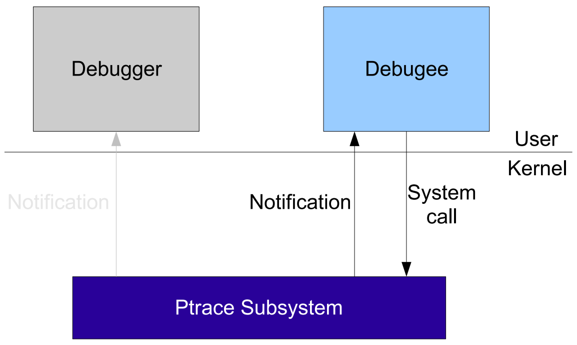


Figure 3.3: User space instrumentation via vPlay-user agent

The application is started by an initial process called, vPlay-*user agent*. vPlay-user agent is implemented as a self-contained statically linked application program with its load address chosen to be in an address range not commonly used by the applications. On Linux/x86, we have chosen the address range, `0x08000000 - 0x08031000`, to load the vPlay-user agent. Common Linux/x86 applications don't use addresses below `0x08048000`. As a part its initialization, the agent maps a region of memory with `MAP_SHARED` attribute. The agent exclusively uses this region to hold its internal data structures. Any state created by the agent residing in one process could be accessed by its counterparts in other processes via the shared memory region. Agents in different processes or threads of the application communicate through the shared memory region arbitrated through `futex` based synchronization.

The agent starts the application by creating a child process and directly mapping the memory regions and segments described by the application's binary into memory. Loading of an application binary is typically performed by the kernel as a part of the `exec` system call. vPlay-user agent, however, implements the `exec` operation in user space. The goal in doing so is to retain vPlay-user agent's own memory regions within the application's address space as new memory regions of the application are added. Invoking the standard `exec` system call would cause all existing memory regions along with the agent to be unmapped and replaced with new memory regions specified in the application binary. Performing `exec` in user space allows vPlay-user agent's interception routines to be embedded within the application's address space.

vPlay-user agent installs a signal handler for the `SIGSYS` signal before transferring control to the application by jumping to the application's start address. Once the application takes control, any events posted by the `ptrace` subsystem will be handled by the signal handler which is a part of the vPlay-user agent. Installing a signal handler for `SIGSYS` is disallowed by intercepting and disabling the `sigaction` family of system calls. The signal handler takes necessary actions such as recording the system call return value and argument data when `SIGSYS` arrives. Application passes the system call arguments in processor registers which are saved on the signal stack by the kernel when the `SIGSYS` signal is posted to the application. Within the signal handler context, the agent is able to process the system call arguments by reading and writing to the signal stack. System call return value can also be altered as desired by modifying the respective register on the signal stack. The agent can virtualize the system call, emulate or nullify it, or process it in any other way it wants by calling the same system call with altered parameters or other system calls, all in user space.

`ptrace` notifications are disabled whenever the agent runs its own code. A flag in the `ptrace` extension indicates whether to post system call events to the application. When a monitored application thread makes a system call, the `ptrace` extension first resets the flag and sends a `SIGSYS` signal to the thread. Since the notifications are disabled, any system calls made by the agent code that runs as a part of the `SIGSYS` handler would not cause further signals to be issued. After the agent completes its processing, it re-enables the notifications and returns control to the application.

The memory region occupied by the vPlay-user agent is marked read-only while the application code runs so that potentially buggy application code does not accidentally corrupt the agent's memory. Any attempts to change the region permissions by the application are disallowed by intercepting the `mprotect` system call, which is the only user interface to change memory permissions. The permissions for the agent's memory region are changed to read-write by the `ptrace` kernel extension before posting the `SIGSYS` signal. This mechanism ensures that the agent retains secure control over the application's system calls even though it is a part of its user address space. The agent changes the permissions back to read-only before returning control to the application code.

vPlay-user agent emulates the `sigreturn` system call which is normally called implicitly at the end of the signal handler. At the end of the system call processing, control is transferred back to the application by returning from the signal handler. Normally, the signal stack is setup such that the application automatically calls the `sigreturn` system call when it returns from the signal handler. However, that would trigger another debug event resulting in an unbounded recursion. To avoid the problem, vPlay-user adjusts the signal stack to remove the implicit call to `sigreturn` and instead directly performs the `sigreturn` operation in user space. It involves appropriately loading the processor registers saved on the signal stack and jumping directly to the application code.

The initial process started by the agent and the entire process hierarchy rooted at the initial process automatically inherit the instrumentation. The agent's code and the shared memory region are automatically mapped into any children of the agent program and their successors at the same address, as a part of the `fork` system call. The signal handler state is also inherited by the children processes and threads, resulting in any signals received by the children threads to be directed to respective agent stubs mapped at the same address in all the processes. Events generated by children threads are posted to the respective threads and handled by respective copies of the vPlay-user agent embedded within the host process of the thread.

vPlay-user also installs the signal handlers for other signals such as SIGSEGV and SIGBUS, which indicate error conditions. If the application itself attempts to install a signal handler for a specific signal, the corresponding system call is intercepted and the function pointer of the application signal handler is separately saved to be called later. It allows vPlay-user to intercept exceptions caused by application failure, such as a segmentation violation or divide by zero, to cause checkpoints to be written to disk.

vPlay-user agent can be controlled by sending signals to the target application. vPlay-user registers a signal handler for a reserved user signal to enable external process to communicate with the agent. To control the behavior of the agent or direct it to perform an action, the user can simply send the reserved user signal to the respective thread. In particular, it allows an external process to periodically send the `start` and `stop` commands to enforce periodic partial checkpointing and to stop recording and write the checkpoints at any time based on an external fault detection system. Since the reserved signal sent to the application is first processed by the embedded vPlay-user agent, it is able to take necessary action before calling the original signal handler that the user may have installed. When the application is not being recorded or replayed, vPlay-user agent silently forwards all system calls and signals to the application. When the application fails due to an exception such as `SIGSEGV`, any data recorded until that point is saved to disk.

The basic `start` and `stop` primitives that control partial checkpointing are implemented by the vPlay-user agent. A recording interval commences with an external process sending all application threads a reserved signal to have them reach a barrier. At the barrier, the agent first records the current processor context of the thread. Register context is obtained from the signal stack and descriptor entries used by the thread are obtained through the

API provided by the operating system. On Linux, `get_thread_area` is used to read the GDT and `modify_ldt`, to read the LDT.

Like the kernel implementation, when an application is recorded, each thread within the application undergoes recording. Every input that crosses the application boundary is intercepted and recorded. Each thread in the application records its private processor state and one thread per-process records the common memory state. Partial checkpoints generated are stored in separate buffers by the vPlay-user agent within respective processes, and written to disk on request or when a failure is detected.

3.3.2 Tracking Accessed Pages

To record a partial checkpoint for an application, vPlay-user must determine the memory pages that are read by the application. vPlay-user leverages built-in support for dirty and accessed bits provided by the MMU hardware for this purpose. Since the hardware sets the bits transparently and automatically, there is no continuous additional cost in tracking the pages. The accessed and dirty bits are typically used within the kernel to implement page replacement algorithms and virtual memory. A simple extension to Linux's virtual memory subsystem exposes this information to the user space. Windows provides an equivalent API as a standard feature of the memory subsystem, which Windows applications use to implement sparse data structures etc. Since Linux does not currently expose these bits to user applications, we have implemented it as a part of vPlay-user.

The accessed and dirty bit information is typically used within the kernel to implement page replacement algorithms and virtual memory. vPlay-user shares the use of these bits with the kernel based on two pairs of additional shadow bits allocated in the page descriptor of the page, one to be used by the kernel and the other to be used by vPlay-user. The bit sharing is implemented by extending the macros used to set and reset these bits. The macro to reset the dirty or accessed bits is extended as follows. If the original bit in the page table entry was set, the kernel would set the shadow bit in the page descriptor meant to be used by vPlay-user. If it was originally reset, the same shadow bit in the page descriptor is reset. Similarly, vPlay-user would set the shadow dirty bit meant to be used by kernel before resetting the page table entry bit. If the bit was originally reset, it resets the same shadow bit. When the kernel has to read the bit, it would interpret the bit as set if either the page table entry bit or its shadow bit is set. vPlay-user's kernel extension provides a user interface to reset and query the net dirty and accessed bit information. In the case of hardware architectures, such as ARM, that do not provide accessed and dirty bits, their emulated Linux versions (young and dirty bits) can be treated as the hardware bits for the purpose of the above method.

The `start` operation calls the kernel extension to scan the page tables of the process and reset the accessed and dirty bits for each page in its address space. These bits will be set by the processor as the application accesses the pages during its execution. `stop` will query this information later at the end of the recording interval to determine the pages accessed by the application, to be included in the partial checkpoint.

If a page is modified by the application during a recording interval, the original copy of the page needs to be included in the partial checkpoint. To obtain the original copies of the dirty pages, vPlay-user leverages the copy-on-write mechanism implemented as a part of the kernel's `clone` functionality. At the beginning of each recording interval, a child *shadow process* is created. It shares all other resources with the parent except for its virtual memory. The shadow process exclusively acts as a backup copy of the parent's virtual memory, and does no processing other than to wait for requests from its parent. It never modifies any pages used by the application. The shadow and parent processes communicate through the agent's shared memory region arbitrated by `futex` locks. At the end of the recording interval, the `stop` operation examines the accessed and dirty bits of each page in the process address space, obtains the original copies of the dirty pages from the shadow process if needed and kills it. A new shadow process is created to track the original copies of the dirty pages for the next recording interval.

3.3.2.1 Incremental Partial Checkpointing

The page tracking mechanism as described above may include multiple copies of the same page in different partial checkpoints. For instance, if the same page is read by the application in two consecutive recording intervals, it is included in both the partial checkpoints. To avoid this duplication, vPlay-user implements a version of incremental checkpointing adapted to

Algorithm 2: Processing performed at the end of the recording interval to create a partial checkpoint

```
1   delete the oldest partial checkpoint;
    for each memory region in the process address space do
        syscall_record = search recent_maps to find the system call that mapped this region;
        for each accessed page page_address do
            if page_address was read then
                add (page_address, nil) to the initial_page_set;
            end
            if page_address is dirty then
                if syscall_record is not nil then
                    if syscall_record indicates that region is an anonymous private region then
                        page_data = zero_page;
                    end
                    if syscall_record indicates region is a file map then
                        page_data = get the original copy of the page from the file;
                    end
                    add (page_address, page_data) to syscall_record;
                end
                else region not mapped in current interval
                    page_data = get the original copy of the page from the shadow process;
                    add (page_address, page_data) to inital_page_set;
                end
            end
        end
    end
```

suit the semantics of partial checkpointing. The algorithm consists of two parts. The first part is implemented as a part of the `stop` operation (Algorithm 3) and the second part is implemented by the `write_checkpoints` routine (Algorithm 2), which writes the partial checkpoints stored in memory to disk when the application exits or encounters a failure.

A data structure, `initial_page_set`, represents the set of pages contained in a partial checkpoint. It consists of records of type (`page_address`, `page_data`), and indicates the initial set of pages loaded into memory when the application is resumed. Page addresses are unique within the set and `page_data` indicates the contents of that page. Some records of the set may only contain the page address without any associated page data, in which case, the page data would be indicated as `nil`.

At the end of the execution interval, `stop` queries the kernel module to determine which pages in the process address space have been accessed in the previous interval. A page can be in any of the following four states: not accessed (00), read from (01), written to (10), read from and written to (11). If a page was read (01, 11), its address is added to the `initial_page_set` with `nil` page data. If a page was written, the original copy of the dirty page is obtained from the shadow process. The page address and the original page data are added to the `initial_page_set`.

When the application either exits or encounters an exception, `write_checkpoints` (Algorithm 3) writes the accumulated partial checkpoints to disk. Partial checkpoints are processed starting from the earliest one to the most recent one in sequence. For each record in the `initial_page_set`, its `page_address`, is written. If the `page_data` is not `nil`, the page data is also written. If the `page_data` is `nil`, the `initial_page_set`s of the subsequent partial checkpoints are searched to check if the `page_address` exists. If a record with matching `page_address` is found in a subsequent partial checkpoint and the associated `page_data` is not `nil`, no page contents are written for that page. It implies that the page was overwritten in a subsequent recording interval and the original copy of the page will be saved in that partial checkpoint. There is no need to save the page in the present partial checkpoint. If such a record is not found, the current contents of memory at the address `page_address` is saved as the page data. Not finding a matching record implies that the current data at `page_address` was not subsequently modified and it is still valid. Similar processing is applied to each partial checkpoint in sequence.

Algorithm 3: Processing performed to write the partial checkpoints when an exception is encountered

```
1 for each partial checkpoint do
2     write the pages which have associated page data;
3     for the rest of the page addresses do
4         if the page data is not saved in any subsequent partial checkpoint then
5             save the current memory contents as the page data for that page;
6         end
7     end
8 end
```

3.3.2.2 Changes in Memory Region Geometry

If the application unmaps a region of memory during the course of a recording interval, the accessed pages in that region will be missed by the `stop` operation which is only called at the end of the interval. By that time, the memory region would escape the scanning performed by the `stop` operation as it would have been unmapped already. Similarly, if a new region is mapped during a recording interval, the shadow process would not contain the region. vPlay-user handles these cases with the following extension, which intercepts mapping and unmapping operations to capture the accessed pages in memory regions which have been mapped or unmapped during a recording interval.

System calls are recorded as an ordered list of `event_record`s. Similar to the kernel implementation, vPlay-user keeps track of system calls that modify memory regions by also logging them to the `recent_maps` stack. The stack tracks the system calls (`mmap, brk, exec` etc.) which have mapped a memory region within a recording interval. The `recent_maps` stack is emptied at the beginning of each recording interval. When the application unmaps a region of memory, the `recent_maps` set is searched to find the most recent system call `event_record` which maps the region encompassing the region being unmapped. If a matching system call record is not found, it implies that the region was not mapped in the current recording interval. The regions must have been inherited from a previous recording interval, and the accessed and dirty pages are added to the `initial_page_set` in the same manner as described earlier.

If a matching record is found, it is removed from the set, accessed and dirty pages in the region are recorded as described earlier, and the recorded pages are linked with the corresponding system call `event_record` in the main system call log to be written to disk by `write_checkpoints`.

If a memory region is mapped during the course of the recording interval, it would not be a part of the address space of the shadow process which is created at the beginning of the recording interval. The original data of the dirty pages of such a region is determined as follows. If the region is an anonymously mapped private memory region, the original data is marked as a special `zero_page`, to indicate that the page is initialized with zeros. Otherwise, if the page is mapped from a file, the original contents of the dirty page are obtained directly form the file. Since the page must have been recently accessed by the program within the current recording interval, it is likely to be in the file system cache and unlikely to cause disk IO.

In addition to the system calls such as `mmap` that explicitly map new regions into process address space, kernel implicitly maps new pages at the top of the stack as the stack grows. If a page within the stack region, which is not available in the shadow process but accessed in the current recording interval, is found, it is assumed that the page was mapped by the kernel and it's page address is added to the `initial_page_set` with `nil` page data. Since the kernel grows the stack with zero-initialized pages, if a dirty stack page which is not available in the shadow process is found, it is added to the `initial_page_set` with associated page data set to `zero_page`.

3.3.2.3 Initial Conditions

The very first `stop` operation after the application is loaded into memory has to record the pages that have been touched by the application since it is loaded. If any pages have been dirtied during that interval, the original contents of those pages have to be obtained and recorded. vPlay-user applies the same common case approach to the initial loading phase as well. Immediately after performing the user space `exec` operation, vPlay-user makes an implicit call to `start`. Among other steps, it records the initial processor context, where usually only instruction and stack pointers are defined, and creates a shadow process. When an application is initially loaded, it typically contains the application's text and data segments, dynamic loader's executable object, stack and heap regions. The stack region, in particular, contains command line arguments, environment variables etc. The shadow process created as a part of the user space `exec` operation would provide a pristine copy of these regions to the subsequent `stop`. Other shared libraries, such as the C library needed by the application, are loaded by the dynamic loader, which is considered a part of the application itself. The dynamic loader maps the shared libraries in the same way as the application itself would map them, and those regions are treated like other regions mapped by the application.

If one of the application processes forks a child process, the new child process is automatically placed within the purview of vPlay-user instrumentation. It's memory pages are tracked and system calls are intercepted and recorded. The very first `stop` command delivered to the child process would have to record the pages accessed by it since it was forked. vPlay-user implicitly performs a `start` which records the register context at the `fork` system call and creates a shadow process to preserve the original copies of the pages dirtied by the child process within the first recording interval.

Chapter 4

Logging

vPlay performs logging to collect necessary information and application state to deterministically replay each process and thread in a session from an initial state defined by the partial checkpoint through the end of the recording interval. Logging serves two functions. First, it records necessary data, including executable code, which may not be available at the target environment where the application is replayed. Second, it captures information related to the outcomes of nondeterministic events to ensure a deterministic replay.

Memory constitutes a significant portion of the log. The mechanism used to track accessed memory pages that are mapped during the recording interval and add them to respective system call `event_record` is similar to that used for tracking accessed pages for partial checkpointing. vPlay records in order, all systems calls that map memory to a per-process `recent_maps` stack, including a reference to the corresponding system call `event_record` in the per-thread log queue. When a page is first accessed that was mapped during the recording interval, a page fault will occur and vPlay will search the `recent_maps` queue to find the most recently mapped memory region corresponding to the page, which is the current mapping being used by the thread. The page is then added to the respective system call `event_record` or to the respective `shared_memory_object` if it is for a shared memory region.

To ensure deterministic replay, vPlay continuously monitors the application to intercept and record nondeterministic events in its execution. In the piece-wise deterministic execution model of application programs, a replaying program follows an execution course identical to recording as determined by program logic as long as there are no nondeterministic events. In order to make an execution interval deterministic, it is sufficient to record the outcomes of the nondeterministic events that occur within the interval and enforce the same outcomes at replay.

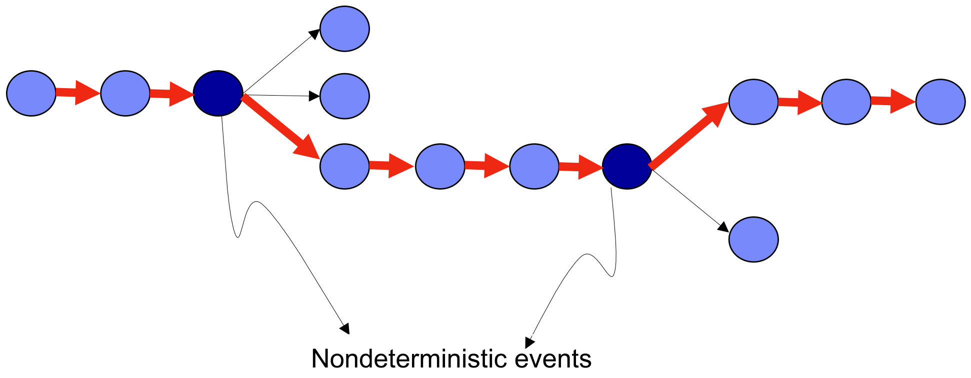


Figure 4.1: Piece-wise deterministic execution model

Nondeterminism originates from various sources such as hardware interrupts, scheduling etc. But from application's perspective, it boils down to four types of events: nondeterministic instructions, system calls, signals, and concurrent accesses to shared memory. vPlay leverages mechanisms provided by Scribe [35] to handle nondeterminism due to hardware instructions, signals and shared memory interleavings, and provides different mechanisms for logging system calls to support replay debugging. vPlay also provides new mechanisms to integrate logging with partial checkpointing to support seamless replay of the application in a different environment. Nondeterministic events are captured in `event_records` which are stored in order in per-thread log queues. Each `event_record` consists of a header followed by data of arbitrary length.

Nondeterminism due to hardware instructions is handled by setting the processor flags to disallow executing these instructions in user mode, causing them to trap into the kernel where the returned values are recorded. By default, they are directly accessible to the application and need no operating system intervention. On x86 CPUs, there are three such instructions, `rdtsc` perhaps being the most common, and they all involve reading CPU counters.

4.1 System Calls

For most system calls, vPlay simply records the system call return value and data, then provides it back to the application during replay. The system call results are captured as unstructured data and replayed as such. vPlay does not modify or otherwise attempt to interpret it. Returning the values on replay instead of re-executing the system calls provides a simple mechanism for handling most common sources of nondeterminism. It includes nondeterminism due to system calls such as `gettimeofday`, `select`, `read` and `write`. For example, consider the case where two processes concurrently write into one end of a pipe, and a third process reading from the other end. Normally, this leads to nondeterministic execution. The data read by the third process depends on the interleaved order in which the first two processes are scheduled. However, vPlay decouples this interprocess interaction by independently recording the system call results. During replay, the writers get the number of bytes written into the pipe as observed during recording, and the reader is passed the data from the log independent of the writers, thereby removing the nondeterminism and the dependency between the reader and writers. The same applies to synchronization and interprocess communication mechanisms such as pipes, semaphores, message queues, file locks, etc. where processes interact through the system call interface.

vPlay uses a data plug-in which encodes the system call interface of the operating system to record the system calls. vPlay consults the plug-in to determine which system call parameters carry data to be returned to the application and their sizes. Each system call is logged as an `event_record` containing the system call return value and data returned to the thread through the system call parameters. For each system call, the plug-in encodes the following three pieces of information: the system call service number, the number and sizes of the system call parameters, and a boolean value indicating whether a specific parameter may contain return data. In addition, the plug-in also specifies the calling convention used by the source operating system to invoke the system call. For instance, Linux/x86 and BSD/x86 use `int 0x80` or `sysenter` instructions to trap into the kernel to service a system call. The system call service number is placed in `eax` register and the system call arguments occupy respective registers. This approach of using a data plug-in decouples the record-replay mechanism from the system call semantics of the operating system and makes both vPlay and partial checkpoints portable across different operating systems.

