Record and Transplay: Partial Checkpointing for Replay Debugging

Dinesh Subhraveti*[1, 2] and Jason Nieh†[2]

[1] *IBM Almaden Research Center*
[2] *Columbia University*
Technical Report CUCS-050-09

Abstract

Software bugs that occur in production are often difficult to reproduce in the lab due to subtle differences in the application environment and nondeterminism. Toward addressing this problem, we present Transplay, a system that captures application software bugs as they occur in production and deterministically reproduces them in a completely different environment, potentially running a different operating system, where the application, its binaries and other support data do not exist. Transplay introduces *partial checkpointing*, a new mechanism that provides two key properties. It efficiently captures the minimal state necessary to reexecute just the last few moments of the application before it encountered a failure. The recorded state, which typically consists of a few megabytes of data, is used to replay the application without requiring the specific application binaries or the original execution environment. Transplay integrates with existing debuggers to provide facilities such as breakpoints and single-stepping to allow the user to examine the contents of variables and other program state at each source line of the application's replayed execution. We have implemented a Transplay prototype that can record unmodified Linux applications and replay them on different versions of Linux as well as Windows. Experiments with server applications such as the Apache web server show that Transplay can be used in production with modest recording overhead.

1 Introduction

When core business processes of a customer are suspended due to an application failure, nothing is more important to the application provider than to quickly diagnose the problem and put the customer back in business. Resolving a critical issue typically starts with reproducing the reported bug in the lab. Once the developer is able to reproduce the bug and examine the internal application state, the resolution follows quickly. However, reproducing a software bug is one of the most time consuming and difficult steps in the resolution of a problem.

Reproducibility of a bug is impacted by heterogeneity in the application environments. A variety of operating systems, corresponding libraries and their many versions, application tiers supplied by different ISVs, and network infrastructure with varied configuration settings make application environments complex and bugs hard to reproduce. The source of the problem might 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. Furthermore, nondeterministic factors such as timing and user inputs contribute to the difficulty in reproducing software bugs.

The common approach of conveying a bug report is often inadequate. Typically a bug report has to be followed up with several rounds of exchange between user and developer. Even if all the necessary data is somehow conveyed to the developer, accurately replicating the application environment in the lab is an error prone and tedious process. Some application vendors [6, 7] provide built-in support for collecting information when a failure occurs. Other sophisticated mechanisms [3] 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. Furthermore, indiscriminate recording and transfer of client data evokes privacy concerns.

To address these problems, we introduce *Transplay*, a software failure diagnosis tool that efficiently records application bugs as they occur in production and replays them across heterogeneous environments. By directly monitoring the application and capturing the bug as it occurs in production, the burden of repeated testing to reproduce the bug is removed. Transplay also side steps the probe effect problem by allowing its instrumentation to be enabled while the application is running in production. In the lab, there is no need to install or configure the original applica-

*dineshs@us.ibm.com
†nieh@cs.columbia.edu

tion, support libraries, or operating system to reproduce the failure. Portions of the application environment, including parts of application and library code necessary to reproduce the failure, are automatically recorded. No source code modifications, relinking, or other assistance from the application is expected.

Transplay performs two key functions. First, it transparently decouples the application from the underlying environment. Applications are decoupled from the operating system by recording system call results, then replaying the results instead of reexecuting system calls during replay. Applications are decoupled from installed binaries by recording specific code pages within executable files, then using the recorded pages during replay. Applications are decoupled from processor MMU structures such as segment descriptor tables by trapping and emulating the offending instructions during replay.

Second, Transplay introduces the notion of *partial checkpointing* to minimize the amount of data to be recorded, while ensuring that all information necessary to reproduce the failure is available. Based on the premise of short error propagation distances [31, 40, 38], Transplay captures the partial state of the application necessary for its execution during the last few hundreds of milliseconds prior to its failure. The resulting *partial checkpoint* is used to partially reconstruct the application at the developer site and allow it to run from an intermediate point prior to the failure to the point of failure. Rather than taking a complete application checkpoint [33, 16], which can require higher overhead and might 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 the failure. To also help reduce privacy concerns, partial checkpointing only allows the application to be deterministically replayed for the specific brief time interval; it cannot run live or replay before or after the specific time interval.