There are three types of system calls which need additional processing beyond log and replay of their return data:

1. **Process control** (`clone, fork, vfork, exit, exit_group`): Since partial checkpoints are taken per-process, system calls for process control need to allocate and deallocate state for recording partial checkpoints, and log queues to record `event_record`s. When a new process is created, the child process is included within vPlay instrumentation by attaching to the new process or through an implicit copy of the agent into the child process in user space implementation. If `exit` system call is encountered, the partial checkpoints maintained in memory are written out to disk.

2. **Address space geometry** (`mmap, munmap, brk, execve`): For system calls that manipulate memory regions such as `mmap`, vPlay logs the memory pages which are mapped during the course of the recording interval, including code pages from shared libraries or data pages from other memory mapped files used by the application. When a system call maps a new memory region into a process address space, the associated `event_record` will contain the subset of pages from the newly mapped region that were accessed within the recording interval. In addition, the newly mapped region is pushed on to `recent_maps` stack. When a region is unmapped (`munmap`), the corresponding region is removed from `recent_maps`.

3. **MMU context** (`set_thread_area, modify_ldt`): When a system call successfully adds a new descriptor to a descriptor table, the linear address of the segment base as represented in the descriptor is associated with the `event_record` of the system to be logged along with other return data of the system call. No special processing is required for `get_thread_area` or when `modify_ldt` is used to read a descriptor, except for recording the system call return data.

4.2 Shared Memory

Replaying shared memory interleaving is critical for deterministic replay, especially on multiprocessor machines. Memory sharing happens either explicitly when multiple processes share a common shared mapping, or implicitly when the entire address space is shared, e.g. with threads. Replaying the order of memory accesses efficiently in software is difficult since one process may access shared memory asynchronously with, and at any arbitrary location within, another process's execution. When applications use shared memory without explicit locking mechanisms, the operating system is completely unaware of these race conditions. In the absence of explicit calls to the operating system, it is difficult to transparently instrument the application code involved in the sharing. Since memory accesses are quick and frequent, it is also difficult and inefficient to log individual accesses to the shared memory. The rest of this section focuses on describing the algorithms and mechanisms to address the nondeterminism originating in interleaved access to shared memory.

On uniprocessor systems, the order of accesses to shared memory may be reproduced by replaying the order and precise length of process time slices. Instead of logging every access to the shared memory region, it is sufficient to log one record per scheduling period specifying the process identifier of the process that was scheduled in that period and the number of instructions executed by it in user mode. However, this method cannot be independently applied to each processor of a multiprocessor system. Two threads running on two different CPUs could simultaneously access a word in memory and the cache-coherency hardware may arbitrarily choose one of the threads to access the word first.

When a process either reads or write to a memory word, the process has a logical exclusive access to the location, during the span of the instruction making that access. In the absence of exclusive access, memory operations would have unpredictable results. Figure 4.2 represents such a memory access as a small solid rectangle on a graph with process address space plotted on y-axis and time on x-axis. We refer to it as an *access event*. An access event represents a unit of recorded information. No two access events can overlap because the process logically holds an exclusive access to the memory location for the access interval. The specific dimensions of an access event depend on the hardware. For example, the hardware may provide exclusive access to a word even though only one byte

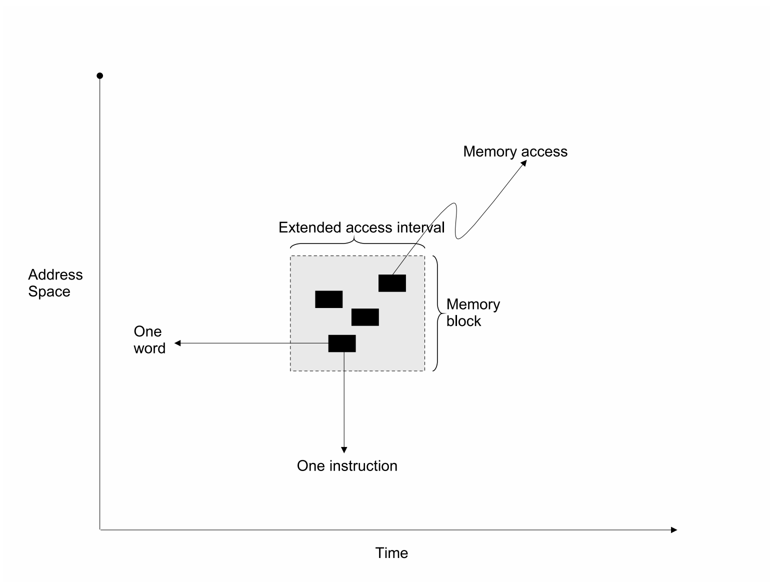


Figure 4.2: Clustered Shared Memory Accesses

is accessed by the process. Applications are unaffected by these hardware attributes and it is possible to artificially change the length of the exclusive access interval or the size of memory block without affecting application level semantics. By providing exclusive access to a block of memory for a duration extending beyond the access interval, several thousands of memory accesses by a single process can be combined into one "composite" access event. In effect, the dimensions of the access event are increased from one word to a memory block and one instruction to an extended access interval, as shown by the dotted rectangle in Figure 4.2. By locality of reference, if a process accesses a memory location, it is likely to access the same location, or other locations near it, in the near future. As shown in the graph, memory accesses by a process tend to occur in clusters. By aggregating near-by words in memory and near-by accesses in time, and providing the processes exclusive access to such aggregated units in space and time, the total number of access events to be recorded would be reduced to a manageable order.

The mechanism to track shared memory interleaving is implemented within the operating system. Convenient choices for aggregated units of time and space at the level of operating system are process time slice and memory page respectively. While the operating system cannot control access to individual memory words, it manages process address space in the units of memory pages. While it doesn't control the execution of individual user instructions, it arbitrates access to the CPU in the units of process time slices. By raising the granularity of shared memory accesses in space and time to the units controllable by the operating system, it is possible to efficiently record shared memory interleaving.

When a shared page is first accessed by a process, it is given exclusive access to that page until it is scheduled out. Each such ownership assignment is logged by recording the identifier of the process, the page number and the duration of the exclusive access, along with a monotonically increasing sequence number to indicate the order. The duration of exclusive access is measured in terms of the number of user-mode instructions executed by the process while owning the page. A process could acquire exclusive access to multiple pages during a time slice. To guarantee deadlock-free operation, each process releases all its pages when it is scheduled out. If a process attempts to access a page which is already owned by another process, it is suspended until the page is released.

In order to implement the above algorithm, two key operating system mechanisms are needed. (1) A mechanism to selectively assign ownership of a shared page to a specific process for an interval and (2) the ability to preempt the execution a process at a precise point in its instruction stream and transfer page ownership to another process. Sections 4.2.1 and 4.2.2 describe these two mechanisms respectively.

4.2.1 Selective Page Ownership Assignment

The main tool to monitor and control memory access in software is the page protection mechanism. The operating system virtual memory manager uses the page-fault interrupts to detect accesses to absent pages, and to implement deferred allocation of pages through copy-on-write mechanism. The same hardware support is used to intercept accesses to writable pages in shared memory. Each operating system process is assigned a page-table data structure which the processor uses to decide which portions of the address space are mapped to physical pages. Each page in the address space is represented by an entry in the page-table called the page table entry. A page table entry contains architecture dependent protection and status bits, based on which the processor takes appropriate actions. In particular, the `page-present` bit indicates whether the corresponding page is present in the physical memory or not. It is possible to force the processor to generate a page fault interrupt when a page is accessed, by clearing the `page-present` bit in the respective page table entry.

When a physical page is shared by two processes, it is represented in the page tables of each process by a corresponding page table entry indicating the mapping. It is therefore possible to give exclusive access of a page to one process, while denying access to another, by setting or clearing the `page-present` bit in their respective page table entries. Threads which share their entire address space, also share their page tables and the page table entries. This is handled by maintaining a set of shadow page tables, one for each thread along with a reference page table within the host process. When a process is scheduled to run, the corresponding shadow page table is loaded into the processor's MMU. When the reference page table is modified, as page mappings are added, changed or removed, the change is propagated to the per-thread page tables.

Exclusive access to shared memory is implemented by instrumenting the operating system's page-fault interrupt handler. An unused bit in the page descriptor is used as a flag that indicates whether exclusive access to the corresponding page is already assigned to another process. If a process accesses a page which is already assigned to a different process, the task is suspended and placed in a wait-queue where it waits until the page is released. When the original process eventually releases the ownership to the page and the ownership is assigned to the waiting process, the page fault handler sets the `page-present` bit in the page table entry of the process, and lets it proceed and access the page. The process now holds exclusive access to the page for the rest of its scheduling period. During this time, the process can access the page any number of times without generating a fault. Any process running in parallel on another processor would cause a page-fault, if it attempts to access the page during this time. Eventually, when the process which is given exclusive access to the page is suspended, either at the end of its time slice or due to a blocking system call, the scheduler releases all the shared pages held by the process. The `page-present` bit and flag bits for every shared page that the process ever acquired access to are cleared.

4.2.2 Page Ownership Transitions

To ensure deterministic replay, precise locations of transfer of page ownerships between different processes have to be recorded. An ownership transfer may occur at an arbitrary point in the instruction stream of a process and the ownership must be preempted at the same point of execution during replay. We describe two alternative approaches to record and replay the point of ownership transfer, one based on hardware instruction counters (Section 4.2.2.1) and another based on sync points (Section 4.2.2.2).

4.2.2.1 Using Hardware Counters

Hardware performance monitoring counters [30] are typically used for performance analysis and statistical application profiling by periodic sampling of counter values. These counters can either count events or measure durations. When counting events, a counter is incremented each time a specified event takes place or a specified number of events takes place. In particular, most hardware counting facilities can count the number of retired instructions in user mode. The value of this counter can be read or written to using an instruction provided for that purpose. The length of the period of exclusive access can be measured by resetting the counter when the process is first given access to a shared page and reading the counter value when it is scheduled out. In addition, hardware counters provide the facility of notifying the operating system by means of an overflow interrupt when the counter value reaches zero. This can be used to force the replaying process to execute a specified number of instructions by placing a negative value in the counter and generating an interrupt once the process has executed those instructions.

However, hardware counters suffer from two key limitations. First, the instruction which reads the value of a counter is not serialized with respect to the other instructions in the processor pipeline. The instructions are executed out-of-order and the reported instruction count could be inaccurate. Second, there may be some latency in the generation of the overflow interrupt. The interrupt handler may be activated well after the overflow is reached. Meanwhile, the processes continue to execute and trigger event counters. Since the precise number of instructions executed by the process has to be recorded and replayed, the above limitations make the hardware counters unusable as such.

Instruction counter inaccuracy is overcome by using processor support for setting breakpoints at specified instruction pointer values and a checksum of register context. Although the instruction count value is inaccurate, accurate values for the program counter and other processor registers are available. Operating system uses this state information to restore the processor's register context when a process is rescheduled after it had been previously suspended. Along with the inaccurate instruction count, program counter value and a checksum of the processor state containing these register values are recorded. In order to stop the replaying process when the stipulated number of instructions have been executed, as specified in the recorded log, the instruction counter is first set to overflow well before the actual number of instructions, while ensuring that the overflow interrupt is generated in the proximity of the target instruction, but before it. In the interrupt handler of the overflow interrupt, a breakpoint is set to the instruction, as specified by the program counter value. This generates a breakpoint interrupt when the process reaches the instruction pointed to by the program counter. However, the process may not have reached the desired point of execution due to the presence of loops in the program. Such cases are discriminated by comparing the processor register state with the recorded checksum. If the checksum of the current processor state matches that of the recorded value, then the process is considered to have reached the same point of execution as the corresponding point in its primary counterpart. If not, the process is released and its execution is continued until it hits the breakpoint again.

The success of the above algorithm depends on the instruction margin used to set the overflow interrupt and the strength of checksum. For Intel processors, our initial measurements show an empirical value for the instruction margin to be about 50 instructions and for PowerPC processor, we found the same value to be about 15 instructions. By including the general purpose registers in a 32-bit checksum, we have been able to implement this procedure for common applications like PostgreSQL. Rare cases of checksum conflict could be addressed by including the memory values on the stack and even the contents of memory locations pointed by the general purpose registers. Furthermore, this method is not employed in every scheduling period. It is only used if the primary process was asynchronously descheduled by an interrupt. Typically, processes are descheduled when they encounter a blocking system call or other synchronous event. In such cases, the system call itself would provide the reference point to preempt the process. The processing necessary to record the instructions executed during the preceding scheduling period is performed by the operating system when the task is suspended, while the task is in the privileged mode. Therefore, the corresponding instructions are not counted, because the instruction counter is programmed to count user mode instructions only.

The size of a typical hardware instruction counter on x86 architecture is 32 bits. The counter value overflows after about four billion instructions and the maximum instruction count value that ever needs to be recorded is bound by the length of scheduling period used by the operating system scheduler. On a fast processor, it is possible for the counter value to overflow while counting the instructions. In such case, the overflow interrupt handler is instrumented to account for the overflow.

4.2.2.2 Using Sync Points

The method described in Section 4.2.2.1 accurately records and reproduces shared memory interleaving by providing a mechanism to log precise locations in the execution of a process and preempt it at the same point during replay. However, it may not be necessary to record arbitrary execution points, as long as the ownership is transferred at a point which is well defined both at record and replay with reference to other synchronous kernel events. Sync points [35] provide such a convenient location for ownership transfers. vPlay adopts the sync point approach for ownership transfers because it makes the mechanism simple and more efficient.

vPlay employs a concurrent read, exclusive write (CREW) protocol [13] to manage page ownerships, and then uses sync points to ensure that transitions occur at precisely the same location in the execution during both recording and replay. Page state transitions are allowed to only take place when vPlay can conveniently track, and later replay them. When a process tries to access an owned page, it faults, notifies the owner, and blocks until access is granted. Conversely, owner processes check for pending requests at every sync point and, if necessary, give up ownership. Note that page faults due to the memory interleaving under the CREW protocol contribute significantly to the pool of sync points.

Although transfer of page ownership is always performed by the owner process, there is one exception to this rule due to interaction of blocking system calls and shared memory. When an owner of a page blocks inside a system call, it cannot transfer its page ownership to another process. This can cause long delays in ownership transfer and even lead to a deadlock. To address this problem, vPlay guarantees that user space shared memory is not accessed by an owner process when it is executing a system call. If another process needs to access a shared memory page owned by the calling process, vPlay can simply transfer ownership to the requesting process knowing that the original owner process will not access shared memory because it is executing a system call. There are no shared memory interleavings to track between the original owner process and the requesting process. vPlay can just identify the location in the original owner's instruction stream at which this ownership transfer occurs as being the occurrence of the system call, which it already logs.

Chapter 5

Virtual Replay

Replay across different operating system environments requires (1) data level independence from the source environment and (2) a virtualization layer which insulates the application from the target to correctly playback the recorded data. Partial checkpointing and logging mechanisms described in Chapters 3 and 4 are designed to provide a complete and self-contained recording of all data necessary to replay the application's execution. The state included in the recording is selected such that the application's replayed execution is completely decoupled from the target environment. In addition, the target environment must also provide a mechanism to launch the application from the recorded data and insulate its execution from the discrepancies of the target host environment.

5.1 vPlay Container

Even with all the data available, natively running an application built for one operating system on a different one is difficult. Applications are typically compiled for a target architecture, operating system and environment. Specific features, interfaces, memory model etc. tie an application to its target operating system. In order to run an application built for one operating system on another, it has to be explicitly ported to the target by making appropriate changes to the source code. Unless the application is originally designed for portability, even porting may not be straight forward and may amount to rewriting the entire application.

Processor emulators can run applications built for one architecture on another but still require the same operating system. Virtual machines such as Qemu [5] and Xen [4] provide full system emulation so that they could host a different guest operating system, which in turn could run the applications built for it. However, they cannot allow an application to run natively without the guest operating system to provide the system services. Since they need to encapsulate the guest operating system and its applications in their entirety, the state associated with them is typically too large for easy sharing with the developers and may contain sensitive client data. Furthermore, they tend to obfuscate application level events necessary for debugging by introducing additional intermediate layers of software.

vPlay provides vPlay Container, a thin virtual replay environment which consistently emulates the semantics of the application's interfaces with the target by decoupling its processes from their address space, memory, binaries, CPU and MMU structures, operating system resources and services, and each other. vPlay Container decouples the application from the target in three key areas which enable it to run the application on a different operating system than the one it is built for. First, the application is decoupled from the memory address space and its content. In particular, the application is decoupled from its binaries by trapping accesses to the code pages and presenting the actual pages captured at the source, thus avoiding any version discrepancies. The ABI incompatibility is addressed through the portable design of the partial checkpoint and the partial checkpoint loader. Second, applications are decoupled from the processor MMU structures such as segment descriptor tables through a binary translation technique designed specifically for user application code, which traps and emulates the offending instructions during replay. Third, the application is decoupled from the operating system by abstracting the system call interface through emulation and replay. System call emulation decouples the application from the various operating system specific resources and dependencies. It also decouples individual processes of the application from each other by virtualizing all inter-process interactions.

5.1.1 Memory Address Space and Content

A key requirement for heterogeneous replay is that the same address space regions used by the application during recording be provided to the application during replay. Since vPlay captures non-relocatable chunks of application binaries directly from application's memory, they have to be loaded at the same address offsets at replay. However, the required address regions may not be available because, as per the general memory layout of the target operating system, the memory regions may be reserved for the operating system or the system libraries. For example, the default Linux/x86 configuration reserves top 1 GB of address space for the kernel use, whereas the default Windows configuration does not use the same size address space layout. Furthermore, system libraries such as Windows' kernel.dll and Linux's virtual dynamic shared object (VDSO) require to be loaded at specific address offsets which they reserve for themselves, preventing the use of their address regions by general applications.

Virtual machines and emulators decouple the user code from the target system by creating a virtual MMU, which maintains the internal state of the MMU in software and uses an elaborate scheme to emulate the memory management features such as segmentation and page fault mechanism. It requires extensive fine grain instrumentation or specific hardware support to implement a virtual MMU. However virtual machines cannot avoid the additional complexity and cost because they virtualize an entire guest operating system instance, which assumes exclusive control over the entire memory address space and resources available to it. Virtualizing the memory address space used by the guest operating system itself and low-level firmware and devices is particularly difficult.

vPlay Container avoids full emulation of the processor MMU by limiting the scope of its virtualization to the replaying application. Applications only have a subset of the total address space available to them, which excludes the regions used by the operating system, firmware and hardware, which significantly simplifies the requirements. Furthermore, common operating systems share the basic memory layout on a given architecture and typically allow the user to configure the way the linear address space is partitioned between the user and kernel space using a boot-time switch. For example, Linux/x86 and Windows/x86 allow the user to partition the 4 GB address space between user and kernel at 1 GB granularity. To provide the address regions required for replay, it is sufficient to align the application address space between the source and target systems. For example, to record Linux/x86 applications and replay them on Windows/x86, a simple way to avoid conflicts is to configure Linux/x86 and Windows/x86 to allocate the bottom 2 GB and 3 GB, respectively, of address space to application programs. That way, Windows system libraries, which only occupy a small region immediately below the kernel region, will not conflict with the application's pages in the bottom 2 GB of address space.

vPlay Container decouples the application from the binaries and the memory state it requires by directly loading the memory content from the recording. When the application maps a new address region, the memory pages which it is going to access from that region are brought into memory from the log. This mechanism also enables replay on a different operating system distribution, regardless of the environment and packages installed. For instance, Linux kernel automatically maps the VDSO region that occupies a memory page within the process address space. A compatible C library uses it as a stub for system call entry. Since both the VDSO page and the respective pages from the C library within the application that use the system call entry stub are obtained from the source environment, the replaying application will run successfully even though the target kernel and the C library in use are different.

5.1.2 Instructions and MMU/CPU Structures

Most instructions dispatched by the application are executed natively. Unlike virtual machine emulators, vPlay does not need to process privileged instructions since a recording never contains them. vPlay only tracks pages within the application address space. Any privileged instructions such as `in` or `cli`, which may be executed as a part of the system calls, are not included. However, there are two classes of instructions that vPlay may need to emulate: (1) instructions explicitly referencing user created segments, and (2) instructions that invoke a system call.

vPlay Container virtualizes the replaying application's access to the global processor structures by emulating the instructions that reference them. The x86 architecture provides global (GDT) and local (LDT) descriptor tables, which describe memory segments in its segmented memory model. They are mostly used and managed by the operating system. However, a limited interface to access them is exposed to the system libraries which implement low level system services. For example, multi-threaded applications on Linux create private memory segments by adding segment descriptor entries to these tables to store per-thread local state, which can be uniformly accessed via instructions that reference the entries. Because the GDT and LDT may be managed differently by different operating systems, vPlay virtualizes application's access to the tables. If the application is replayed on a different operating system from where it was recorded, vPlay emulates those instructions to avoid conflicts with the target operating system.

vPlay intercepts the instructions used to invoke a system call, and emulates the call itself. On the x86 architecture, the interrupt descriptor table (IDT) contains the entry point for the system call interrupt, which is invoked by executing the `sysenter` instruction or the `int` instruction with the index of the system call interrupt descriptor. vPlay intercepts these instructions and emulates the respective system call based on techniques developed in RR [6]. For most system calls, emulation is done by simply returning the results of the system call from the recording, bypassing kernel execution.

5.1.3 Operating System Resources and Services

Emulating system calls provides a uniform and consistent method to decouple the application from various types of operating system resources. Applications alternate between user and kernel space execution, typically performing most of their work in the user space, while delegating resource allocations and other privileged operations to the operating system kernel. The user space portions of an application's execution, by definition, do not depend on the kernel services and can be executed independently, even on a different operating system. The kernel space portions of execution occur through well-defined system call interface, and can be collapsed into a quick replay of the system call results to the application, thus bypassing the kernel execution. There are three classes of exceptions in which system calls must be emulated instead of just returning their results: system calls for process control, system calls that modify address space geometry, and system calls related to MMU context. We discuss these in further detail in Sections 5.3 and 5.4. Replaying the system call results is done in an operating system independent way by vPlay on behalf of the application. As long as the application receives consistent responses to the system calls it makes, the application continues to run as expected.

Any differences in the system call API between the source and target operating systems does not affect replay since the replaying application never directly contacts the target system. The application will replay consistently even though the system calls it makes are unavailable or have different semantics.

5.2 Virtual Replay Mechanism

To replay a piece of previously recorded application, the user chooses a process and an interval of execution to replay by selecting the corresponding partial checkpoints. To reproduce a deterministic replay of interleaved shared memory accesses among application processes, vPlay computes a *shared memory closure* of the selected process and replays all processes in the closure together as a session. A shared memory closure of a process `p` is the smallest set of processes consisting of `p`, such that no process within the set shares memory with a process outside the set. All threads within each process in the closure are included in the session and replayed together.

To aid debugging, replay can also be done across consecutive recording intervals by coalescing the partial checkpoints and concatenating the respective logs. The pages required by a process is computed by consolidating the partial checkpoints representing the interval. In particular, the new `initial_page_set` is computed by taking the union of `initial_page_set`s of individual partial checkpoints. If a particular `page_address` appears in the `initial_page_set` of more than one partial checkpoint, the element with non-`nil` `page_data` in the earliest partial checkpoint is added to the new `initial_page_set`. During replay, memory pages accessed by the application are loaded into memory in stages. The application is initially resumed with the pages contained in the `initial_page_set`. The rest of the pages accessed by the application during the course of its execution are loaded progressively at each system call that maps the region. When the application makes a system call that maps a new memory region during replay, the corresponding system call `event_record` would contain the set of pages to be loaded into memory at that point.

Virtual replay consists of two phases. (1) *Load phase*, where the coalesced partial checkpoint of each process in the session is restored. (2) *Replay phase*, where the application threads are deterministically re-executed within vPlay's control. Transition from load phase to re-execute phase occurs when control is transferred to the application code. With vPlay instrumentation, the control is transferred with an implicit `sigreturn` at the end of the signal handler that loads the processor context. In case of replay through Pin (Section 5.4, control is transferred through a special system call which is intercepted and interpreted by loading the processor context.

Load phase is performed by the partial checkpoint loader. As a part of the load phase, the partial checkpoint loader prepares the process context required for the application to run independent of the target. It includes creating and populating the memory regions, creating the application processes and threads, and loading user created segment descriptor table entries. If the target operating system's segment layout matches with that of the source and it provides an API to access the tables, vPlay loads the entries into the tables. If not, emulation of instructions that explicitly reference the segments is done by vPlay.

5.2.1 Integration with the Debugger

vPlay integrates with a standard interactive debugger in order to closely monitor and analyze the execution of the application being replayed. The virtual replay mechanism itself just resumes the application and replays it for a specified interval. However, for this process to be useful for root cause analysis, it is necessary to expose the finer steps taken by the application during its execution. Particularly, it is necessary to allow the programmer to set breakpoints at arbitrary functions or instructions, watch the contents of various program variables at each step etc., at the source code level.

vPlay integrates with GDB by providing a GDB script that directs the load phase until the application is fully initialized for the user to start interacting through the debug interface. It also contains the necessary GDB directives to load the symbol information for the application being debugged. The script begins the debugging session with the invocation of the program that performs the load phase as the debuggee, which reads the partial checkpoint files, reconstructs their address space and initializes their threads. The debugger does not intervene during this process. The latency of the load phase is usually imperceptible to the user. After the application is loaded, a single forward step within the GDB script transfers control to the application code. The application is presented to the user in a stopped state while the debugger shows the register state and the source line of the application a few moments prior to the failure. The user can then set break points, single step through the source lines to examine program variables and monitor application's interactions with the operating system and other processes, to analyze the root cause.

Any inputs needed by the application are automatically provided by vPlay. For instance, when the application attempts to read from the console, the input is directly provided from the log rather than waiting for user input. When the application executes the system call interrupt instruction in a debugging session, the perceived state of the application's registers and memory after returning from the instruction would be identical to its state at the corresponding point during recording. Any outputs generated by the application are captured into an output file. Any graphical output produced by the application may not be directly visible to the user since the window manager may be external to the application and not recorded. However, the user can observe the interactions between the application and the window system within the debugger interface.

When vPlay-user is used, although the instrumentation mechanism itself is based on `ptrace`, it does not interfere with existing debugger semantics of the `ptrace` interface. Any `ptrace` notifications destined to external processes continue to occur. However, as the application executes, the `ptrace` subsystem generates additional application events in the form of signals that notify vPlay-user agent of the application's events. Although these events are extraneous to the application, they do not perturb its execution. Most debuggers allow these events to be masked from the user during a debugging session. For example, GDB can be configured to not print the notification signals by adding `"handle SIGSYS noprint"` to `.gdbinit`.

Category	**System Call**	**Linux**	**Windows**
Process control	`fork`	emulate with `fork`	emulate with `CreateProcess`
	`clone`	forward to the OS	emulate with `CreateThread`
	`exit_group`	wait for other threads	wait for other threads
Memory geometry	`mmap, brk, execve`	emulate with `mmap` and `munmap`	emulate with `VirtualAlloc`
	`shmat`, `mmap` with `MAP_SHARED` flag	emulate with `shmat`	emulate with `MapViewOfFile`
	`munmap`	forward to the OS	emulate with `UnmapViewOfFile`
MMU context	`set_thread_area, modify_ldt`	forward to the OS	update `selector_base` table

Table 5.1: vPlay system call emulation

5.3 Replay Across Linux Distributions and Kernel Versions

5.3.1 Load Phase

The load phase is performed by the Linux version of the partial checkpoint loader. It is implemented as a statically linked program which creates the application processes, restores their address space, places them in a vPlay Container and transfers control to the application code. The partial checkpoint loader itself is built to be loaded at an unconventional address region to avoid conflicting with the pages of the application and does not use the standard program heap or stack. The partial checkpoint loader begins by creating the per-session shared memory regions as defined by the `shared_memory_objects`, and mapping them into its address space. The sparse set of memory pages in each `shared_memory_object` are then loaded into respective shared memory regions, and the regions are unmapped.

A set of processes, each to become one of the processes recorded in the partial checkpoint, are recursively created with unconventional address regions used as their stacks, to avoid conflict with the application's stack pages. Each process begins restoring itself by attaching to the shared memory regions indicated by the `shared_maps` set in its partial checkpoint. Each page in the `initial_page_set` is then mapped as an **independent, private, anonymous, writable** region and its initial page content is loaded. After the page content is loaded, its protection flags are set to their original recorded values through `mprotect` system call. For example, if the page was originally a file map of a read-only shared library, it is first mapped as a writable anonymous region to load its contents, and the original page permissions are restored afterwards.

After the process address space is prepared, they are placed within a vPlay Container, initialized with pointers to the log queues. Each process recursively creates its threads, which load respective descriptor table entries using the Linux API, and enter a `futex` barrier. Once all threads reach the barrier, the main replay thread invokes vPlay to attach to the threads and start replaying. Each thread then executes the instructions to restore the processor registers. When the instruction pointer is finally restored through a `jmp` instruction, the thread starts running the application code.

5.3.2 Replay Phase

Even though vPlay Container provides sufficient decoupling to replay across different operating systems, for the sake of efficiency, vPlay omits the virtualization of segment descriptor tables when replaying between different Linux systems. Since different Linux versions manage the GDT and LDT in the same way, and provide the API to load the entries required by the application, vPlay directly restores the segments captured from the source and allows the application to natively execute the instructions that reference the user segments without emulation. The target Linux system would correctly interpret the instructions as per the segment entries established on the target processor during the load phase.

Most system calls made by the application are handled by simply copying the data from the respective `event_records`. Table 5.1 lists three main classes of exceptions, where further processing is performed beyond data copy. In particular, for the `fork` system call, vPlay creates a new child process and preloads the pages indicated in the `event_record`. These pages include the pages accessed by the child process in the recording interval which were not present in the parent's address space. For the `exit_group` system call, vPlay defers its execution until all other threads in that process exhaust their `event_records`, to avoid their premature termination. For system calls that map a new memory region (`mmap`, `brk`, `execve` etc.), the pages indicated in the system call's `event_record` are mapped and preloaded into memory. For system calls that map a System V shared memory region or a shared, memory mapped file, the `shared_memory_object` indicated in the `event_record` of the system call is mapped. For `clone`, `set_thread_area` and `modify_ldt` system calls, the system call is simply forwarded to the underlying kernel. The interleaving of shared memory accesses as recorded in the event stream is enforced among replaying processes and threads and any signals received by the application within the interval are delivered at respective points using the Scribe [35] mechanisms.

5.4 Replay Across Linux-Windows

vPlay Container for replaying Linux applications on Windows is implemented using Pin instrumentation [39]. It is conceptually similar to replaying on Linux as discussed in Section 5.3. We highlight the steps which are different below.

5.4.1 Load Phase

The load phase is performed by the Windows version of the partial checkpoint loader in user space using the Windows API. To replay the application, the partial checkpoint loader itself is started under the control of vPlay *pintool* [39]. Replay is invoked on the command line as: `pin -t vPlay.dll -- loader.exe <recording file>`. `vPlay.dll` is a pintool which implements the replay phase and `loader.exe` implements the load phase.

vPlay pintool does not interfere with the loading process performed by the partial checkpoint loader. The creation of processes, partial reconstruction of their address space and creation of threads within them is performed as already outlined, except using equivalent Windows APIs. Once the partial checkpoints are loaded, each thread leaves the synchronization barrier and makes a special system call, which is normally undefined in Linux and Windows. The system call activates vPlay pintool by notifying it of the completion of the load phase and transition into replay phase. vPlay pintool reads the respective log file of the thread to obtain its saved processor context and loads it using Pin's `PIN_ExecuteAt` API function, which turns the control over to the application code.

5.4.2 Replay Phase

vPlay pintool continues with the replay phase to monitor the application to satisfy the requests it makes. In particular, vPlay emulates the key categories of the Linux system calls listed in Table 5.1 using equivalent Windows APIs. For other system calls, vPlay pintool traps the system call interrupt instruction, copies system call return data to the application, increments the instruction pointer to skip the system call instruction and allows the application to continue normally. In the absence of such a mechanism, executing the Linux system call interrupt instruction would cause a general protection fault on Windows. When new memory regions are mapped, respective memory pages that will be accessed by the application in its future execution are brought into memory in a way similar to Linux replay, except using the Windows semantics. For instance, Windows treats memory address space and the physical memory that backs it as separate resources, whereas Linux transparently associates physical pages to memory mapped regions. To emulate the Linux system calls that map new memory regions, vPlay reserves both the address space and the memory together.

Instructions explicitly referencing user segment registers are treated through a trap and emulate mechanism. Windows configures the CPU descriptor tables in accordance with its memory layout which is different from that of Linux. A segment selector, which is an index into the segment descriptor table, used by the Linux application may point to a different region of memory on Windows or may not be valid at all. Any attempts to update the Windows descriptor tables may result in a conflict with the way Windows uses its resources. vPlay resolves these conflicts by intercepting and emulating the offending instructions within the Linux application's binary and the system calls that modify the descriptor tables.

vPlay uses two key-value table data structures, `segment_selector` and `selector_base`, to emulate the instructions with segment register operands. At any given time during replay, the `segment_selector` table maps a segment register to the selector it contains, and the `selector_base` table maps a selector to the base linear address of the segment that it points to. When an instruction which refers to one or more of its operands through a segment register is encountered during replay, vPlay computes the location of each operand in the flat address space using the formula, `(segment base + operand base + displacement + index*scale)`, where `segment base` is the base address of the segment and is obtained by joining the two tables on the selector. The remaining terms have instruction semantics and are obtained from the instruction. Using Pin's API, vPlay rewrites the original instruction such that the final linear address of the operand is used rather than referencing the segment register. The tables are initialized based on the descriptor table state captured in the partial checkpoint. As the application executes during replay, the `segment_selector` table is updated by intercepting the `mov` instructions that load the segment registers with selectors and the `segment_selector` table is updated by intercepting the `set_thread_area` and `modify_ldt` system calls, which provide the mapping between the segment base address and the selector.

5.5 Replay Across Hardware Architectures

Since vPlay directly captures binary instructions from the application's memory, it requires that the source and target hardware architectures be the same. In order to provide replay across different architectures and to provide a general support for virtual application address space, we have done a preliminary integration between vPlay and Qemu-user [5]. Qemu-user is an ancillary component of Qemu [5] which allows an application built for one architecture to be executed on a different architecture of the same operating system. It leverages a subset of Qemu's functionality to execute a user application on a virtual CPU without the need for a guest kernel running underneath the application. While this is currently work-in-progress, we have been able to replay simple Linux/x86 partial checkpoints on Linux/ppc hardware.

vPlay provides three extensions to Qemu-user that enables it to replay a partial checkpoint on a different operating system and architecture. First, a custom partial checkpoint loader which enables Qemu-user to read files in vPlay's partial checkpoint format and load them into memory. Qemu-user currently only supports binaries in ELF format. Second, a system call replay mechanism that replaces Qemu-user's existing system call translation component in case the loaded binary is a partial checkpoint. Third, the SoftMMU [5] functionality from Qemu is integrated such that application addresses are translated to addresses which are valid on the host, thereby effectively providing the replaying application with an independent virtual address space.

The replay operation is initiated by invoking Qemu-user with the partial checkpoint file as an argument. Qemu-user goes through the normal initialization process necessary to execute a Linux user application. It starts with basic initialization of the virtual CPU data structures followed by loading the application executable. If the executable is an ELF binary, Qemu-user invokes its ELF loader component which appropriately maps the binary into memory in user space. In case of virtual replay, it calls the partial checkpoint loader, which maps the pages as already outlined. The intermediate register state as represented in the partial checkpoint is then read and returned to Qemu-user for further initialization.

Qemu-user continues with the initialization by loading the registers returned by the partial checkpoint loader. It then sets up the Interrupt Descriptor Table (IDT) to handle exceptions (in particular, to trap system call interrupt), and the Global Descriptor Table (GDT) with descriptors referring to the 4 GB linear address space, as used by Linux. Any additional descriptor entries previously setup by the application at recording time are added to the respective descriptor tables by calling the emulated versions of `set_thread_area` and `modify_ldt` system calls implemented by Qemu-user which act on the virtual CPU. Finally, Qemu-user loads the segment registers with right selectors and starts the virtual CPU. At this point, the checkpointed application resumes execution at the beginning of the recording interval, as dictated by the instruction pointer value. The execution continues until a processor exception is raised.

If a system call exception is raised, Qemu-user normally forwards the system call to the underlying kernel after performing necessary translation. While replaying a partial checkpoint, however, Qemu-user calls vPlay's system call replay mechanism, which returns the previously recorded system call result back to the application. vPlay continues to monitor the application and ensures a deterministic replay until the execution reaches the end of the specified interval. In particular, system calls that add new memory regions to the address space are handled as described earlier, except that the target architecture specific versions of map and unmap system calls (`target_mmap` and `target_unmap`) provided by Qemu-user are used to map the pages. `set_thread_area` and `modify_ldt` are also handled similarly except that the Qemu versions of the respective system calls, which act on the virtual CPU are used.

Chapter 6

Evaluation

We have implemented vPlay as a kernel module on Linux 2.6.11 and 2.6.18 kernels and associated user-level tools that interact with the kernel module through an ioctl interface to perform the record and replay functions. The prototype can produce recordings of multi-threaded and multi-process Linux applications and replay them across the two kernels. We have also implemented the vPlay Container abstraction and a user-level replay tool for Windows based on Pin binary instrumentation [39], which currently only replays recordings of non-threaded Linux applications on Windows. Our unoptimized prototype works with unmodified applications without any library or base kernel modifications.

Using our prototype, we evaluate vPlay's effectiveness in (1) replaying recordings across environments differing in software installation, operating system and hardware, (2) capturing the root cause of various types of real software bugs on server and desktop applications, (3) minimizing runtime overhead and storage requirements of recording applications.

6.1 Platform Independence

Table 6.1 shows the software and hardware configuration of the record and replay environments and Table 6.2 lists the application workloads used for the experiments. Recording was done on a blade in an IBM HS20 eServer BladeCenter, each blade with dual 3.06 GHz Intel Xeon CPUs with hyperthreading, 2.5 GB RAM, a 40 GB local disk, and interconnected with a Gigabit Ethernet switch. Each blade was running the Debian 3.1 distribution and

Record-Replay Debian	**Replay Windows**	**Replay Gentoo**
IBM HS20 BladeCenter	Lenovo T61p Notebook	Apple MacBook Pro
3.06 Ghz Intel Xeon	2.4 GHz Intel Core 2 Duo	2.66 GHz Intel Core i7
Debian 3.1	Windows XP 2.16	VMware Fusion 3.0 / Mac OS X 10
Linux 2.6.11	2 GB RAM, 160 GB disk	Gentoo 1.12
2.5 GB RAM, 40 GB disk		Linux 2.6.18
		4 GB physical RAM, 512 MB to VM
		500 GB physical disk, 8 GB virtual disk

Table 6.1: Diversity of replay environments used for the experiments

the Linux 2.6.11 kernel and appears as a 4-CPU multiprocessor to the operating system. For server application workloads that also required clients, we ran the clients on another equivalent blade. The server applications were the Apache web server in both multi-process (`apache-p`) and multi-threaded (`apache-t`) configurations, the MySQL database server (`mysql`), and the Squid web cache proxy server (`squid`). `httperf-0.9` was used as the benchmark for the web servers and web proxy to generate 20,000 connection requests. The desktop applications were a media player (`mplayer`) and various compute and compression utilities (`gzip`, `bc`, and `ncomp`). The applications were all run with their default configurations. Each application workload was recorded in three different ways by taking partial checkpoints at three different intervals: 5, 10, and 15 seconds.

Replay was done in three environments, each significantly different in its hardware and software configuration from the others: (1) another blade in the BladeCenter running Debian 3.1, (2) a Lenovo T61p notebook with an Intel Core 2 Duo 2.4 GHz CPU, 2 GB of RAM, and a 160 GB disk running Windows XP 3.0, and (3) and a VMware virtual machine with 2 CPUs, 512 MB of RAM, and an 8 GB virtual disk running Linux 2.6.18 kernel on Linux Gentoo 1.12 distribution using VMware Fusion 3.0 on a MacBook Pro notebook with 2.66 GHz Intel Core i7 processor, 4 GB of RAM and 500 GB disk. None of the recorded application binaries were installed or available in any of the environments used for replay. The Windows and Gentoo replay environments had completely different software stacks from the Debian recording environment. Furthermore, the IBM blade center, the Lenovo notebook and the MacBook notebook with VMware virtual hardware represent diverse

Name	Description	Workload
`mysql`	MySQL database server	MySQL 3.23.56, 10 threads, run `sql-bench`
`apache-t`	Apache webserver multithread configuration	Apache 2.0.48, 57 threads, run `httperf 0.9`
`apache-p`	Apache webserver multiprocess configuration	Apache 2.0.54, run `httperf 0.9`
`squid`	Squid cache proxy server	Squid 2.3, run `httperf 0.9`
`bc`	Arbitrary precision expression evaluator	bc 1.06, compute π to 5000 decimal places
`gzip`	Gzip compression utility	Gzip 1.2.4, compress 200 MB `/dev/urandom` data
`ncomp`	Ncompress compression utility	Ncompress 4.2, compress 200 MB `/dev/urandom` data
`mplayer`	Mplayer media player	Mplayer 1.0rc2, play 10 MB 1080p HD video at 24 fps

Table 6.2: Application workloads

hardware configurations. Gentoo's 2.6.18 Linux kernel and Debian's 2.6.11 Linux kernel were very different with many new core features and code changes added between the two kernels. All of the application recordings were determinstically replayed correctly across all three different replay environments except for `mysql` and `apache-t`, which were replayed in the two different Linux environments but not in Windows due to lack of threading support.