Transplay integrates with standard interactive debuggers to provide functionality such as setting breakpoints within a discrete interval of an application's execution, as recorded in a partial checkpoint. A developer can repeatedly reexecute and observe the last few moments of the application's execution before encountering a fault, and examine the contents of variables and other program state at each source line of replayed execution to expose the steps that the application took to reach the failure state.

We have implemented a Transplay prototype in the Linux kernel that can record application execution on one Linux system and replay it on a completely different Linux system based on a different distribution without any of the original application binaries or software libraries. We have also implemented a user-level replay tool for Windows using binary instrumentation that can replay Linux applications on Windows. We demonstrate Transplay's ability to record and replay execution across Linux and Windows operating systems using several real applications, including the Apache web server. Our measurements show that recording overhead is less than 15% on various real applications, and respective partial checkpoints consume only a couple of megabytes.

2 Usage Model

Transplay is a tool for recording and replaying specified intervals of an application's execution. Once Transplay is installed on the same machine as a production application, it continuously records its execution. When a fault occurs, Transplay outputs a set of partial checkpoints taken before the fault. When recording a multi-process application, partial checkpoints are saved separately for each process, along with information identifying the process that had the failure.

The recorded information can be then sent to the application developer, who can use Transplay to replay the partial checkpoints to reproduce the failure. Because partial checkpoints are per process, a developer does not need to replay the entire application. The developer could just select the process where the fault occurs to simplify problem diagnosis, and Transplay will replay just that process, with its interactions with other application processes virtualized.

Transplay integrates with the GNU Project Debugger (GDB) 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 the replay are captured into an output file and made available to the user. For instance, if the application writes into a socket, the user would be 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 to the system call. For root cause analysis, Transplay allows the programmer to set breakpoints at arbitrary functions or instructions, 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 [8] separately in a symbol file which would be accessible to developers.

3 Partial Checkpointing

Transplay introduces a novel notion of partial checkpointing. A partial checkpoint represents the partial state of the application necessary to replay the application's execution for a specified interval. Since the recording is only for a brief interval of time, the space needed to store the partial checkpoint can be small. Even though the application itself may be large in its memory footprint and processing large quantities of data, it only accesses a fraction of its state during a brief interval of time.

A partial checkpoint has four key characteristics. First, 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. Second, it is only useful for replaying the specific time interval, not for running the application normally. Third, it is captured over the specified time interval, not at a single point in time. Fourth, the state captured is completely decoupled from the underlying application binaries and the operating system.

Partial checkpointing significantly differs from conventional checkpointing. Unlike conventional checkpointing, partial checkpointing does not capture the complete state of an application, only what is needed by the application for a specified time interval. 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 OS resources, is not included in the partial checkpoint. It allows partial checkpointing to be implemented without intrusive kernel changes and enables a partial checkpoint to be replayed even on a different operating system. Unlike conventional checkpointing, partial checkpointing does not allow normal application execution to resume from a checkpoint. Unlike conventional checkpointing which captures all of its state at a given point in time, partial checkpointing captures its state across a time interval. Since it must know what the application will do in a given time interval to know what state to save, a partial checkpoint is captured over the entire time interval and is not fully saved until the end of the time interval. Because the state needed by an application in its future execution can be arbitrarily large, conventional 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 checkpointing also differs from incremental checkpointing [35]. While an incremental checkpoint tracks the pages written by the application since last checkpoint, partial checkpoint tracks the pages read by the application. Incremental checkpointing requires storing an initial full checkpoint followed by a series of increments. On the contrary, partial checkpointing ignores all prior application state and records only pages read by the application within the current recording interval.

Transplay's checkpointing approach significantly differs from virtual machine based snapshotting and logging. While VM snapshots provide portable replay, their space requirement is 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. On the other hand, since Transplay 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.

Transplay divides the recording of an application into periodic, contiguous time intervals. For each interval, it records a partial checkpoint for each application process that executes during that interval. 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 result in larger partial checkpoints. As the application executes, a series of partial checkpoints are generated and the most recent set of checkpoints is stored in a fixed size memory buffer. Storing a set of partial checkpoints rather than just the most recent one ensures that a certain minimum amount of execu-

tion context is available when a failure occurs. Partial checkpoints are maintained in memory to avoid disk I/O and minimize runtime overhead. 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.