6.2 Debugging with vPlay: Example Bug Scenarios

This section illustrates through several real life software bugs listed in Table 6.3, how vPlay is used to perform root cause analysis. Figure 6.1 shows a graphical screenshot of a typical vPlay debugging session with a partial checkpoint loaded within the GDB debugger. The application is paused at a breakpoint in the source code and the user is able to use standard debugging facilities. For example, mousing over a variable shows its content.

Most of the application bugs were taken from Bugbench [37], which is an assorted collection of real life bugs reported in popular open source software with an intent of providing a benchmark to measure the effectiveness of existing bug detection tools based on static and dynamic code checking. In addition to the common types of bugs in the BugBench suite, we also included bugs which only occur due to incompatible target environment. Those type of bugs were not present in the Bugbench suite and we obtained them from Internet forums where they were reported. The bug types include nondeterministic data race conditions, different types of memory corruption issues such as buffer overflow etc. We recorded each faulty application while the bug is triggered in the Debian environment. The applications were run by applying workload with specific characteristics such that the bug is triggered. In some cases, the experiment had to be repeated many times before the bug manifested. The recordings of the bugs were then analyzed by deterministically replaying them within the GDB debugger using vPlay in the Gentoo environment. We also reproduced all the bugs in the Debian environment and all the bugs except `mysql` and `apache-t` in the Windows environment.

Name	**Application**	**Bug**
`mysql`	MySQL 3.23.56	Data race
`apache-t`	Apache 2.0.48	Atomicity violation
`apache-p`	Apache 2.0.54	Library mismatch
`squid`	Squid 2.3	Heap overflow
`bc`	bc 1.06	Heap overflow
`gzip`	Gzip 1.2.4	Global buffer overflow
`ncomp`	Ncompress 4.2.4	Stack smash
`mplayer`	Mplayer 1.0rc2	Device incompatibility

Table 6.3: Example Software Bugs

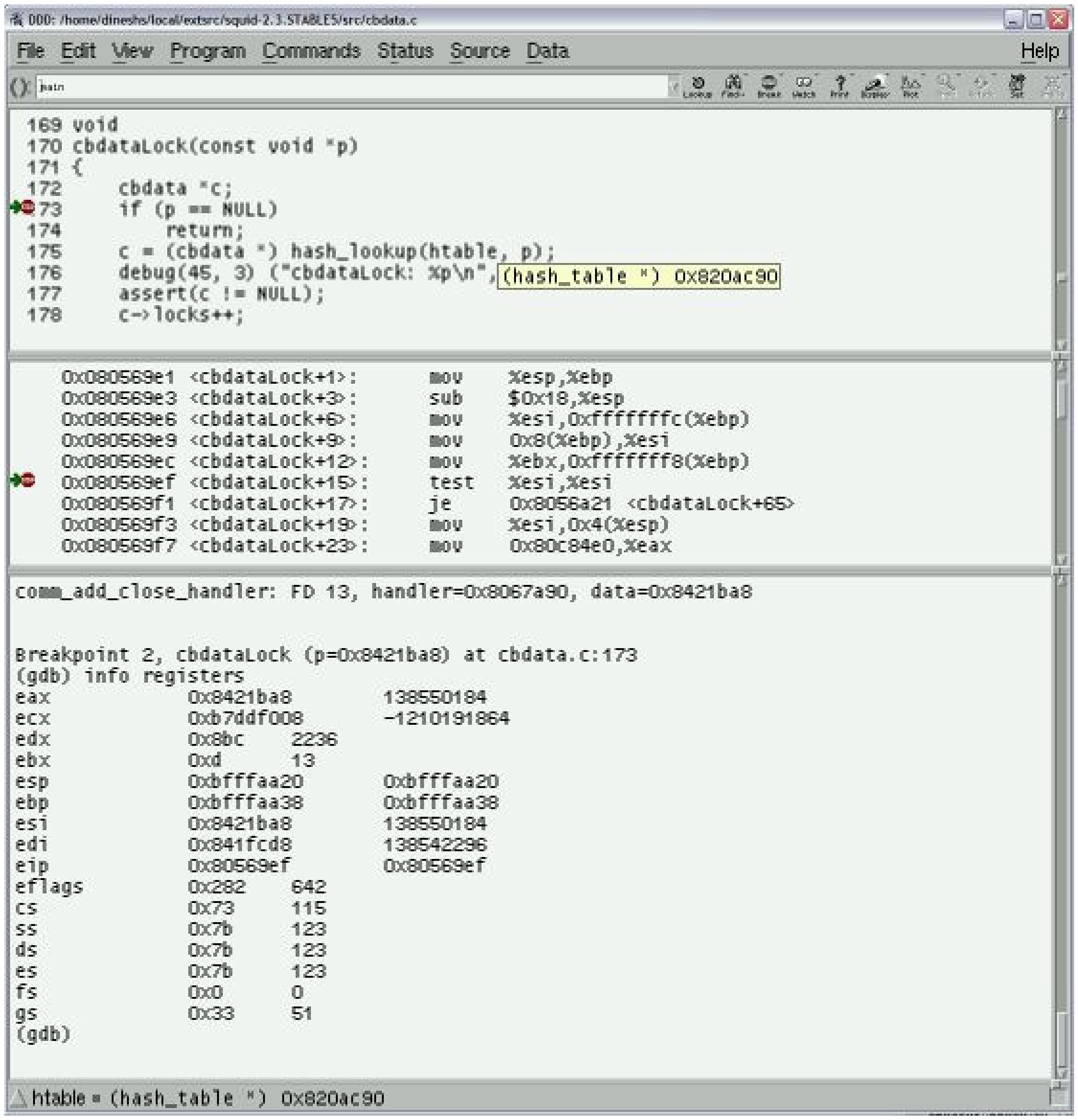
A primary use case of vPlay was in debugging itself during the development process. While vPlay captures and exposes the buggy execution of an application, a recording representing a bug-free execution serves as a test case for vPlay itself. A part of vPlay's regression test suite consists of a set of recordings captured from bug-free intervals of application's execution. While replaying a recording, if vPlay notices a discrepancy between the application's actual execution and the events recorded in the log, it points to an internal inconsistency. This was also the mechanism we used to identify various application interfaces which needed to be emulated and the state of resources to be captured.

In our experiments with real software bugs, we found that vPlay is able to guide the debugging process towards the root cause of the failure where most conventional debugging methods fall short. In particular, vPlay was able to aid in debugging the following type of bug scenarios:

1. **Bug depends on specific library version:** Applications often rely on libraries and code components supplied by third parties and expect them to provide specific programming interfaces for them to be compatible with the application. With growing complexity of applications, it is challenging to accurately determine the dependency graph among various application components. Any incompatibilities between application components surface as failures or faulty behavior. Such incompatibilities could arise due to outdated library versions, incomplete or incorrect installation, overwriting of common libraries by co-deployed applications that need a later version of the libraries etc. Reproducing and diagnosing such failures can be difficult due to the large search space of possible errors. Since vPlay captures the actual code pages in the offending libraries and uses them during replay, the behavior is guaranteed to be reproduced. In addition, the information regarding the provenance of the data included in the recording directly points to the problem source.

2. **Bug triggering data on disk:** Sometimes data on disk such as an ill-formatted entry in a large data file, or a particular script in a large source code repository could be the source of the bug. In such cases, identifying the specific piece of bug-triggering data would be crucial to problem diagnosis. Existing record-replay tools based on system call interposition may capture the data accessed through system calls but any data directly accessed from the memory regions would be missed. vPlay narrows down the problem by eliminating potentially large data sets in the source environment from consideration and focuses analysis to the small amount of data captured as a part of the recording.

3. **Nondeterministic race conditions:** Shared memory race conditions are notori-



```
169 void
170 cbdataLock(const void *p)
171 {
172     cbdata *c;
173     if (p == NULL)
174         return;
175     c = (cbdata *) hash_lookup(htable, p);
176     debug(45, 3) ("cbdataLock: %p\n", (hash_table *) 0x820ac90
177     assert(c != NULL);
178     c->locks++;
```

```
    0x080569e1 <cbdataLock+1>:     mov    %esp,%ebp
    0x080569e3 <cbdataLock+3>:     sub    $0x18,%esp
    0x080569e6 <cbdataLock+6>:     mov    %esi,0xfffffffc(%ebp)
    0x080569e9 <cbdataLock+9>:     mov    0x8(%ebp),%esi
    0x080569ec <cbdataLock+12>:    mov    %ebx,0xfffffff8(%ebp)
    0x080569ef <cbdataLock+15>:    test   %esi,%esi
    0x080569f1 <cbdataLock+17>:    je     0x8056a21 <cbdataLock+65>
    0x080569f3 <cbdataLock+19>:    mov    %esi,0x4(%esp)
    0x080569f7 <cbdataLock+23>:    mov    0x80c84e0,%eax
```

```
comm_add_close_handler: FD 13, handler=0x8067a90, data=0x8421ba8


Breakpoint 2, cbdataLock (p=0x8421ba8) at cbdata.c:173
(gdb) info registers
eax            0x8421ba8        138550184
ecx            0xb7ddf008       -1210191864
edx            0x8bc    2236
ebx            0xd      13
esp            0xbfffaa20       0xbfffaa20
ebp            0xbfffaa38       0xbfffaa38
esi            0x8421ba8        138550184
edi            0x841fcd8        138542296
eip            0x80569ef        0x80569ef
eflags         0x282    642
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

```
htable = (hash_table *) 0x820ac90
```

Figure 6.1: Interactive debugging session within vPlay integrated debugger showing assembly instructions within the partial checkpoint and corresponding source lines

ously hard to reproduce and debug. Bugs that depend on thread scheduling could take a large number of iterations to manifest. Even so, an occurrence of the faulty behavior may only prove that the bug does exist, but it does not necessarily help with root cause analysis, which may require many repeated occurrences. vPlay's software-level support for record-replay of threaded applications captures the specific interleaving of accesses to shared memory which cause the faulty behavior. Once captured, the same interleaving and the buggy behavior can be deterministically reproduced every time.

4. **Offensive client requests and external inputs:** Certain bugs manifest only under specific workloads or when specific types of external inputs are presented to the application. Identifying the source data responsible for the failure is key to addressing those failures. vPlay captures the required application inputs and replays it back to the application in a way which is consistent with the application semantics, even if the target system itself does not support the corresponding interface.

5. **Lost program context:** Conventional debugging methods, such as core dump analysis, which do not preserve historical execution context are often ineffective for many difficult cases of memory corruption. The final state of the application may bear no link to the original root cause of the problem. Examining the crash site on a post-facto basis may not provide any leads. In such cases, the relevant execution context is either completely lost due to program counter itself being corrupted or even if the program is in a consistent state, it's final state may be seemingly unrelated to the root cause. Replaying the execution steps prior to the failure with the intermediate state of program memory pages preserved, exposes the program control flow that lead to the corrupt end state of the program.

In general, vPlay's approach of capturing software bugs into a self-contained recording reduces the overall turn around time of problem determination in two ways. First, vPlay helps reproduce and diagnose otherwise hard-to-reproduce and complex bug scenarios. Second, vPlay also streamlines the bug fixing process for even simple bugs which, according to anecdotal evidence, represent most of the support tickets received by large technical support organizations. In addition to reducing the time taken to fix complex bugs, the vPlay model also expedites the resolution of simpler issues by containing debugging to a streamlined and repeatable process.

In our experiments, once a recording of the bug occurrence was captured, vPlay was able to deterministically replay the bug every time, even in a different environment, and was useful to diagnose the root cause of each bug. For example, for the `mysql` and `apache-t` non-deterministic data race bugs, vPlay correctly captured the specific interleaving of shared memory accesses required to reproduce the bug. vPlay was able to capture all data required to reproduce these bugs with partial checkpoint and log sizes orders of magnitude smaller than the application's memory footprint. In general, vPlay captured the bug-triggering conditions and input required to reproduce all bugs. For instance, the malformed client request which caused `squid` to fail and the relevant code snippet from the input program that triggered a heap overflow in `bc` were part of the log recorded by vPlay. In case of `apache-p` and `mplayer`, the bugs occurred due to incompatibility with the target environment. For `apache-p`, one of the processes would silently exit when it notices unexpected behavior from a function in one of the libraries it uses due to an incompatible version. Since vPlay captured the code page in the library where the offending function resides, vPlay was able to reproduce the faulty behavior even on the system where the right version of the library was installed. Other record-replay tools which only record at the system call interface would not be able to capture these types of bugs. Similarly, vPlay correctly captured the root cause of the problem for `mplayer`, which failed due to an incompatible audio device at the system on which it was run.

6.2.1 `ncompress`: Stack Smash

`ncompress` is a commonly used compression utility based on LZW algorithm. In function `comprexx` of its code, a local array is initialized without checking the array bounds, which could result in stack corruption in case of an overflow.

```
void
comprexx(fileptr)
        char    **fileptr;
        {
                int             fdin;
                int             fdout;
                char    tempname[MAXPATHLEN];

                strcpy(tempname,*fileptr);
                ...
        }
```

Running the program causes a segmentation violation and examining the `core` file in GDB shows that the instruction pointer doesn't point to a valid region of memory. In fact, all original register contents were erased due to wild stores over the stack.

```
$ gdb core core
Program terminated with signal 11, Segmentation fault.
#0  0x61616161 in ?? ()
gdb $ bt
#0  0x61616161 in ?? ()
gdb $ info registers
eax            0x0      0x0
ecx            0x1000   0x1000
edx            0x0      0x0
ebx            0x61616161       0x61616161
esp            0xbfffeb90       0xbfffeb90
ebp            0xbfffebb8       0xbfffebb8
esi            0x61616161       0x61616161
edi            0x61616161       0x61616161
eip            0x61616161       0x61616161
eflags         0x10282  [ SF IF RF ]
cs             0x73     0x73
ss             0x7b     0x7b
ds             0x7b     0x7b
es             0x7b     0x7b
fs             0x0      0x0
gs             0x33     0x33
```

This is a common occurrence in debugging memory corruptions, where the program ends up in corrupt state with `NULL` or invalid program counter. The program could have reached this end state from anywhere. Since the stack overflow erased the stack contents, the core file does not contain the necessary context to debug the crash. The reported symptom or the core dump itself do not provide any lead.

We captured the same bug as a partial checkpoint by running the program within vPlay. The partial checkpoint not only contained the relevant code context to debug the problem but also the specific code pages such that it was not necessary to reinstall the application at the target. The application was revived in a consistent state to a point prior to the crash with the program counter pointing within the caller of `comprexx` function. Stepping through the source got the program to line 907 where it crashed again with a segmentation violation. We replayed the partial checkpoint for a second time to examine the `fileptr` variable which contained the bug triggering input. Obviously, its contents were too big for the local array into which the program was trying to copy.

6.2.2 `gzip`: Global Buffer Corruption

```
local int get_istat(iname, sbuf)
    char *iname;
    struct stat *sbuf;
{
  int ilen;  /* strlen(ifname) */
  static char *suffixes[] = {z_suffix, ".gz", ".z", "-z", ".Z", NULL};
  char **suf = suffixes;
  char *s;
#ifdef NO_MULTIPLE_DOTS
  char *dot; /* pointer to ifname extension, or NULL */
#endif

  strcpy(ifname, iname);

```

When invoked with the bug triggering input, the `gzip` program crashes with a core dump due to an overflow of an array in the global data segment. Examining the core file in GDB shows that the exception occurred in `free()` libc function. Even though the program counter points to a valid function, there is no direct correlation between the root cause of the problem and its manifestation as a segmentation violation in `free()`.

vPlay correctly captures the program execution leading to its failure state. With no descriptive account or additional information from the user, the partial checkpoint directly provides all data necessary to reproduce and fix the problem. The program was resumed to a point before the crash, and setting a watchpoint at the corrupt memory clearly pointed the root cause to be the `strcpy()` function at line 829. Variable `iname` contained the bug triggering input string.

6.2.3 `squid`: Heap Overflow

`squid-2.3` contains a security flaw [49] which results in denial of service when triggered by a specially crafted input FTP request. Each offending request causes one of the server processes to silently crash with a segmentation violation, eventually rendering the server unusable. Running a GDB backtrace on the core file just lists a long series of meaningless values on the stack, and the program counter points to a valid but zero initialized portion of memory.

These types of failures are particularly difficult to debug without precisely knowing the type and characteristics of the workload presented to the application at the time of the failure. A server like `squid` may receive thousands of complex requests and pin-pointing exactly which set of events triggered the failure can be difficult. Anecdotal references indicate that the server failures are typically reproduced using custom-built load generators which fabricate artificial workload based on the server logs. However, the logs produced by the server may not have adequate information to accurately generate the bug-triggering workload and reproduce the failure.

vPlay captured the offending client request along with other context within a partial checkpoint measuring less than 1 MB in size. It was used to repeatedly replay the crash even on a different system, with all interactions between the crashed server process and external clients simulated within the virtual vPlay environment.

6.2.4 `bc`: Heap Overflow

`bc` is a standard unix tool with a bug which causes its parser to crash when interpreting certain types of bc scripts. The bug was introduced due to a typo in the code, which uses the variable `v_count` in place of the variable `a_count`. In the `bc` code snippet below, the array variable `arrays` is initialized with `a_count` number of elements but the loop on line 171 iterates over the array `v_count` number of times, which represents the number of variables in the `bc` program.

```
165   /* Increment by a fixed amount and allocate. */
166   a_count += STORE_INCR;
167   arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var_array *));
168   a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170   /* Copy the old arrays. */
171   for (indx = 1; indx < old_count; indx++)
172     arrays[indx] = old_ary[indx];
173
174
175   /* Initialize the new elements. */
176   for (; indx < v_count; indx++)
177     arrays[indx] = NULL;
```

The bug was triggered with a `bc` input program containing several dozens of variables and arrays, all declared within a function, with variable count larger than the array count. The program was syntactically correct and otherwise legitimate, but running it in `bc` caused it to crash with a segmentation violation.

Analyzing the core dump would not yield any pointers to the root cause. The key to debugging this type of bugs is the availability of right program inputs. In this case, the input happens to be in a disk file containing a piece of offending `bc` script. In general, applications have access to large amounts of disk data and identifying the portion of that data which triggers the bug can be challenging. Existing debugging tools based on system call level record-replay may capture the program inputs accessed through the system call interface but the applications could directly read parts of disk files through memory mapped regions. vPlay's memory tracking approach is able to record every piece of required data, including that read directly from memory.

vPlay provided the required debugging context by means of a self contained partial checkpoint. Through a combination of single stepping and watch points, it was clear that the root cause of the problem was the `for` loop on line 176.

6.2.5 `mysql-t`: Race Condition

This is a rare race condition where the state of the database does not agree with the state of the recovery log. Deletion of all rows in a table is optimized by creation of a new table. However, the optimized code path enters the transaction into Mysql's recovery log (`bin_log`) after releasing all the locks. If the server receives another request after the creation of the new table but before it is logged, the recovery log would be in an inconsistent state with respect to the state of the table.

The bug was triggered by issuing the following statements concurrently from two clients:

```
Client 1: mysql -u root -D test -e 'DELETE FROM tab'
Client 2: mysql -u root -D test -e 'INSERT INTO tab values(1)'
```

When the bug was triggered, examining the log file showed that the delete and insert entries were out of order with respect to the order of their original execution. vPlay correctly captured the execution order of the two threads with respect to the lock that protects the recovery log. In each replayed execution, the thread doing table delete was made to wait until the insert statement has been logged, even though the delete thread began executing first.

6.2.6 `apache-t`: Atomicity Violation

Threaded version of `apache-2.0.48` exhibits a race condition due to concurrent writes to shared memory. In the code below, a structure `buf` allocated on heap is concurrently accessed by multiple apache threads as they process requests and add messages to the log. The log message is added to `buf->outbuf` and the current pointer within that buffer is maintained by `buf->outcnt`. However, writing to the buffer and updating the pointer are not protected by a critical section, and the entries logged by concurrent threads could overwrite each other.

```
1327 static apr_status_t ap_buffered_log_writer(request_rec *r,
           ...

1358       for (i = 0, s = &buf->outbuf[buf->outcnt]; i < nelts; ++i) {
      memcpy(s, strs[i], strl[i]);
      s += strl[i];
    }
    buf->outcnt += len;
    ...
}
```

The bug was triggered by two concurrent `httperf` clients, each making 10000 requests for two different files. Given the rarity of the bug occurrence, the buggy control path had to be exercised several thousands of times, which as a side effect also resulted in a significant amount of log (about 2 MB of log and 1 MB of partial checkpoint per thread group, per second). Examining the log file showed several places where the log entries were jumbled up. The entire experiment was run within vPlay to record the partial checkpoint.

Once the partial checkpoint was captured, vPlay was able to reproduce the pathological interleaving of threads across log entries. Regardless of the new ordering of threads enforced by scheduler, vPlay produced the correct interleaving of threads with respect to the shared log buffer such that the same garbled log message text was produced on each replay.

6.2.7 `apache-p`: Library Mismatch

The bug was triggered by prematurely interrupting the install script which upgrades `apache-2.0.54` to `apache-2.2.15`. The interruption caused one of the core apache libraries (`libapr utils`) to be left in its old version, whereas another component (`libapr`), that depends on `libaprutils` to be upgraded. The server starts correctly but the processes begin to silently exit when `httperf` workload is placed. No exception is received by the processes and no log entries related to the errors are generated. Errors such as this occur for a variety of reasons and an unsuspecting programmer may be misled into trying to reproduce the failure and validating the code functions in the programmer's environment.

Recording the behavior with vPlay captured the partial checkpoint for the failed process. The partial checkpoint contained the binary instructions relating to the un-upgraded version of `libaprutils` library, along with provenance metadata indicating its .so name and version. Replaying the partial checkpoint within the debugger showed the code that the failed process executed and how the interaction between the base `httpd` code and the code in the library were not compatible, causing it to prematurely exit.

6.2.8 `mplayer`: Device Incompatibility

When an HD video file is opened and played by `mplayer`, the video appears in its window but no sound is produced. There could be many reasons for such a behavior given the complexity and number of hardware devices, audio/video formats etc. Regardless of the reason, this type of problems are annoying and it is not trivial to determine the cause even for an experienced user.

A simple recording of the faulty run with vPlay captured all required data to inform the developer of the reason for the lack of sound. The format of the media was supported by one of the plugins mapped by `mplayer`, but a failed `ioctl` call indicates that the target device is configured incorrectly and incompatible.

6.3 Performance and Storage Overhead

Name	**Time**	**Memory**	**Partial**	**Log**
`mysql`	105 s	121 MB	538 KB	29 KB
`apache-t`	57 s	221 MB	1305 KB	2284 KB
`apache-p`	59 s	4188 KB	935 KB	2570 B
`squid`	82 s	7192 KB	991 KB	4 KB
`bc`	55 s	2172 KB	349 KB	2714 B
`gzip`	68 s	1820 KB	321 KB	1341 B
`ncomp`	82 s	1440 KB	293 KB	1229 B
`mplayer`	40 s	44 MB	1393 KB	9513 KB

Table 6.4: Application workloads used for performance characterization

We have evaluated the runtime performance and storage overhead of vPlay using the application workloads listed in Table 6.2. Table 6.4 lists the execution time for each application workload when run natively on Linux without vPlay, and Figure 6.2 shows the normalized runtime overhead of recording the application workloads compared to native execution. For the 5 second intervals, the recording overhead was under 3% for all workloads except for `squid` and `mysql`, where the overhead was 9% and 17%, respectively.

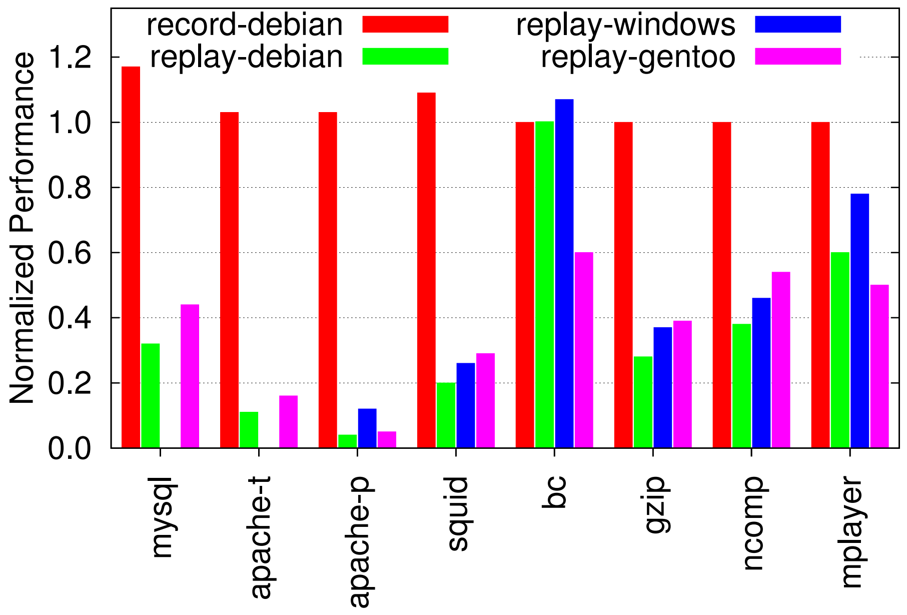


Figure 6.2: Normalized record-replay performance

Figure 6.2 also shows the speedup of replay on Linux and Windows. Replay was generally faster than recording, several times faster in some cases. Two factors contribute to replay speedup: omitted in-kernel work due to system calls partially or entirely skipped (e.g. network output), and time compressed due to skipped waiting at replay (e.g. timer expiration). `bc` did not show any speedup because it is a compute-bound workload which performed few system calls. Speedups on Windows were smaller due to the additional overhead of binary instrumentation and emulation required to replay on Windows. The binary instrumentation overhead was less for longer replay intervals as Pin's overhead of creating the initial instruction cache for emulation is amortized over the replay interval. The difference in replay performance on Gentoo and Debian systems was due to the difference in the underlying hardware used in the two systems. Even though the Mac system had a faster processor compared to the blades, the VMware virtual environment added additional overhead. Figure 6.3 shows recording overhead as a function of the length of the recording interval for the application scenarios. Overhead was generally smaller with longer recording intervals, where checkpointing was done less often.

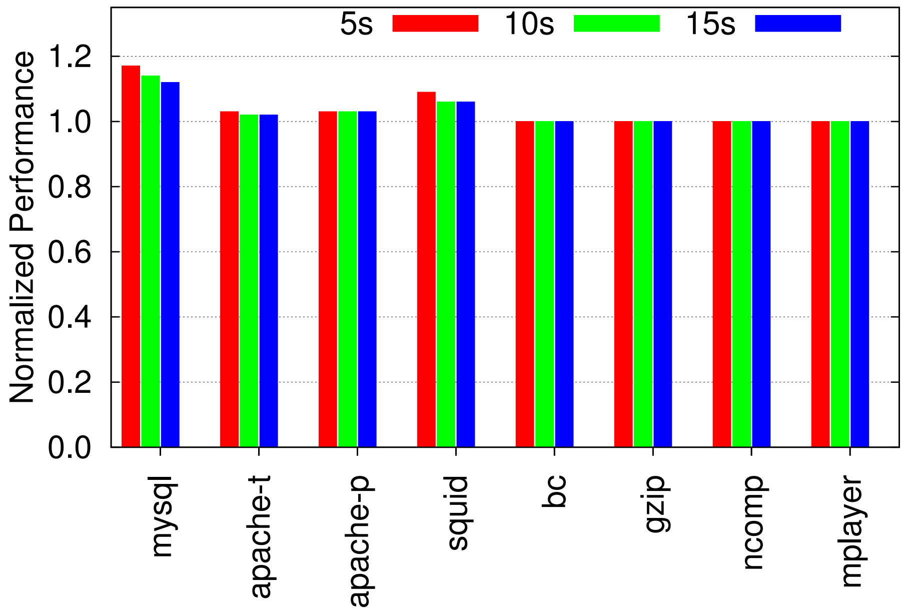


Figure 6.3: Recording performance vs recording interval

Figure 6.4 shows a measure of partial checkpoint latency, the average time it takes to atomically finish recording one interval and start recording a subsequent recording interval while doing a periodic recording of the applications. It includes the time taken for the application threads to reach the synchronization barrier so that a consistent initial state of the application for the partial checkpoint can be recorded. The application is not completely stopped during this time. Some of the application threads may still be running application code while others reach the barrier. The barrier is created in the kernel when checkpoint request arrives and each application thread reaches the barrier the next time it enters the kernel. Once all threads reach the barrier, the rest of the processing is done. The latency is less than a few hundred milliseconds in all cases. The average latency was the same for the 5, 10, and 15 second recording intervals.

We saved the last three partial checkpoints and their associated logs for each application and characterized their size and composition. For `mplayer`, only the last two partial checkpoints and logs were saved for the 15 second recording interval due to its relatively

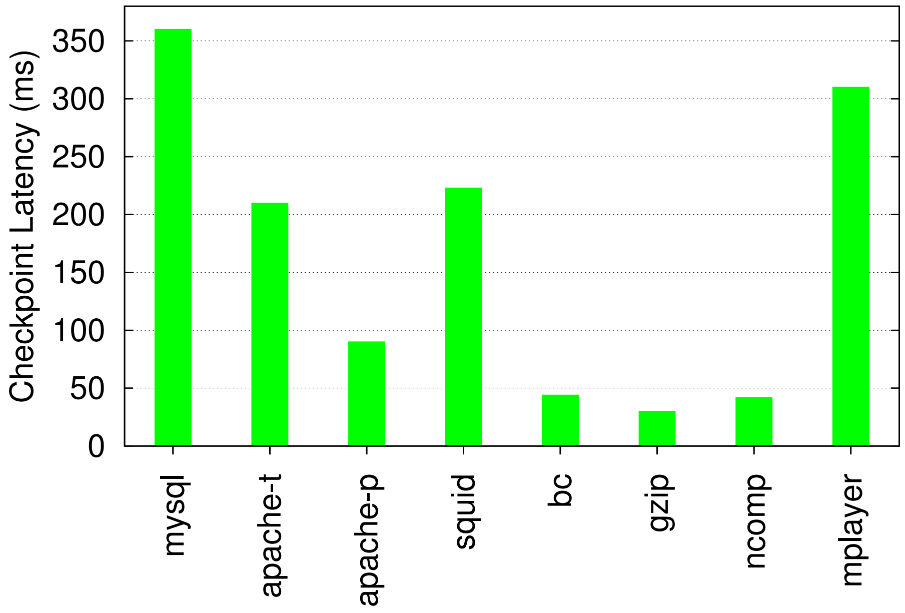


Figure 6.4: Checkpoint latency

short execution time. We only considered complete intervals, so if 5 second recording intervals were used and an application had a partial checkpoint at the end of its execution accounting for the last 2 seconds of execution, that partial checkpoint was not included in this characterization.

Figure 6.5 shows the average total size of partial checkpoints across all processes of each workload for 5, 10 and 15 second recording intervals. Partial checkpoint sizes are modest in all cases, no more than roughly 5 MB for even the longest recording intervals. Most of an application's memory pages are untouched during any particular interval of execution. For example, the largest partial checkpoint was roughly 5 MB for `mysql`, which had a virtual memory footprint of well over 100 MB. Figure 6.5 also shows the size of the partial checkpoints when compressed using `lzma`, as denoted by the patterned bars. In addition to the fact that the partial checkpoint data compressed well, the high compression ratios indicated were also due to our unoptimized prototype which would end up storing duplicate code pages with the same content for multi-process applications. While the cost of taking regular full checkpoints is usually highly correlated with checkpoint size due to the large

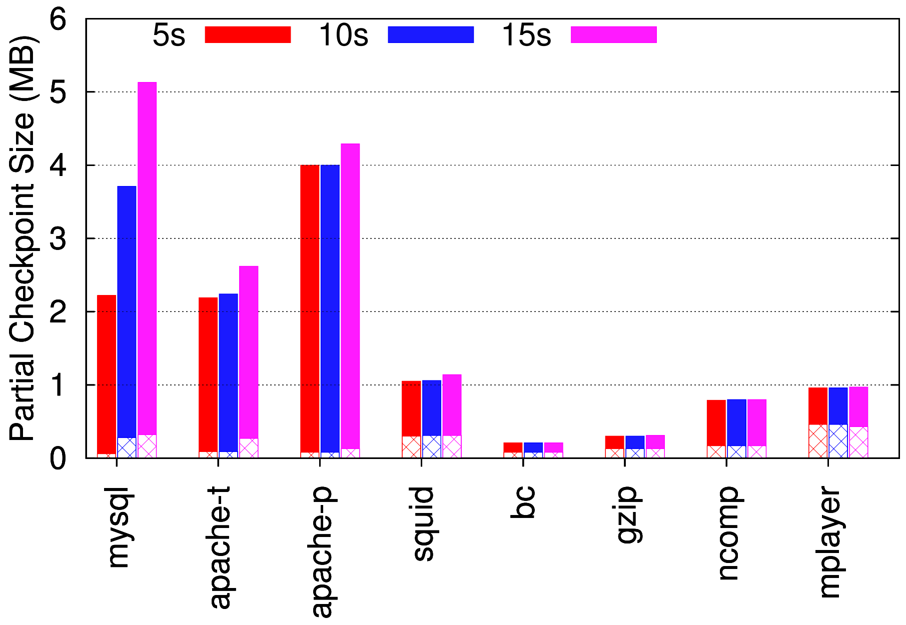


Figure 6.5: Partial checkpoint size

amount of memory state that needs to be saved, Figures 6.4 and 6.5 show that partial checkpoint latency is not correlated with partial checkpoint size because the sizes are quite small.

Figure 6.6 shows the total size of logs generated by all processes of each application for 5, 10 and 15 second recording intervals. `mysql` had the most log data due to the high density of system call events carrying input data presented by `sql-bench`. For a 5 second recording interval, the log size was 59 MB. While this is significant storage overhead, the log does not accumulate over time. Even though vPlay continuously records the application, it only stores the most recent execution history within a buffer of fixed size. `bc` was mostly compute bound and had the least log data, less than 1 KB, which is not visible in Figure 6.6. Figure 6.6 also shows the compressed log sizes, as denoted by overlaid pattern bars. The logs of most workloads compressed well, except for `gzip`, `ncomp` and `mplayer`, for which negligible compression was obtained and hence the compressed values are not shown. The log of `gzip` and `ncomp` mostly contained the 200 MB of random data, which does not compress well. The log of `mplayer` was dominated by the compressed video file, which also

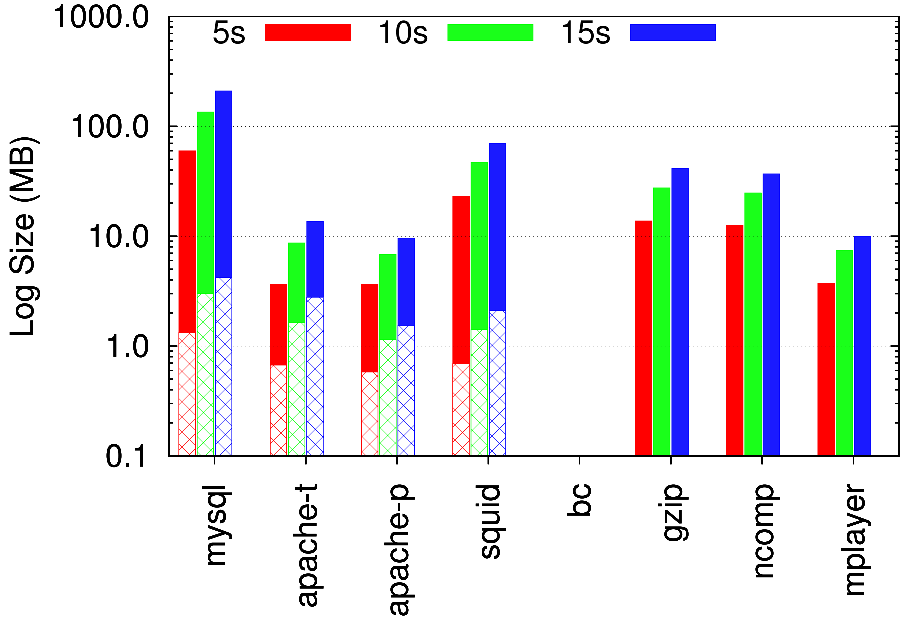


Figure 6.6: Log size

does not compress well.

Figure 6.7 shows the composition of each application's log. The three bars shown for each application correspond from left to right to the 5, 10, and 15 second recording intervals, respectively. The log data is classified into four categories: sys is system call records and integer return values, output is the data returned from system calls, mmap is pages mapped during the recording interval, and shm are events corresponding to page ownership management of shared memory. In most cases, the log was dominated by output data which is returned to the replayed application through system calls. One of the primary goals of vPlay is to decouple the application from its source environment and vPlay meets this goal, in part by logging more data than other record-replay systems that require an identical replay environment. `bc` produced a small log, mostly containing the system call records. `apache-t` shows many page ownership management events in its log since it is a multi-threaded application with many threads. `mysql` has fewer threads and less page ownership management events. Log data due to memory mapped pages was generally small relative to other constituents of the log because most of the memory mappings occurred

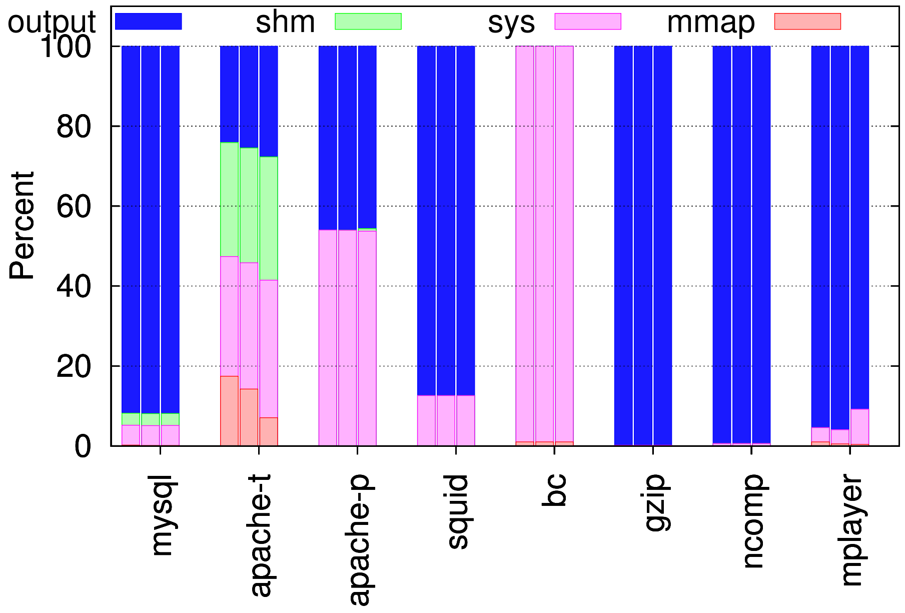


Figure 6.7: Log composition

at the beginning of the applications and the logs are for the last few complete intervals of application execution.

Table 6.4 shows the partial checkpoint and log sizes for vPlay to capture and reproduce each bug. We also measured the virtual memory footprint of each application as reported by `top` command to provide a rough measure of the amount of state required to run it. In practice, applications typically require far more data than the content of their virtual memory. They also indirectly rely on the state represented by their environment and the operating system. In all cases, the size of the sum of the partial checkpoint and log is much less than the virtual memory footprint of each application. While the log generated by vPlay contains the complete set of events and data necessary to reproduce a bug, it represents all application activity occurring within that interval. Within the same recording interval where the `mplayer` bug was triggered, it was also mapping various codec libraries and accessing their pages to initialize them. This additional noise accounts for the large log size produced by `mplayer` bug. Note that the partial checkpoints and logs required to capture the bugs is in general much less than what was required to record the more

Application	**Bug**	**Error Propagation Distance**
MySQL 3.23.56	Data race	–
Apache 2.0.48	Atomicity violation	–
Apache 2.0.54	Library mismatch	1581
Squid 2.3	Heap overflow	2860
bc 1.06	Heap overflow	2005
Gzip 1.2.4	Global buffer overflow	261
Ncompress 4.2.4	Stack smash	87
Mplayer 1.0rc2	Device incompatibility	18852

Table 6.5: Error propagation distance in terms of number of instructions

resource-intensive application workloads shown in Figures 6.5 and 6.6.

Figure 6.8 shows the error propagation distance for each bug listed in Table 6.5. To measure error propagation distance, we instrumented vPlay to log the value of the time stamp counter along with each `event_record`, and calculated the time between two closest events that encompass the root cause of the bug and the appearance of the bug symptom. In all cases, the observed value was less than half a second, validating vPlay's assumption of short error propagation distances and demonstrating that a modest recording interval of 5 seconds as used in our experiments is more than sufficient to capture and reproduce the bugs. Table 6.5 also shows the error propagation distances in terms of the number of instructions executed by the application between the root cause of the problem and the failure. The instruction count is omitted for the multithreaded applications where the root cause and the failure occur in different threads. We observe that the application quickly reached the failure point after encountering a bug, within a window of several thousand instructions in every case.

6.4 Evaluation of the User Space Partial Checkpointing Implementation

We have also experimented with a vPlay-user prototype, which generates the recordings of Linux applications and replays them on other Linux distributions and Windows. The

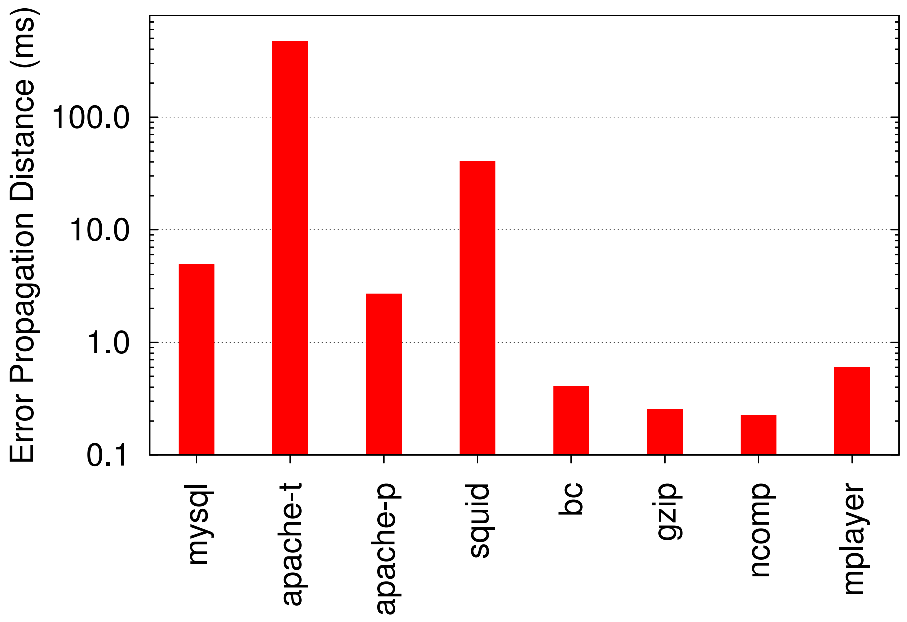


Figure 6.8: Error propagation distance

prototype was implemented mostly in user space based on the two kernel extensions discussed in Section 3.3. The user-space prototype did not support shared memory or signals. The experimental setup consists of two identical machines, each with an Intel Core 2 Duo 2.4GHz processor and 2 GB of RAM. One of them is installed with Ubuntu 8.10, and the other is installed with Windows XP version 2.16 and Fedora 11, in two bootable partitions of its hard disk. Ubuntu system and Fedora systems run Linux-2.6.26 kernel with the vPlay-user kernel extension.

Application	**Description**
`apache-2.2.11`	Web server (`httperf` workload)
`squid-2.3`	Cache proxy server (`httperf` workload)
`gzip-1.2.4`	Uncompress a 64MB compressed file
`bcbench-1.06`	Calculate pi to 2000 places
`abiword-2.6.6`	Word processor
`gv-3.5.8`	Document viewer