Partial checkpoints maintained in memory can be written to disk at any time by stopping the current recording interval, causing the accumulated partial checkpoints in memory to be written to disk. 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, while a partial checkpoint 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.

If the application encounters a failure at the beginning of the recording interval, the most recent partial checkpoint may not contain sufficient context for root cause analysis. Transplay 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 the execution through the intermediate checkpoints, finally reaching the point of failure. The user can progressively go, as far back as necessary, within the available checkpoint set 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 the recording interval can be reached by rolling forward from the latest checkpoint prior to the desired execution point, thus simulating reverse execution.

Transplay uses a user space instrumentation frame work which provides the basic `start` and `stop` primitives that control partial checkpointing. The primitives are implemented by a *Transplay agent* program, which starts the application to be recorded, runs within the application context, and occupies a shared memory region mapped at an unused portion of its address space, [1] allowing for a quick transition of control from application code to the agent, when application events occur. The agent installs a common signal handler to all signals that the application may receive, and processes them first before forwarding them to the application. It allows Transplay to intercept exceptions caused by application failure, such as a segmentation violation or divide by zero, to cause checkpoints to be written to disk. It also enables an external process to communicate with the agent via a reserved signal to start and stop recording or to write checkpoints at any time based on an external fault sensor. Transplay agent tracks application's system calls by providing a simple kernel extension to the system call tracing component of `ptrace` which allows the originating thread itself to be signaled when a system call is made, along with any external debugger which may have been registered to receive the notifications. Within the signal handler context, the agent is able to process the system call by reading and writing to the thread's registers available on the signal stack, that point to system call arguments in Linux. The agent can emulate the system call, nullify it or process it in any other way, all in user space, before returning control to the application code. Further `ptrace` notifications are disabled while the agent code itself is running. The agent's shared memory region is uniformly mapped across all application processes at the same address, and the agent persists across `exec` system call by intercepting it and performing the `exec` operation in user space while retaining the region it is occupying. The memory region occupied by the Transplay agent is marked read-only and the application is not allowed to change the permission, so that potentially buggy application code does not accidentally corrupt the agent's memory.

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. The processor context marks the initial point of execution during replay. It consists of the state of the CPU/FPU registers and the per-thread state of the processor MMU such as descriptor entries in the segment descriptor tables. Register context is obtained from the signal stack and descriptor entries used by the thread are obtained through the API provided by the OS. On Linux, `get_thread_area` is used to read the GDT and `modify_ldt`, to read the LDT.

After recording the processor context, Transplay starts monitoring the application's interfaces to capture every input that crosses the application boundary. The inputs can be viewed as two forms of execution state. One is the initial set of accessed memory pages. The application code itself is considered an input, and if the application executes a particular function in a particular shared library, the specific code page(s) containing the function are recorded. Sec-

[1] On our Linux/x86 prototype, Transplay agent program is loaded at the address range `0x08000000 - 0x08031000`. Common Linux/x86 applications do not use addresses below `0x08048000`.

tion 4 describes how accesses to memory pages are tracked for constructing this initial state. The other is a continuous log of relevant application events, including a log of system call results, which is used to control the replay of the application. Section 5 describes how Transplay performs the logging. Memory state is recorded per application process and the log is recorded per thread. This results in a self-contained checkpoint of application's execution that can be used to replay the application with no dependency on the source environment. `stop` command concludes the recording interval and collects the logs generated during the execution. The processor context together with subsequently recorded application inputs within a recording interval forms the state necessary to deterministically replay the application's execution during that interval.

When an application is recorded, each thread within the application undergoes recording. Every 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 Transplay agent within respective processes. Transplay decouples individual processes from other processes whenever possible, so that each process can be independently replayed and debugged. For example, system call logging automatically decouples processes communicating via pipes or other interprocess communication mechanisms such as semaphores, message queues and file locks.

4 Tracking Memory Pages

To record a partial checkpoint for an application, Transplay must determine the memory pages that are read by the application during an interval of its execution. Transplay 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. Transplay cooperatively shares the use of these bits with the kernel by keeping track of kernel use of and changes to these bits by extending the macros used to read and manipulate these bits.