Table 6.6: Application scenarios used for experiments with user space implementation

The application scenarios evaluated in the experiments are listed in Table 6.6 and the application environment and configuration used were generally different from the benchmarks used for the kernel based prototype. Recording was performed on the Ubuntu machine with each application continuously recorded while the measurements were taken. At any point of time, seven most recent partial checkpoints were maintained in memory. For `apache` and `squid`, `httperf` benchmark [44] was used to generate a workload of 200 connections per second and the resulting connection response time was measured, `gzip` was recorded while it was uncompressing a 64 MB compressed file that decompresses to a 285 MB clear text file, `bc` was calculating the value of pi to 2000 decimal places, and `abiword` and `gv` were each displaying a document while they were being monitored and recorded. In each application scenario, a fabricated failure event was triggered during the benchmark by sending the application a `SIGSEGV` signal, so that vPlay-user would write-out the last seven partial checkpoints. The resulting partial checkpoints were then replayed individually on Fedora and Windows systems. The experiment was repeated six times with varying lengths of recording intervals from 125 ms to 4000 ms on a log scale. We removed the applications used in the experiment from the Fedora system, and so the replay exclusively relied on the checkpointed memory and binary pages.

Figure 6.9 shows percentage performance overhead compared to native execution without recording for four applications: `apache`, `squid`, `gzip` and `bcbench` [73] at recording intervals varying from 125 ms to 4000 ms. Squid showed the highest worst case overhead of 15% at 125 ms recording interval. The overhead was 13% for `apache`, 6% for `gzip` and 5.5% for `bc` at the same interval length. In all cases, the overhead became unnoticeable at sufficiently long recording intervals.

Figure 6.10 shows the storage space occupied by different constituents of a recording representing a one second recording interval. It consists of three parts - the amount of memory read and dirtied by the application, and the amount of data returned via system calls. In most cases, the memory pages dominate the partial checkpoints. A significant portion of this overhead originates in the file system data captured by vPlay-user which enables the applications to be replayed without an identical install base at the target site. The large system call log shown by `gzip` accounts for the contents of the compressed file

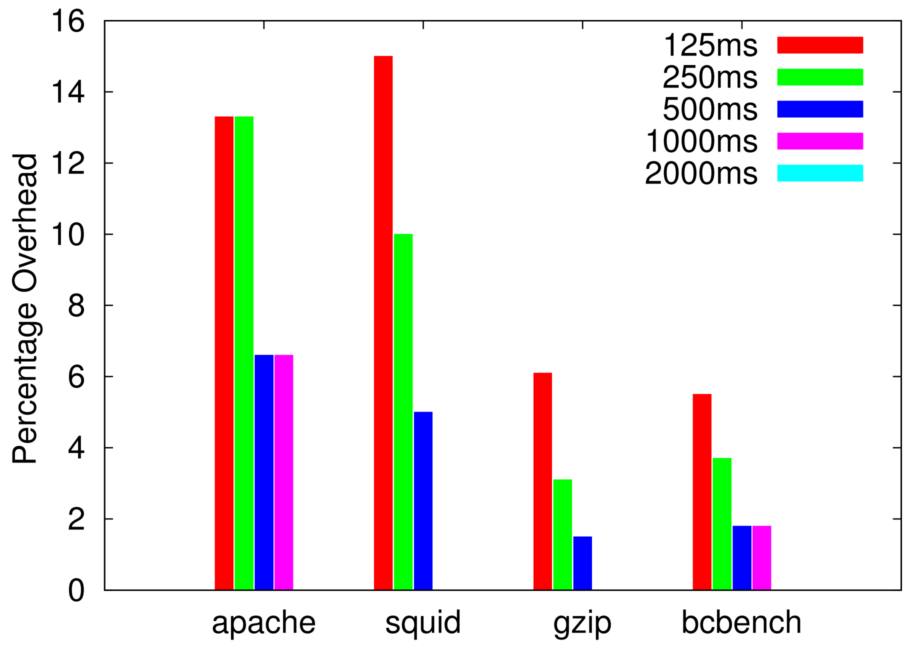


Figure 6.9: Recording performance

read through the `read` system calls. `bcbench` on the other hand is mostly CPU-bound and its partial checkpoints contain little system call related data. Figure 6.11 shows the rate at which the total size of a partial checkpoint grows with the length of the recording interval.

Figure 6.12 shows the time taken to perform `start` and `stop` operations. In general, `start` operation is relatively light and the time it takes is dominated by the creation of the shadow process. `stop` is heavier because it has to scan the page tables of the application to determine the pages accessed in the last recording interval. `gzip` behaves anomalously because of the large system call data held by vPlay-user agent in its address space. When the shadow process is created during `start`, the `clone` system call copies the page table entries of the entire address space of the process including those within the vPlay-user agent memory region. However, vPlay-user doesn't scan its own memory region when it checks the accessed and dirty bits, making `stop` relatively lighter.

Figures 6.13 and 6.14 show the times taken by load and replay phases of the virtual replay operation respectively on Fedora and Windows systems. As expected, the load and replay times are greater on Windows than on Linux. vPlay-user uses its native instrumentation

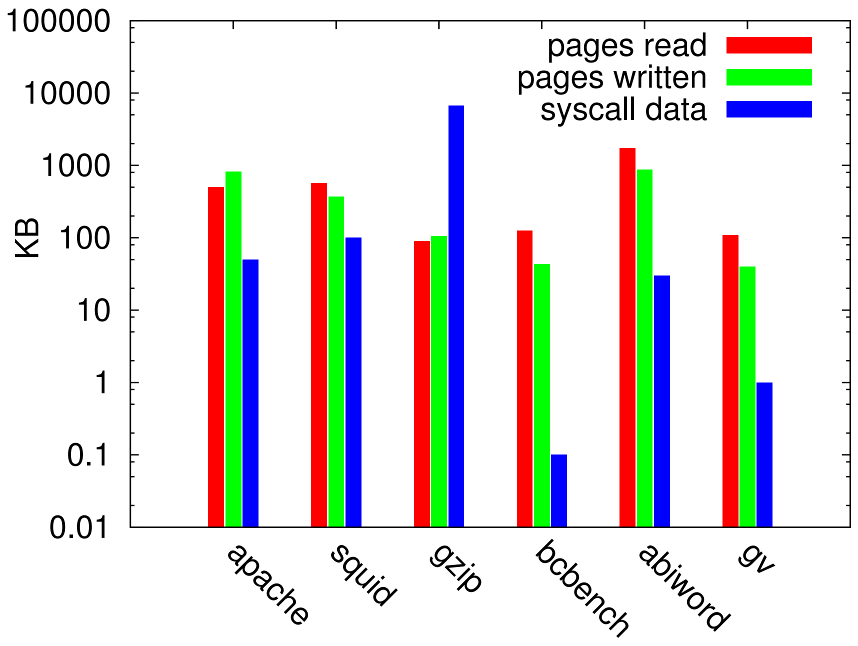


Figure 6.10: Space breakdown (1 second interval)

mechanism to replay on Linux, which intercepts and replays the system calls more efficiently. The higher replay times on Windows is due to Pin's instruction-level instrumentation. We observe that the replay times for applications such as `apache` and `squid` were much smaller compared to their recording times. A recording of a one second interval could be replayed in a few tens of milliseconds. This speedup is due to the fact that server applications spent a significant part of their time in `poll` and `select` system calls. During replay, vPlay-user readily returns from these system calls without the wait.

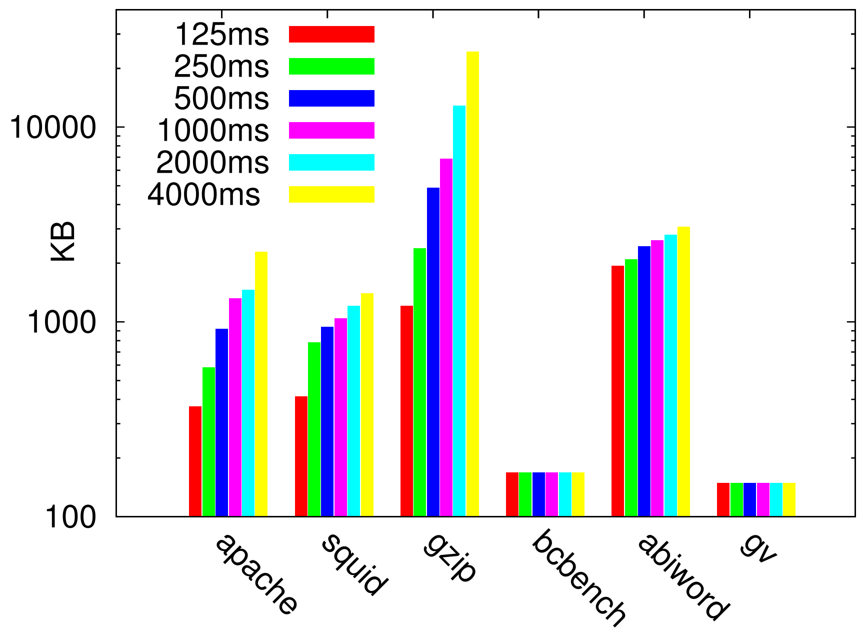


Figure 6.11: Recording storage

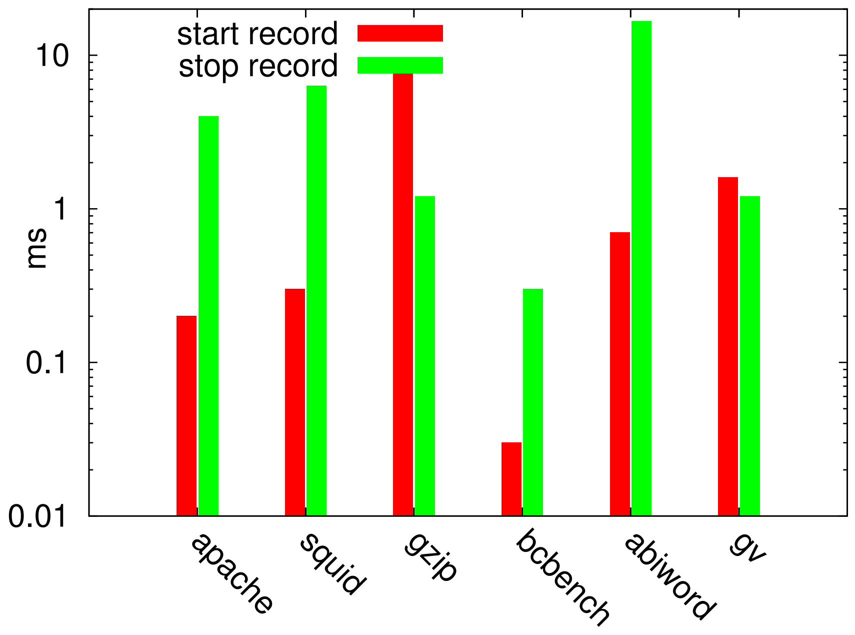


Figure 6.12: Recording latency (1 second interval)

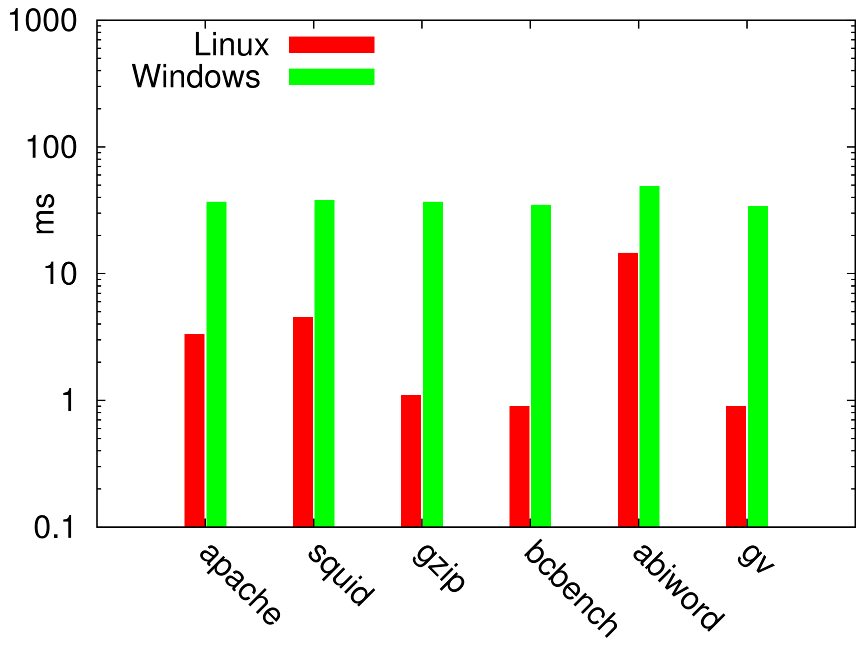


Figure 6.13: Load time

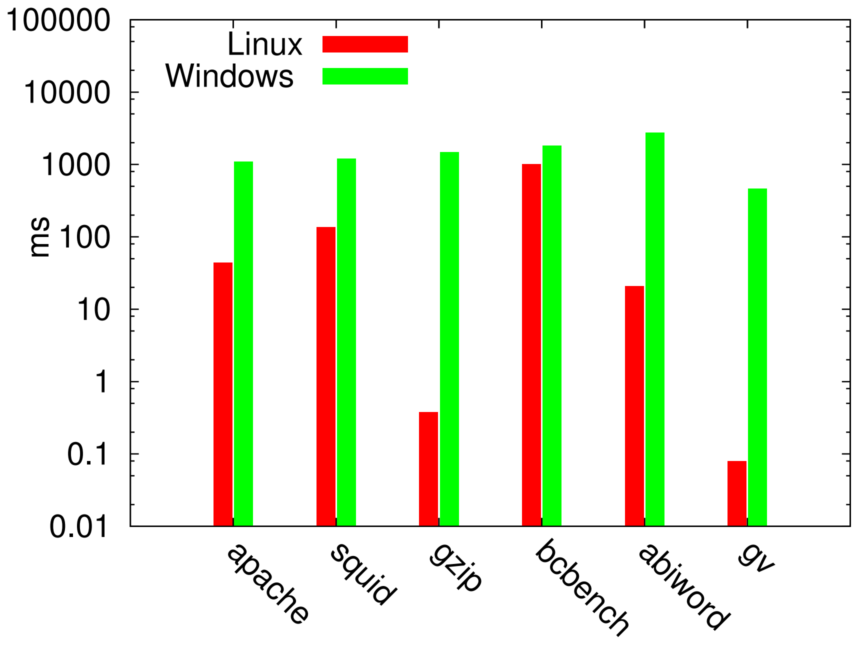


Figure 6.14: Replay time

Chapter 7

Related Work

Many diagnosis and debugging tools have been developed. While interactive debugging tools [21] are helpful for analyzing bugs that can be easily reproduced, they do not assist with reproducing bugs. Techniques for compile-time static checking [75; 17; 38] and runtime dynamic checking [25; 29; 16; 24; 27; 79] are useful in detecting certain types of bugs, but many bugs escape these detection methods and surface as failures, to be reproduced and debugged in the developer environment.

Because of the prevalence of buggy software, bug reports are commonly used. Some application vendors [20; 41; 45] provide built-in support for collecting information when a failure occurs. Other sophisticated mechanisms [28] may provide more comprehensive data including traces and internal application state, in an attempt to ensure that sufficient context is recorded to be able to reproduce and possibly fix the bug. However, they are often limited in their ability to provide insight into the root cause of the problem because they represent the aftermath of the failure, not the steps that lead to it. Privacy mechanisms have been developed to reduce privacy concerns regarding bug reports [7], but like all bug report mechanisms, they do not provide the ability to deterministically replay bugs to reproduce them.

7.1 Checkpointing

Checkpointing has been a focus of extensive study. Checkpointing systems [61; 65; 56; 71; 31] allow application state to be rolled back to a point in the past. Some of them [8; 18] have been applied to cyclic debugging, where the intent is to reduce the waiting time in repeated debugging cycles. Most of these techniques are only applicable to compute-bound parallel jobs. More recent implementations [51; 22; 34] of checkpointing are able to checkpoint a more general class of applications. Even though checkpointing the complete state of an application has proved to be difficult [22] and requires extensive kernel support [42], they typically aim to checkpoint the application state as completely as they can to minimize the impact of checkpointing on the application after it resumes. In particular, they checkpoint the entire virtual memory of the application even though most of the state may not be relevant for debugging. Given large memory footprints of modern applications, these techniques usually store the checkpoints on secondary storage, incurring high overhead during the process. As a result, they cannot afford to take frequent checkpoints necessary for debugging, especially when the application is running in production.

Partial checkpointing is substantially different from regular checkpointing [55; 51]. In particular, the system state of the application maintained internally by the operating system, such as the state of file descriptors and the state of various operating system resources, is not included in the partial checkpoint. It allows partial checkpointing to be implemented without major kernel changes and enables a partial checkpoint to be replayed even on a different operating system. Regular checkpointing allows normal execution to be resumed for an arbitrary amount of time. Because the state needed by an application in its future execution can be arbitrarily large, regular checkpoint implementations typically impose dependencies on the underlying system to reduce storage requirements, such as requiring that files in persistent storage be available to the resumed application. In contrast, partial checkpointing does not impose such a requirement because, the specific portions of data on disk, including portions of application binaries themselves needed by the application during replay, are included in the partial checkpoint. Partial checkpoints are also small, since they do not need to support normal execution, only deterministic replay over a fixed time interval.

Various checkpoint optimizations have been developed, including incremental checkpointing [57] and logging. Logging records a checkpoint by recording old values of memory locations before they are updated, then recovers the checkpoint when needed by taking the final system state and restoring the logged values in reverse log order. Partial checkpointing differs from logging in at least four important ways. First, logging is used for data memory locations and still requires the original code, including all application binaries, to use the checkpoints. In contrast, partial checkpoints include any necessary code pages used and do not require access to any of the originally used code. Second, logging provides a full checkpoint which contains more state needed for normal execution, whereas partial checkpoints are only for deterministic replay over a fixed interval. Third, logging can require saving multiple old values of a memory location to produce an initial checkpoint, whereas partial checkpointing requires saving only one value of each memory location. Finally, previous proposed logging approaches are designed for hardware support not available on commodity machines.

Netzer and Weaver [50] proposed an optimal tracing mechanism that has some similarities to partial checkpointing. Similar approaches have also been designed for hardware support not available on commodity machines [78; 47; 36]. Tracing creates a checkpoint for replaying from some starting point by recording values of memory locations when they are initially read, then restoring all of those values upon replay. Tracing differs fundamentally from partial checkpointing as it does not support replay in a different environment and requires the availability of the same instrumented application code during replay. In contrast, partial checkpointing is primarily designed for replay without requiring the original application code or other software used during recorded execution. Tracing incurs very high recording overhead, up to seven times native performance on simple applications, whereas partial checkpointing is low overhead and fast enough for production use.

7.2 Record and Replay

Record-replay mechanisms provide the ability to record the execution of a running application and use that recording to replay its execution later. Replay may also be done live such that the recorded and replayed instances of the applications run simultaneously on different hosts. Most record-replay techniques [36; 64; 62; 69; 46; 53; 2; 35] have been applied to debugging and recovery. All of these approaches impose crucial dependencies between the environment at the time of replay and the original production recording environment. All previous approaches assume the availability during replay of all software code used during recorded execution. Combining the key features of transparency, determinism, and low overhead has been difficult to achieve with record-replay, especially for multi-threaded applications on multiprocessors. Hardware mechanisms face a high implementation barrier and do not support record-replay on commodity hardware. Application, library, and programming language mechanisms require application modifications, lacking transparency. Virtual machine mechanisms incur high overhead on multiprocessors, making them impractical to use in production environments [15]. To reduce recording overhead, various mechanisms [2; 53] propose record-replay that is not deterministic. Building on SCRIBE [35], vPLAY addresses these shortcomings using a lightweight operating system mechanism to provide transparent, fully deterministic record-replay for multi-threaded applications on multiprocessors with low overhead.

Operating system mechanisms [64; 69] may record input data through system calls, but still require the availability of all files, including application binaries, during replay. For example, consider use of a memory mapped file or access to a memory mapped device, both of which would impose dependencies on devices and files from the original recording environment. Neither of these types of data would be included by recording system call arguments or results, as has been previous proposed. Application, library, and programming language mechanisms [62; 23] not only require access to binaries during replay, but they also require access to source code to modify applications to provide record-replay functionality. In contrast, vPLAY requires no access to any software from the production recording environment, including application, library, or operating system binaries. Flashback [69] proposes a lightweight checkpointing scheme based on `fork` system call, which allows a programmer to record and replay certain type of bugs. Repeated testing to trigger the bug, recording its occurrence and replaying it to analyze its root cause, all have to occur in one user session at the programmer site. The checkpoints it generates cannot be saved to persistent storage, or transmitted to an offsite programmer for analysis. It also doesn't address nondeterminism due to shared memory. In general, checkpoint/rollback schemes that don't allow production use require the bug to be reproduced offline through repeated runs. Some times the bug may never occur due to probe effect introduced by the system.

A number of speculative tools leverage record-replay or checkpointing. Triage [72] proposes a diagnosis protocol to automatically determine the root cause of a software failure in production. ASSURE [66] and ClearView [54] attempt to automatically diagnose a failure and automatically patch the software, with a goal of quickly responding to vulnerabilities. While such techniques may work for a limited set of well characterized bugs, they are generally not suitable for many common bugs which require intuitive faculties and application-specific knowledge of a human programmer. For instance, the right set of program inputs and environment manipulations to be used for each repetition of the execution heavily depends on the application and is generally not possible to automatically generate.