Only pages that were read during a recording interval are needed during replay. If the application only writes to a page, but does not read from it, such a page is not required during replay. However, accessed and dirty bits provided by most processors are not sufficient to determine if a written page has also been read since both reads and writes to a page set the accessed bit. We conservatively include all accessed pages in the partial checkpoint even though the application may not have read from some of them.

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, Transplay 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 process (*shadow process*) is created which shares all other resources with the parent except for 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. At the end of the recording interval, `stop` 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 and kills it. A new shadow process is created to track the original copies of the dirty pages for the next recording interval.

If one of the application processes forks a child process, the new child process is automatically placed within the purview of Transplay instrumentation. Transplay 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. For the same reason, Transplay also performs an implicit `start` immediately after performing the user space `exec` to initially load the application.

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, Transplay implements a version of incremental checkpointing adapted to suit the semantics of partial checkpointing. The algorithm consists of two parts. The first part is implemented as a part of the `stop` operation and the second part is implemented by the `write_checkpoints` routine, 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 elements 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

elements 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 recording interval, `stop` queries the kernel module to determine which pages in the process address space have been accessed. If a page was read, 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` 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 element 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 an element with matching `page_address` is found in a subsequent partial checkpoints 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 an element is not found, the current contents of memory at the address `page_address` is saved as the page data. Not finding a matching element implies that the current data at `page_address` was not subsequently modified and still valid. Similar processing is applied to each partial checkpoint in sequence.

4.1 Changes in Memory Region Geometry

If the application unmaps a region of memory during the course of a recording interval, the pages in that region that may have been accessed will be missed by the `stop` operation which is called only at the end of the interval, by which time, the memory region would not exist. Similarly if a new region is mapped during a recording interval, the shadow process would not contain the region. Transplay 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 `syscall_ record`s. Transplay keeps track of system calls that modify memory regions by also logging them to the set, `recent_maps`. The set indicates the system calls (`mmap, brk, exec`) which have mapped a memory region within a recording interval. The `recent_maps` set 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 `syscall_record` which maps the region encompassing the region being unmapped. If a matching syscall 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 `syscall_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`.

5 Logging

5.1 System Calls

To replay applications, Transplay must log system calls. For most system calls, Transplay simply records the system call return value and the return data con-

tained in system call parameters and provides it back to the application during replay. The data is captured as unstructured binary data and replayed as such. Transplay does not modify or otherwise attempt to interpret it. During application execution recording, each system call is logged as a `syscall_record` data structure. It contains the system call return value and data returned to the application through the system call parameters. Transplay agent maintains an ordered list of `syscall_records` for each thread of the host application process, for logging purpose.

Transplay uses a data plug-in which encodes the system call interface of the operating system, to record the system calls. Transplay consults the plug-in to determine which system call parameters carry data to be returned to the application and their sizes. For each system call, the plug-in encodes the following three pieces of information: 1. system call service number 2. the number and the sizes of system call parameters 3. whether a specific parameter may contain return data. In addition, the plug-in also indicates the system call interrupt vector and the calling convention. For instance, Linux/x86 and BSD/x86 use `int 0x80` or `sysenter` instruction 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 Transplay and partial checkpoints portable across different operating systems.

There are only two types of system calls which need additional processing beyond recording their results when recording application execution. Since partial checkpoints are per process, system calls for process control (`clone, fork, vfork, exit, exit_group`) need to allocate and deallocate state for recording partial checkpoints. Since partial checkpoints require tracking memory pages, system calls that deal with address space geometry (`mmap, munmap, brk, execve`) need additional processing as discussed in Section 4. In general, when the application invokes a system call that requires additional processing beyond mere log and replay, Transplay uses the native services available on the target operating system to process the system call. For instance, when the application calls `mmap`, an equivalent system call available on the target operating system is used to map the memory region.

5.2 Nondeterminism

Recording and replaying the system calls addresses most common sources of application nondeterminism. This 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, Transplay decouples this interprocess interaction by independently recording the system call results. During replay, the writers are returned 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 readers and writes. The same applies to synchronization and interprocess communication mechanisms such as semaphores and message queues, where processes interact through the system call interface.