Several replay systems [18; 9; 52; 19] address application nondeterminism as an independent problem. They provide varying degrees of support for nondeterminism by recording and replaying the nondeterministic events that affect the application. Most of them are able to record and replay system calls. Replay is generally restricted to identically configured systems running the same operating system and they cannot handle discrepancies in the application environment. Due to high frequency of nondeterministic events, they produce large amounts of data, especially for long application runs. Some [18; 52] address shared resource nondeterminism by capturing the interactions among threads and replaying them. They require cooperation from the application and are nontransparent. The system described in [63] uses a notion of repeatable scheduling which records and replays each context switch at a thread library level to make the application deterministic. However, replaying the global scheduling instead of preserving the reference order to the resource of contention imposes high overhead. The approach is limited to uniprocessor systems and requires modifying the application. [3] provides a record and replay system for concurrent shared memory accesses in multiprocessors. It relies on monitoring the memory bus using a specialized hardware to log memory references. While the application itself doesn't incur any runtime overhead, the system generates a large amount of log. Dunlap et al [15] propose virtual machine based deterministic replay in SMP environment. They report an overhead of several times the uninstrumented case. A few industry vendors [70; 40; 26] also offer hardware or co-hardware/software solutions for fault-tolerance based on record and replay. They commonly connect the primary and secondary servers by means of proprietary hardware.

Hardware mechanisms [78; 47; 43] record data accesses at an instruction granularity, but do not record code and rely on the availability of binaries to replay instructions. BugNet [48] uses load based checkpointing where the operand values accessed by load instructions are recorded at the hardware level to replay the execution. Load-based checkpointing is conceptually equivalent to other checkpoint-logging systems, except that the concept is applied at the instruction level, where input operands to the instructions are recorded, but instructions themselves are not recorded. Partial checkpointing is conceptually different from this approach because vPlay considers all data including the instruction opcodes as external inputs in order to produce a self-contained recording.

7.3 Virtualization and Emulation Mechanisms

A number of virtualization mechanisms exist which decouple the application from the underlying system. Virtualization enables benefits such as consolidating applications on to the same physical hardware for better utilization, providing a flexible infrastructure fabric implemented in software which can be easily managed, templatizing application environments for repeatable deployment, seamless application management where applications and their environments can be moved between physical machines etc. Virtualization mechanisms typically employ a model where a virtual abstraction of a lower layer in the software-hardware stack is created such that the next layer above can run on top of the virtual abstraction. The two most widely used types of virtualization are hypervisors [74; 5; 4] and containers [51; 59; 60], which abstract the hardware and the operating system respectively.

Hypervisor virtualization [74; 5; 4] techniques provide a virtual hardware abstraction which can run a guest operating system along with all its applications on top of a different host operating system. The guest operating system provides the interfaces and services necessary for the application to function. The hypervisor creates virtual instances of hardware resources which appear as real hardware to the guest operating system. Qemu [5], for example, uses a binary emulation method based on gcc which allows it to run applications built for an operating system and architecture on a different operating system running on a different architecture. Binary instructions dispatched by virtual CPU are emulated on the host operating system and architecture and the entire system, including the system chipset and devices are emulated. In general, a hypervisor internally maintains the state of the virtual CPU, memory, disk and other resources and allows a snapshot of the state to be created. A user could create a virtual machine snapshot which encapsulates the entire state required to run an application together with its operating system and other processes running on it. The virtual machine snapshot could be shared with the developer so that the developer does not have to replicate the original production environment to reproduce a bug.

Using virtual machines to encapsulate and share the application environment has several problems. First, the size of a virtual machine snapshot is large. Since hypervisors provide a virtual view of the hardware, the internal structure of the operating system that runs on it, the applications and their behavior is generally unknown at the hypervisor level. Even though the problem occurs only within the application, there is no way for a hypervisor to only capture the application state. As a result, a virtual machine snapshot contains the state of the application along with the state of the entire guest operating system, all of the physical memory, and all storage state. The large snapshots are not easy to be shared between the users and developers. Creating a snapshot can also be a slow process. Second, since the state of the virtual machine may encapsulate the application environment in its entirety, it may contain sensitive user state which cannot be externally disclosed. Third, a hypervisor is a complicated piece of software and it needs to intercept, emulate and virtualize low level hardware events occurring at high frequency. Even though many recent advances make them quite efficient, they still impose considerable overhead which may not be acceptable for types of workloads.

Containers [51; 59] or application virtualization is an alternate form of virtualization which partitions an operating system into multiple instances, each with its private set of resources. It is typically implemented at the system call layer by intercepting the system calls which can potentially cause resource conflicts with the host. Containers provide a virtual operating system environment which can host applications independent of applications running on the host or in other containers. A Container decouples the application running within it from the underlying operating system, by providing it with a private view of the system resources. The application directly runs on the underlying operating system but its access to the operating system resources are intercepted and virtualized, such that it runs consistently even as the container migrates from one system to another.

Even though containers decouple the application from the underlying system, they cannot insulate the application from major differences in the operating system environment. In particular, since the application directly runs on the target operating system, its general memory layout, services provided etc. are expected to be the same as the original environment. For example, an application built for a different operating system from the underlying target operating system cannot be supported. Furthermore, like virtual machines, a container also requires that the state of persistent storage along with the application environment is available at the target.

Operating environments such as Cygwin [14] and Wine [77] implement a compatibility layer which provides the necessary libraries and build environment to allow applications of one operating system to be compiled and built on a different operating system without requiring them to be developed from scratch. Cygwin provides a Linux-like environment on Windows, with many Linux applications ported to the Windows Cygwin environment. Even though Cygwin based Linux applications look and behave like their native Linux counterparts, they are win32 applications from the perspective of the operating system. They are compiled as Windows binaries and use native Windows services. vPlay Container environment on the other hand is very different and allows Linux applications to be natively replayed on Windows.

7.3.1 Virtual Machine Based Replay

Virtual machine mechanisms [15; 10; 33] may allow replay on a different host environment from recording, but typically rely on the availability of complete virtual machine image, including all software code, its entire file system and additional file snapshots, to resume the execution. Not only does this require a large amount of data, but this is often impractical for bug reproducibility as customers are unlikely to allow application vendors to have an entire replica of all of their custom proprietary software. Decoupling provided by partial checkpointing is also different from that of virtual machines, which may decouple replay of an application in a guest operating system from dependencies on the hosting environment, but still require during replay the complete virtual machine image with all of the installed binaries used at the time of recording.

The space requirement of virtual machine snapshots would be several orders of magnitude higher than that of partial checkpoints. Since VM snapshotting involves checkpointing the state of the entire operating system and its applications, including the state of secondary storage, the amount of data in each snapshot is large and it can take several tens of seconds or minutes to complete. Crosscut [11] aims to extract a subset of data offline from a complete recording of a VM to reduce the size. However, it still requires a heavy weight instrumentation during recording and the original log it generates is large. On the other hand, since vPlay only captures the most relevant application level state, it is able to take several partial checkpoints of the application per second. The high checkpoint frequency also allows for quick forward and backward movement of execution during replay. Furthermore, virtual machine based logging imposes high runtime overhead given the large number of low level hardware events. For instance, only a fraction of the network traffic processed by a virtual network card would be visible to the application and consumed by it.

Chapter 8

Applications and Extensions

8.1 Debugging as a Service

Cloud computing is an emerging service paradigm where managed virtual assets are offered to the users as a service by the cloud service provider. The assets typically consist of a preconfigured operating system or application platform packaged as instances of virtual machine appliances. Since the user or the customer does not have to acquire dedicated resources and manage them, cloud resources are used by businesses to simply their infrastructure by out sourcing their operations to the cloud provider.

Minimizing application downtime is a common objective for cloud providers, end-customers and software vendors alike, yet problem determination in a cloud environment remains an elusive and time-consuming process. As legacy applications adapt to the incipient cloud environments which are significantly different in their response characteristics to the physical hardware, many latent application bugs surface. A cloud environment hosts several virtual machines with over-committed memory and CPU resources on the same physical hardware in a multi-tenant configuration. The sharing of resources leaves the applications vulnerable to interference from other virtual machines and triggers unexpected behavior.

Most existing debugging tools are designed to be used by the developers in a development environment and are unsuitable for a cloud ecosystem. When an application fails in a managed cloud environment, the user has little access to or control over the underlying environment. Since traditional debugging processes cannot be applied, the user has to depend on the cloud itself for assistance.

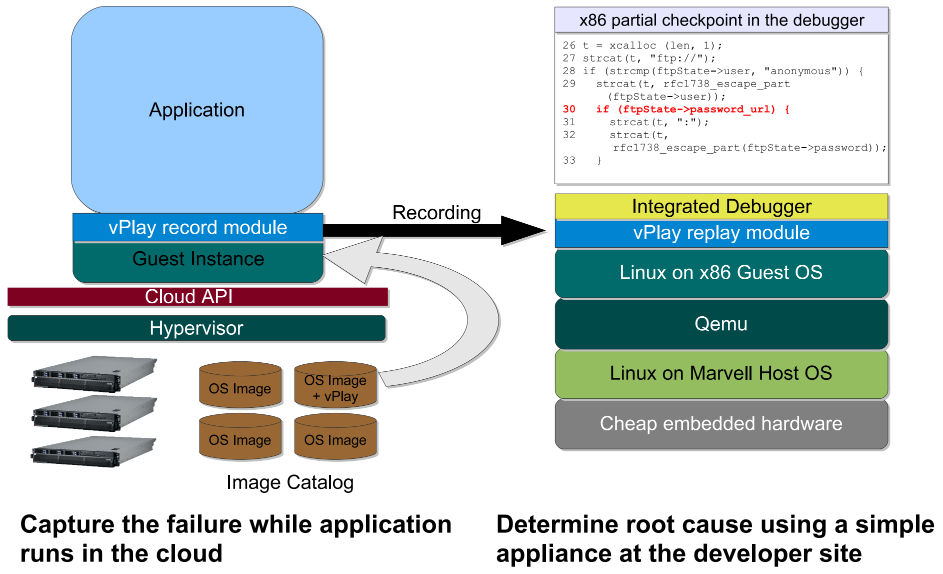
While the cloud computing model presents certain challenges to applications which are trying to adapt to the cloud, it also provides an excellent substrate for experimental and innovative services. Features and services involving multiple software components, requiring custom changes and configuration to the software stack could be easily deployed on the cloud in a contained manner. For example, one of the hurdles to a widespread adoption of vPlay is that the vPlay system needs to be installed on the platform where the application runs in production, and at the target where the developer analyzes and resolves the problem. Sophisticated debugging tools such as vPlay with rich feature set and support for a wide range of unmodified applications often rely on specific kernel extensions to gain access to the application internals, which may not be supported by main-stream kernels. However, deploying these extensions into existing environments is challenging in practice. Any kernel extensions deployed as kernel modules on a host platform may violate the service agreement with the operating system distribution vendor, increasing the adoption barrier.

In this section, we present a practical debugging framework and a model with emerging cloud computing eco-systems as a reference. The framework consists of two components, namely recording and replay appliances. The recording appliance is a part of the cloud infrastructure and it produces a recording of bugs encountered by an application in the cloud. The replay appliance is provided as a simple hardware device which reads previously generated recordings and reproduces the bugs captured within them for the developer to analyze.

The solution benefits the end-users by minimizing the application downtime, the application vendors, by enabling them to quickly fix the problems, and the cloud providers, by enabling them to offer value-added debugging services.

8.1.1 Recording Appliance

The cloud model allows debugging to be offered as a cloud service without requiring explicit changes to the user environment. A cloud service provides a catalog of virtual machine appliances, each typically customized for a specific application and purpose. The user selects and instantiates the desired appliances to host the user's applications. For each