Nondeterminism due to specific user input or external inputs processed by the application are also captured within the system calls. Nondeterminism due to hardware instructions such as `RDTSC` is handled through a trap and emulate mechanism. Linux provides a `prctl` interface to cause a SIGSEGV signal to be sent when `RDTSC` instruction is executed. The resulting SIGSEGV signal is intercepted by Transplay agent to emulate and record the instruction. Capturing nondeterminism due to concurrent accesses to shared memory is an important and difficult problem that requires an elaborate solution. We separately implemented a record and replay mechanism to efficiently capture concurrent accesses to shared memory, which we are currently integrating with Transplay.

6 Partial Replay

A partial checkpoint is self-contained and contains all data necessary to independently replay the application's execution for a specified interval. Applications alternate between user and kernel space execution, typically performing most of their work in user space, while delegating resource allocations and other privileged operations to the operating system kernel. The kernel parts of the 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 two classes of exceptions in which system calls must be reexecuted instead of just returning their results: system calls that modify the address space geometry (`mmap, munmap, brk, execve`) and the system calls related to processor's MMU context

(`set_thread_area, modify_ldt`). We discuss these in further detail below. Replaying the system call results is done in an OS independent way by Transplay agent on behalf of the application and hence the application never directly contacts the target operating system. As long as the application receives consistent responses to the system calls it makes, the application continues to run as expected. 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 OS.

To replay a piece of a previously recorded application, user first chooses a specific application process and an interval of execution to replay, by selecting a sequence of partial checkpoints from an available set of previously recorded partial checkpoints. All threads within the chosen process are resumed together. The memory address space is partially reconstructed with just the portions of state accessed by the application during that interval. The state required by the application 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 `syscall_record` would contain the set of pages to be loaded into memory at that point.

Transplay provides two alternative mechanisms to replay the application from a partial checkpoint. *Alternative 1:* A mechanism based on Transplay instrumentation, which is efficient but only applies to replay across Linux systems. *Alternative 2:* A second mechanism based on Pin [28], which is less efficient but allows a partial checkpoint to be replayed on Windows. Since speed is not a primary concern in offline interactive debugging, the replay mechanism based on Pin is useful even though it is relatively slow. Regardless of the mechanism, partial replay consists of two phases. 1. *Load phase*, where an initial set of memory pages (`initial_page_set`) in the consolidated partial checkpoint are loaded into memory. 2. *Reexecute phase*, where the application is deterministically reexecuted under the control of the instrumentation. The transition from load phase to reexecute phase occurs when control is transferred to the application code.

6.1 Partial Replay Across Linux

The load phase is performed by Transplay agent. As a reminder, Transplay agent is a statically linked program with an unconventional load address. Since Transplay agent is loaded at a region of address space which is not normally used by applications, it is able to load application's memory pages without overwriting itself. However, the address space used for the stack region is usually the same for all applications. To avoid potential memory conflict when restoring the target application's stack, Transplay allocates a separate region of memory to be used as its own stack. The new stack is put into effect, as the first step of the load phase, by loading the top address of the region into the stack pointer.

After restoring memory regions, Transplay agent creates the application threads contained in the process. Each new thread initializes itself and sends itself a reserved signal to restore the thread specific state. Within the signal handler, the thread restores the segment descriptors as saved in its respective log using `set_thread_area`, `modify_ldt` system calls. Then the processor context on the signal stack is replaced with that saved in the log. When the signal handler returns, the implicit `sigreturn` at the end of the signal handler loads the processor context, and control is directly transferred to the application code. If the log indicates that the execution involves shared memory interaction with other processes, all processes participating in the interaction are identified and started simultaneously upfront from their respective partial checkpoints. Each individual process is partially reconstructed as previously outlined. Transplay agent continues to monitor the application and ensures a deterministic replay until the execution reaches the end of the specified interval. Most application system calls are handled by returning the system call result from the log. If the application makes a system call that maps a new memory region into the process address space, the specific pages within the region that the application is going to access within the current recording interval are mapped and loaded in advance. If the application calls `clone` system call, Transplay creates a new thread, adds it to Transplay instrumentation and reads the respective log file to replay the system calls that it will subsequently make. If the application makes either `set_thread_area` or `modify_ldt` system calls, they are simply forwarded to the underlying kernel.

Transplay is able to replay a partial checkpoint on a different operating system distribution regardless of the environment and packages installed because the application binary pages are captured directly from the source system. For instance, Linux kernel automatically maps a virtual ELF shared object (VDSO) 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. 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.