```
t = xcalloc (len, 1);
strcat(t, "ftp://");
if (strcmp(ftpState->user, "anonymous")) {
  strcat(t, rfc1738_escape_part
    (ftpState->user));
  if (ftpState->password_url) {
    strcat(t, ":");
    strcat(t,
     rfc1738_escape_part(ftpState->password));
  }
```

Figure 8.1: Debugging as a service in the cloud

appliance available in the catalog, an alternate vPlay-enabled appliance is provided. The vPlay-enabled appliances contain vPlay's recording function integrated within them and are configured such that when they comes up, the primary application of the appliance is already instrumented and undergoes recording. When the application encounters a failure, the appliance automatically generates corresponding partial checkpoints and logs.

The appliance model of deployment makes vPlay transparent and seamless to the user. The recording functionality is contained to the appliance, and no other component in the cloud is affected. The cloud provider would offer support for the entire appliance as an integrated software stack, relieving the user from having to work with individual ISVs and distribution partners. The user has the flexibility to choose the standard, or vPlay-enabled versions of an appliance based on the need. For instance, vPlay-enabled appliances could be used to run unhardened beta software, which is likely to contain defects. Or they could be used internally, as a part of development-testing cycle. Applications or usage-scenarios which are highly performance sensitive could use the standard versions of the appliances.

The cloud model also facilitates privacy. Since the end customers must already have some level of trust with the cloud provider to be hosting their applications in the cloud, the cloud provider could act as a neutral third party between the end customer and the application vendor, and host a debugging service that the developer can access. Since the developer would only have access to the service API exposed by the cloud provider and not to the underlying infrastructure, risk of the developer circumventing privacy features built into the platform as an operating system extension is minimized. The integrated vPlay recording function monitors the application running within the appliance, and any failure that occurs is captured into a small recording consisting of relevant partial checkpoints and logs. The collected data is shared with the vendor of the failed application for analysis and debugging.

The data generated by the record component is encrypted at the source. Even though vPlay only records discrete pieces of relevant data required to reproduce the bug, it may contain sensitive client data and the user may not approve unrestricted access to it. Internally encrypting the data as a part of its generation, before it is shared with anyone ensures that the privacy is maintained.

8.1.2 Replay Appliance

The replay appliance is based on a simple hardware device which assists the application developer in reproducing and fixing previously recorded bugs. A tailored hardware device built-in firmware limits the use of the appliance to debugging and prevents arbitrary manipulation of data contained in the recording. The device accepts the encrypted recording of the bug and relevant symbol information as inputs through a secure REST web interface using which the developer can analyze the bug. The interface shows the application state at the beginning of the recording interval and provides the standard debugger controls to interact with. To debug the problem, the developer would be able to set breakpoints and single step through the source lines through the web interface. The architecture of the device is shown on the right hand side of Figure 8.1.

The replay component integrates with a debugger and runs within a virtual machine instance consisting of a guest operating system environment. The guest environment also runs the web services necessary to expose the debugger interface to the developer through a REST API. The guest environment itself runs on a hypervisor host which emulates the target architecture of the recording, while running on an underlying low-power embedded hardware such as Marvell. Marvell hardware provides a cheap but full-featured server platform. An emulator such as Qemu not only provides the virtual machine but also enables the entire replay environment along with the operating system to be run on the embedded hardware. Linux/Marvell operating system runs as the lowest software layer, directly on the embedded hardware. For in-premises customer use where privacy is not a concern, the replay component can also be provided as a virtual appliance. The virtual replay appliance would still provide all the benefits of replaying the bug independent of the source environment, except that it is run on a standard hypervisor host rather than a hardware device.

The replay device is designed to be tamper-proof. The only external means to access the device is the streamlined web interface which takes a recording and provides debugging control over its replay. The hardware guarantees that the developer cannot use the captured state for any reason other than problem diagnosis. The recording of the bug is internally decrypted using a key included in the device at manufacturing time. The key is only available to the cloud framework which produces the recordings and the device that replays it. This security model is analogous to the Digital Rights Management model used by content providers to distribute their content to the end users. The encrypted digital content in the media can be played only by authorized play-back devices but cannot be used to extract the data or misuse it otherwise. In the case of debugging, the developer is able to single step etc, but not arbitrarily access partial checkpoint data. The debugging interface is streamlined such that features like dumping arbitrary regions of application memory are disabled.

In contrast to the conventional debugging approach, the replay device allows the failure to be securely played back by the developer, similar to logic analyzers used by hardware engineers, without having to acquire specific hardware, replicate the customer setup, install specific version of the software and configure as per the bug report etc. The tamper-proof device acts as a trusted component in the target replay environment. It also insulates the user from vPlay's kernel dependencies etc by integrating those dependencies within the appliance. Additionally, the device provides a convenient point for pay-per-use metering and chargeback services.

8.2 Application Beyond Debugging

8.2.1 Micro-aps

Like hardware, software is growing more and more powerful, packing more functionality within each application. However, common usage patterns of applications exercise only a fraction of the application functionality. Most configuration options and menu items remain unexplored and unused. Even if the user is interested in just one particular feature of the application, the entire application with all its dependencies is required to be installed. An application like Photoshop, for instance, occupies several gigabytes of disk space with support for hundreds of image transformations, but the user may just want to apply one particular photo effect to an entire image collection.

Similar to the way hardware is partitioned into multiple cores or multiple virtual machines for better usability and simpler management, it may be beneficial to create functional partitions of otherwise monolithic software. We refer to these functional partitions as *micro-aps*. Micro-aps deliver the effect of their respective function without requiring the entire application. One approach to isolate individual functions of an application is to modify the source code and build each piece of functionality as a separate application. However, doing so can be complicated, expensive or infeasible. Instead it is desirable to quickly and easily create micro-aps by directly manipulating the functionality at the binary level, without requiring source code changes. Partial checkpointing provides a convenient method to do that.

8.2.1.1 Creating Micro-aps

Micro-aps are created using the `start` and `stop` primitives provided by vPlay. In case of debugging, vPlay continuously monitors the application and keeps track of its most recent execution to capture failures which could happen at any time. However, since Micro-aps represent a specific user-specified function of an application, no continuous recording is necessary. They are created by issuing the `start` command, followed by issuing the application input to trigger the desired application behavior and then issuing the `stop` command.

Micro-aps are essentially vPlay recordings with data and logic quilted together into one package. vPlay's recording mechanism naturally creates them. Simply running an application within vPlay gathers coherent portions of the application and creates a mapping between the activated functionality and a self-contained slice of the application which implements that functionality, along with respective data. Application developers typically create test cases as a part of a typical development-test process, which exercise various features of the application. The test-cases map respective features to a region of the overall application binary and can be used to create the Micro-aps.

vPlay recording mechanism is extended to capture additional information to enable Micro-aps to connect with external entities when they run. Unlike debugging, not all external interactions of the application are nullified or virtualized. In addition to the data that is directly consumed by the application, vPlay also records the identity and configuration of the external entities that the application interacts with. As we illustrate in the examples below, this includes information such as the path of the file where a processed image is stored, the name of the playback audio device and the sampling frequency used etc.

8.2.1.2 Running Micro-aps

Running a Micro-ap is quite different from replaying a vPlay recording. Whereas a vPlay recording is replayed for the exclusive purpose of analysis and debugging, a Micro-ap is expected to run live. In particular, a Micro-ap must be connected to the external world as it runs, rather than just run within an isolated container with its interfaces virtualized. To do that, vPlay must emulate system calls that are not otherwise intercepted for debugging. For example, when the application makes a write system call on a file descriptor, the descriptor must be valid and the write system call must actually occur.

A running Micro-ap must be connected to the external entities with which it interfaces. When a Micro-ap is replayed, vPlay supplies the data captured as a part of the recording as input to the Micro-ap logic. The logic processes the input data and produces the output. For common user applications, most of the application output boils down to writes to various types of file descriptors (regular files, devices, sockets connected to external clients etc.) Normally, any output generated by the replaying application is either captured to a file or simply discarded, as in the case of interactive debugging where the output is displayed to the user. For Micro-aps, these outputs would have to be delivered to their data sinks to drive respective external entities.

Outputs generated by a running Micro-ap is routed through a vPlay *proxy* which maintains a mapping between the data sources of the application and the data sinks of the external entities. The proxy is implemented as an extension to vPlay's system call replay mechanism. For each data source in the application such as an open file descriptor to a regular file or an open socket connection, the proxy initiates a separate connection within the context of the replay process and creates a mapping between the two. It establishes and configures the Micro-ap's connections as it were when the `start` command was issued. This is required because the application may have created the connection and configured it before the `start` command was issued and expects it to be valid when subsequently used. Any new connections setup during the execution of the Micro-ap are emulated. For example, when a Micro-ap opens a file, the proxy intercepts the system call and opens the file. Similarly, when the Micro-ap subsequently writes to the file, the proxy intercepts the system call and forwards the data to the respective file. Any file descriptors used by the application would always be invalid during replay since the file is never directly opened by the application. They just act as pointers to their counter parts maintained by the proxy. In fact, the application may not even be able to directly open the file if the target operating system happens to be different from the source. In that case, the proxy translates the semantics of the write operation between the application and the target environment.

From among the recorded identities of the external entities with which the application interacted during recording time, the user selects a subset of interfaces that need to be "live" during replay. For example, it may not be necessary for the Micro-ap to interact with all external clients or perhaps it may not have to produce any trace files. Note that, even though data from the application is transmitted to the external world, any data sent back to the application from the external clients is discarded, since application inputs always come from the recording.

In some specific cases, it may be possible to templatize a Micro-ap by removing data from it and retaining the code such that it can be used with alternative data. For example, a replaying database server may receive a request for a different row than one originally recorded. Data portions of the recording can be identified based on the provenance information stored in the recording. Alternate data from the user is provided to replay. However, the ability to replay with alternate data depends on the application and in general, the Micro-ap model works best for external entities which are passive data sinks. We illustrate the typical usage model with a few applications where the Micro-ap approach is suitable.

Self-decoding Data: A variety of data can be packaged as Micro-aps along with the logic to decode it. Self extracting executables of compressed files is an example. The compressed data is included within the data segment of the executable which implements the decompression logic. With logic packaged along with the data, the target platform does not need to have the specific application that can decompress the data. While self-extracting compressed files are created using a special feature provided by the compression utility, vPlay can be used as a general technique to create Micro-aps for different types of data transformations. vPlay allows the decoder logic to be quickly isolated by running the application within vPlay to capture its recording as a Micro-ap so that the end user can readily use it without needing the codec or driver. This approach is different from building the decoder into a virtual machine appliance, which would require a different appliance for each data type.

Audio Player: Applied to an audio player, vPlay recording would automatically isolate the required logic of the respective codec from a binary which may support many audio formats. The codec is packaged along with a segment of the audio file, so that it could be played independent of the target environment. Before running the Micro-ap, the proxy would open the audio device, set the sampling frequency and other parameters as previously recorded. When the Micro-ap runs, it directly renders the recorded data to the audio device.

Load Generator: Applications often require testing with real-world workloads before they are released. However, collecting a realistic workload can be a difficult task. Using vPlay it is possible to create a Micro-ap representing a canned user workload which can then be applied repeatedly as a part of regression testing. For example, to capture a user-generated workload for a database server, the web application which interfaces with the database server and the external users can be recorded. Before the workload Micro-ap is executed, the proxy would establish a connection to the database server and maps application's data source to it. The user click stream and requests from a real usage are captured as requests to the web application, which in turn translates them into a database workload.

8.2.2 Tracing

Application traces contain a wealth of information about the behavior of the application and the specific events and interactions that occurred during the execution of the application. They are used for a variety of reasons including forensics, debugging, performance analysis and archiving. Since traces are typically captured as simple text, they lend themselves well for use with tools such as log analyzers which mine the traces to extract relevant information in a platform independent way.

Capturing traces requires instrumenting the application control paths. An application relies on services provided by layers of software and hardware below it and its events can be traced by instrumenting at any of those layers. The frequency of events and the overhead of instrumentation depends on the level of instrumentation. For instance, tracing the execution at instruction level granularity can be orders of magnitude more expensive in runtime overhead and space requirement than tracing the application at the system call level. On the other hand, the granularity of tracing required depends on the purpose for which traces are used. Fine grain traces may be necessary for optimizing critical application control paths whereas high level traces may be sufficient for visualizing application's interactions with external components.

Some applications maintain a running trace of application events in order to assist the debugging process in case of a failure. Typically such tracing is limited to high level events such as system calls or network packets. While existing tracing tools can provide fine-grain traces, they cannot be applied to production software due to the runtime cost associated with tracking application events occurring at high frequency. Also, storing execution traces as static data can consume large storage space and extracting relevant information from large quantities of trace through search can be cumbersome. Determining the root cause of many common bugs requires finer grain detail such as memory reference trace or a trace of instructions.

Using vPlay for tracing provides two key advantages. First, the user does not need to know at what granularity to trace the application. vPlay's recording mechanism captures just the right amount of data necessary within a recording to reproduce the complete execution. Any required trace can be generated by running replay through existing tracing tools. A recording produced by vPlay serves as a compact representation of an application trace, which implicitly encodes the application state at each point of its execution. Relevant application state can be quickly obtained by setting breakpoints or watchpoints within the debugger. While conventional tracing only provides traces which are static, partial checkpoints can be used to derive a variety of application traces such as instruction trace, system call trace and memory reference trace, by replaying the application under instrumentation tools such as Pin.

Second, vPlay efficiently captures the partial checkpoints of applications in production and allows relevant traces to be derived offline by running replay through existing tracing tools.

8.2.3 Application Monitoring and Visualization

Applications represent a significant enterprise asset and keeping them running at optimal level is critical for business. Many tools [68] exist which monitor the enterprise environment and report the health of various applications. They collect generic pieces of information obtained from tools like `iostat` and `vmstat`, and typically present it as a graphical summary. While such tools are regarded as generally useful, they fail to convey a comprehensive end-to-end picture of the application environment. Many key events in a data center occur as internal application events which are not captured by standard utilities.

A recording generated by vPlay provides exhaustive knowledge about the functioning of an application in a given interval. vPlay captures every program input along with its provenance information, and the resulting recording would be able to expose key application events missed by conventional monitoring tools. While standard monitoring and tracing is enabled during normal operation, vPlay recording could be used to randomly sample the application behavior to gain a deeper and more comprehensive understanding of the application health. The additional information about intimate application events provided by vPlay could be used to enrich the pool of data collected by standard tools for offline analytics and visualization. For example, precursor symptoms for a degraded performance and an eventual crash of an application may appear as unusual execution patterns within the application. The early signs of divergence from normal or healthy patterns can be identified with trained machine learning algorithms used by log analysis tools if detailed internal events of the application are presented.

8.2.4 Information Archiving

Information archiving is a major market segment in data storage driven by regulatory compliance requirements. Governmental regulations require businesses in some industries to maintain records of certain transactions. In order to prove that a particular action has been taken or a transaction executed, the result of the action is typically archived as static data. Instead, it may be more effective in some cases to archive the action itself as a partial checkpoint. A partial checkpoint not only serves as a record of the transaction, but it also implicitly captures the required proof elements along with substantial detail associated with the action into a compact representation. The action may involve nondeterministic steps which are also captured by the recording. The provenance information for various program inputs that vPlay includes would provide the necessary context to the forensic investigation team in case of a security audit. For example, a bank may be required to notify affected clientele about an update to the terms of a mutual agreement. While a static record of such a notification may include high-level data elements such as date, place etc., a partial checkpoint would capture the context of the transaction at a much greater depth. It would include the identity of the server which negotiated the transaction, the specific response received etc., which helps the corporation to be better prepared in case of a litigation.

8.3 Limitations

vPlay is an initial effort to address fundamental problems of existing debugging model using a simple and lightweight virtualization mechanism. However, there are a number of limitations, yet to be addressed:

Recording Windows Applications: Current implementation of vPlay is largely based on the semantics of Linux applications. The resources and interfaces used by the Linux applications are recorded and the same resources and interfaces are emulated on other environments including Windows. Even though the general concepts developed as a part of vPlay may be applicable to recording applications of other operating systems, the implementation may require different engineering. For example, Windows applications are vastly different from Linux applications. Their structure, API available to them, the general programming model are all very different. Windows implements its system calls as a part of win32.dll and kernel.dll system libraries. In many cases, the system calls completely occur in the user space without switching to the kernel mode, which makes the user-kernel boundary difficult to intercept. Windows also implements a lower-level *native API*, which is perhaps closer to the system call API on Linux, but the semantics of the API are undocumented. In general, lack of access to Windows internals makes transparent instrumentation of Windows applications difficult.

Short error propagation distances: Not all failures may be reproduced by vPlay. Although reported as rare [48], the root cause of some failures may lie far in the past, outside the recent execution context recorded by vPlay.

Accurate system call specification: In order to accurately record and replay system call responses, vPlay requires an accurate representation of the system call API in the form of a data plug-in as described in Section 4.1. Some system calls, especially `ioctl` interface dealing with uncommon devices may have poorly specified semantics, making it difficult to record such system calls. Given additional support from the kernel [12], the memory side effects of those system calls may also be captured correctly.

Connections with external components during replay: The replay function of vPlay is designed to be used with a debugger for analyzing the faulty execution of the application. Any external interactions of the application are not reproduced. As a result, the user may not be able to directly see the effect of replay on external components. For example, replaying an interval of a graphical application would not show the graphical output of the application during that time. However, the user can analyze the interaction between the application and the window system within the debugger to identify any problems related to the graphical output.

Specialized hardware and memory-mapped devices: vPlay assumes that the application uses standard resources arbitrated by the operating system. Any specialized resources and devices used by the application may have unstandardized semantics associated with them and generally not possible to support without specific integration. For instance, artificially making a device page absent may impact the correctness or the device may allow reading the page only once by the application such that vPlay cannot read it again for recording it.

Chapter 9

Conclusions and Future Work

vPlay is the first system which can capture production software bugs and reproduce them deterministically in a completely different environment, without access to the original production environment. vPlay accomplishes this by relying only on an innovative recording mechanism which provides data level independence between the application and its source environment and a lightweight virtualization layer which insulates the replaying application from the target environment. There is no need for access to any originally executed binaries or support data, no need to run the same operating system, and no need to replicate the original setup or do repeated testing.

vPlay introduces partial checkpointing, a simple and novel mechanism to record the partial state required to deterministically replay an application, including relevant pieces of its executable files, for a brief interval of time before its failure. Partial checkpointing minimizes the amount of data to be recorded and decouples replay from the original execution environment while ensuring that all information necessary to reproduce the bug is available. vPlay integrates with a standard unmodified debugger to provide debugging facilities such as breakpoints and single-stepping through source lines of application code while the application is replayed. The captured state, which typically amounts to a few megabytes of data, can be used to deterministically replay the application.s execution to expose the steps that lead to the failure. No source code modifications, relinking or other assistance from the application is required.

Our experimental analysis on real applications running on Linux shows that vPlay

(1) can capture the root cause of real-life software bugs and the necessary bug triggering data and events, (2) can capture partial checkpoints of unmodified Linux applications and deterministically replay them on other Linux distributions and on Windows, and (3) is able to generate partial checkpoints of applications such as Apache and MySQL with modest recording overhead and storage requirements. These results demonstrate that vPlay is a valuable tool that can simplify the root cause analysis of production application failures.

9.1 Future Work

The concepts and techniques presented in this dissertation are geared towards addressing software failure diagnosis. However, vPlay exposes a number of follow-on research and engineering challenges.

General execution across operating systems: Porting applications to various operating systems and maintaining them is an expensive undertaking for development organizations. Even though the vPlay Container abstraction is only applied to the limited case of application replay for problem diagnosis, it could be used as a lightweight virtual environment to host applications built for on operating system on another. A running application relies on the services provided by the operating system. In case of replay, the pre-computed results of most system calls are already available from the recording which can just be returned to the application. In general execution however, all system calls made by the application have to be emulated on the target system. Compatibility layers such as Cygwin and Wine would simplify the emulation. For instance, the Linux system calls made by a Linux application running within the vPlay Container environment on Windows could be emulated by calling corresponding equivalents provided by Cygwin's POSIX API. The same techniques provided by vPlay Container for handling instruction and processor level conflicts would also apply for the general execution case.

Distributed applications: Distributed applications represent important class of applications which are complex and often require significant hardware resources. Several types of distributed applications such as data-intensive map-reduce applications or compute-intensive MPI jobs could benefit from isolated replay of individual processes from a large distributed application. It may be possible to leverage specific attributes of these applications or perhaps directly integrate with their middleware to enhance them with built-in debugging support. These issues remain to be explored.

Operating system bugs: vPlay relies on the kernel to correctly record application's execution, and assumes that the kernel itself is bug free. While reproducing kernel level bugs is not supported, it is possible to record and replay an entire kernel running in the user space within a virtual machine such as Qemu.

Relaxing replay determinism: vPlay disallows any debugging operations that would potentially alter the deterministic execution course of replay. For instance, writing to the registers or other program variables may make the application take an execution course which does not represent its original execution during recording. But in some special circumstances it may be possible to relax the requirement of deterministic replay. For example, it may be beneficial to be able to increase the logging level of the replaying application for debugging, by allowing the user to modify respective logging flags. Replay would take a slightly different path than recording to print extra log statements. By identifying and capturing additional peripheral program state, it may be possible to accommodate small digressions from the the originally recorded execution.

Client privacy: While vPlay strives to minimize the amount of state necessary to be recorded and transmitted to the developer, a partial checkpoint may still contain sensitive client data. To further reduce the recorded state, a partial checkpoint can be preprocessed on-site to generate a memory reference trace by replaying it through an offline tracing tool and filtering out unaccessed memory locations from the pages stored in the checkpoint. To completely avoid having to transmit any raw data, it is conceivable to provide a remotely accessible web interface to a hosted debugger which runs the partial checkpoint at the client site within an isolated Java application which can run on any browser. Integrate replay with an x86 emulator implemented in Java to replay within a browser. The customer would host the web application in his premises and the developer would access the link to debug. Since the code contained in the partial checkpoint is user code, an x86 emulator implemented in Java would be able to replay the recording.

Bibliography

[1] T. Allen et al. DWARF Debugging Information Format, Version 4, Jun 2010.

[2] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, Oct 2009.

[3] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the 1991 ACM Workshop on Parallel and Distributed Debugging (PADD)*, Aug 1991.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, Oct 2003.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr 2005.

[6] P. Bergheaud, D. Subhraveti, and M. Vertes. Fault Tolerance in Multiprocessor Systems via Application Cloning. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS)*, Jun 2007.

[7] M. Castro, M. Costa, and J. P. Martin. Better bug reporting with better privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2008.

[8] S. Chen, W. K. Fuchs, and J. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, 2001.

[9] J. D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT)*, Jun 1998.

[10] J. Chow, T. Garfinkel, and P. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Jun 2008.

[11] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-Stage Replay With Crosscut. In *Proceedings of the 6th International Conference on Virtual Execution Environments (VEE)*, Mar 2010.

[12] F. Cornelis, M. Ronsse, and K. D. Bosschere. TORNADO: A Novel Input Replay Tool. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Jun 2003.

[13] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with Readers and Writers. *ACM Communications*, 14:667–668, Oct 1971.

[14] Cygwin. http://www.cygwin.com.

[15] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, Mar 2008.

[16] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles (SOSP)*, Oct 2001.

[17] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A Tool For Using Specifications to Check Code. In *Proceedings of the 2nd Symposium on Foundations of Software Engineering (SIGSOFT)*, Dec 1994.

[18] S. I. Feldman and C. B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, May 1988.

[19] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Apr 2006.

[20] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct 2009.

[21] GNU. GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/.

[22] C. Goater, D. Lezcano, C. Calmels, D. Hansen, S. Hallyn, and H. Franke. Making Applications Mobile Under Linux. In *Proceedings of 8th Linux Symposium*, Apr 2006.

[23] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec 2008.

[24] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, Jun 2002.

[25] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of The 1992 Winter Usenix Conference*, Dec 1992.

[26] Hewlett-Packard. http://www.hp.com/go/nonstop.

[27] S. Horwitz. Debugging Via Run-time Type Checking. *SIGSOFT Software Engineering Notes*, 25(1):58, 2000.

[28] IBM. WebSphere Application Server V6: Diagnostic Data, http://www.redbooks.ibm.com/redpapers/pdfs/redp4085.pdf.

[29] Intel. Assure, http://developer.intel.com/software/products/assure/.

[30] Intel. Intel corporation, ia32 intel architecture software developper's manual, 2004.

[31] G. Janakiraman, J. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, July 2005.

[32] A. Kangarlou, D. Xu, P. Ruth, and P. Eugster. Taking Snapshots of Virtual Networked Environments. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, Nov 2007.

[33] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Prooceedings of the 2005 USENIX Annual Technical Conference*, Jun 2005.

[34] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *In Proceedings of the 2007 USENIX Annual Technical Conference*, Jun 2007.

[35] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Jun 2010.

[36] T. LeBlanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4), Apr 1987.

[37] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *PLDI Workshop on the Evaluation of Software Defect Detection Tools*, Jun 2005.

[38] M. Luján, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of Java Array Bounds Checks in the Presence of Indirection. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande (JGI)*, Nov 2002.

[39] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic

Instrumentation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2005.

[40] Marathon-Technologies. http://www.marathontechnologies.com.

[41] Microsoft. Dr. Watson Overview, http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx.

[42] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *2nd ACM Workshop on Hot Topics in Networks*, Nov 2003.

[43] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2009.

[44] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Jun 1998.

[45] Mozilla.org. Quality Feedback Agent, http://kb.mozillazine.org/Quality_Feedback_Agent.

[46] M. Musuvathi, S. Qadeer, T. Ball, G. rard Basler, P. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec 2008.

[47] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2006.

[48] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, Jun 2005.

[49] National-Vulnerability-Database. CVE-2010-0308, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0308, Feb 2010.

[50] R. Netzer and M. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 1994.

[51] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, Dec 2002.

[52] D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *Proceedings of the ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD)*, 1989.

[53] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay With Execution Sketching on Multiprocessors. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, Oct 2009.

[54] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically Patching Errors in Deployed Software. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, Oct 2009.

[55] J. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, University of Tennessee, Jul 1997.

[56] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference*, Jan 1995.

[57] J. Plank, J. Xu, and R. Netzer. Compressed Differences: An Algorithm for Fast Incremental Checkpointing. Technical Report UT-CS-95-302, University of Tennessee, Aug 1995.

[58] S. Potter, J. Nieh, and D. Subhraveti. Secure Isolation and Migration of Untrusted Legacy Applications. Technical Report CUCS-005-04, Columbia University, Feb 2004.

[59] D. Price and A. Tucker. Solaris Zones: Operating Systems Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)*, Nov 2004.

[60] L. V. Project. Linux Vserver, http://linux-vserver.org.

[61] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical report, Lawrence Berkeley National Laboratory, Nov 2003.

[62] M. Ronsse and K. De-Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[63] M. Russinovich and B. Cogswell. Replay for Concurrent Nondeterministic Shared Memory Applications. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[64] Y. Saito. Jockey: A User-space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG)*, Sep 2005.

[65] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2005.

[66] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-Healing Using Rescue Points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2009.

[67] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. Technical Report FTR-96-04, University of British Columbia, 1996.

[68] Splunk. http://www.splunk.com.

[69] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Jun 2004.

[70] Stratus. http://www.stratus.com.

[71] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal, 20(227):40-48*, 1995.

[72] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Oct 2007.

[73] Y. Urayama. bcbench, http://www.yagoto-urayama.jp/oshimaya/nbug/etc/bench/bcbench.html.

[74] VMware. http://www.vmware.com.

[75] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[76] Wikipedia. Dependency Hell, http://en.wikipedia.org/wiki/Dependency_hell.

[77] WINE-HQ. http://www.winehq.org.

[78] M. Xu, R. Bodik, and M. Hill. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, Jun 2003.

[79] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.