6.2 Partial Replay on Windows

To demonstrate the effectiveness of Transplay approach to replay checkpoints across two completely different operating systems, we developed a replay mechanism based on Pin [28] binary instrumentation which can replay partial checkpoints of Linux applications on Windows. It also shows that partial replay can be implemented in multiple ways.

The load phase is performed by the Windows version of Transplay agent in user space using the Windows API. First, the address space of the application is partially reconstructed as outlined in alternative 1. Then, individual application threads are created. Each thread makes a special system call, which Pin instrumentation layer intercepts and invokes *Transplay Pintool* [28] to perform the reexecute phase of replay. Transplay 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 over control to the application code.

Transplay Pintool continues to monitor the application to satisfy the requests it makes during its execution. When it makes a Linux system call, the 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 particular, 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 alternative 1, except using the Windows API.

Application events related to the processor MMU are treated through a trap and emulate mechanism. Windows configures the CPU descriptor tables based on 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. Also, any attempts to update the Windows descriptor tables may result in a conflict with the way Windows uses its memory resources. Transplay 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. At any time during replay, Transplay maintains a table that maps the segment registers available to the user applications (`fs, gs`) to the base linear address of the segment that they currently point to. The table is populated by intercepting the `set_thread_area` and `modify_ldt` system calls and the `mov` instructions that load these segment registers during replay. The `set_thread_area` and `modify_ldt` system calls provide the mapping between the segment base address and the selector, while the `mov` instructions provides the mapping between the selector and the segment register. Subsequently, when the application executes an instruction that refers to a memory location through a segment register, the target instruction is rewritten such that it fetches the memory operand at the right offset relative to the base address of the segment pointed to by the segment register, as indicated by the table.

6.3 Partial Replay with GDB

Transplay integrates with GDB to provide debugging facilities during partial replay. Although the instrumentation used to monitor the application is based on a `ptrace` extension, it does not interfere with existing semantics of the `ptrace` interface. Any `ptrace` notifications destined to external processes continue to occur. As the application executes, the `ptrace` subsystem generates additional application events in the form of signals that notify Transplay agent of the application's events. Although these events are extraneous to the application, they do not perturb its execution. A simple GDB configuration script is provided to mask out these events to ensure transparency in debugging. The script also contains necessary GDB commands that load the appropriate symbol information and direct the replay process until the application is fully initialized for the user to start interacting through the debug interface.

The GDB configuration script begins the debugging session with the invocation of Transplay agent

itself as the debuggee. The agent reads the partial checkpoint file, reconstructs application's address space and initializes the processor context on the signal stack. The debugger doesn't intervene during this process. The latency of partially reconstructing the application from a partial checkpoint file is usually imperceptible to the user. After the application is loaded, the agent hits a preconfigured breakpoint at a special symbol placed immediately before turning over control to the application. A single forward step within the GDB script returns from the signal handler and into the application code.

At this point, the application is stopped within the debugger at a state few moments prior to its failure when it was recorded. The debugger shows the register state and the source line where the application is currently stopped. The user can then set break points, single step through the source lines to examine the intermediate values of program variables and monitor application's interactions with the operating system and other processes. Any inputs needed by the application are automatically provided by Transplay to preserve replay determinism. For instance, when the application attempts to read from the console, the input is directly provided from the partial checkpoint rather than waiting for user input. If the user wishes, he or she can also "step into" the system call to see the actions taken by Transplay on behalf of the user. System call instructions are often embedded within the system libraries and developers usually skip through these portions of code during normal debugging. Once the application returns from the system call, the perceived state of the application's registers and memory would be identical to its state at the corresponding point during recording.

7 Limitations and Extensions

Short error propagation distances: Not all failures may be reproduced by Transplay. Although reported as rare [31], the root cause of some failures may lie far in the past, outside the recent execution context recorded by Transplay.

Client privacy: While Transplay 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 web environment.

Application level bugs: Transplay 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. We are exploring the right interface points between Transplay and Qemu that would allow a guest kernel to be correctly and efficiently recorded and debugged.

Accurate system call specification: In order to accurately record and replay system call responses, Transplay requires an accurate representation of the system call API in the form of a data plug-in as described in Section 5.1. 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 [11], the memory side effects of those system calls may also be captured correctly.

Read-only debugging: Transplay 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.

Replay across different hardware architectures: Transplay currently requires that the source and target hardware architectures be the same. In order to provide replay across different architectures, we have done a preliminary integration between Transplay and Qemu-user [9]. Qemu-user is an ancillary component of Qemu [9] 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. We extended Qemu-user by providing a partial checkpoint loader which enables it to load a partial checkpoint file in addition to its built-in support for ELF binaries, and a system call replay mechanism that replaces Qemu-user's existing system call translation component when running a partial checkpoint. While this is currently work-in-progress, we have been able to replay simple Linux/x86 partial checkpoints on Linux/ppc hardware.

Application tracing: In addition to interactive debugging, Transplay can be used to efficiently generate fine-grain traces of applications running in production, which can be used for post-analysis, debugging or archival. While existing tracing tools can

provide fine-grain traces, they cannot be applied to production software due to their runtime cost. Also, storing execution traces as static data can consume large storage space and extracting relevant information through search can be difficult. A partial checkpoint, on the other hand, implicitly encodes the application state at each point of its execution and serves as a compact representation of an application trace. It allows a variety of traces such as system call and memory reference traces, to be derived offline by running replay through existing tracing tools. Relevant application state can be quickly accessed by setting breakpoints or watchpoints within the debugger. We are currently developing a Pin based tool to generate application traces from partial checkpoints.

8 Experimental Results

We have implemented Transplay as a prototype on Linux and Windows operating systems. Our Transplay prototype generates partial checkpoints of unmodified Linux applications and replays them on other Linux distributions running different versions of the kernel and libraries, and on Windows. The target Windows system we used reserves the top 2GB of its address space for kernel use. To avoid conflict with the Linux applications' use of address space on Windows, we configured the Linux kernel to limit user space allocations to the lower 2GB of address space while recording. Alternatively, Windows kernel can be configured to only use the top 1GB of the total 4GB address space by passing it the `/3GB` boot option.

The experimental setup consists of two identical machines, each with an Intel Core 2 Duo 2.4GHz processor and 2GB 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 runs a modified Linux-2.6.26 kernel with the `ptrace` extension and an interface to extract accessed and dirty bit information and the Fedora system runs a modified Linux-2.6.26 kernel with the `ptrace` extension to support Transplay's native instrumentation based replay.

The application scenarios evaluated in the experiments are listed in Table 1. 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. `apache` was configured to run with three processes and used shared memory. Even though it used shared memory, we did not notice any nondeterminism originating from it. We were able to correctly replay its partial checkpoints. For `apache` and `squid`, `httperf` benchmark [30] 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 64MB compressed file that decompresses to a 285MB 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, an artificial failure event was triggered during the benchmark by sending the application a `SIGSEGV` signal, so that Transplay 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 125ms to 4000ms 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.

We have also experimented with several real software bugs reported in Bugbench [26], to verify that our prototype can correctly capture and replay them. In each case we ran the faulty application on Ubuntu with Transplay enabled. The bug was triggered using specially crafted input. The same partial checkpoint produced by Transplay, representing the last 200 ms of the execution was then replayed on both Fedora and Windows. We were able to single-step through the source lines and examine the contents of various program variables at each step. For instance, the malformed URL request was apparent in the input buffer which caused the Squid proxy server to fail due to a heap overflow bug. While reproducing another bug in `bc`'s program parser, relevant code snippet of the bad input program that triggered a memory corruption in `bc` was captured in Transplay's log along with other events necessary for the bug to manifest. In each case, we verified that the execution trajectory is identical. In case of `gzip` and `bc` for example, we verified that the output text of `gzip` and the value of pi generated by `bc` matched in all cases. In general, any divergence would automatically surface during replay as an unexpected event which doesn't coincide with recorded log.

Figure 1 shows the normalized performance of four applications: `apache`, `squid`, `gzip` and `bcbench` [2] at recording intervals varying from 125ms to 4000ms. Squid showed the highest worst case overhead of 15% at 125ms 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.

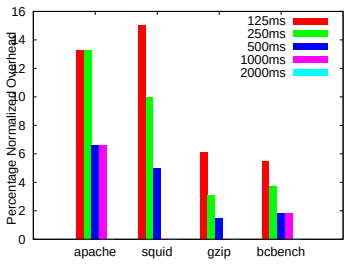

Figure 1: Performance Overhead

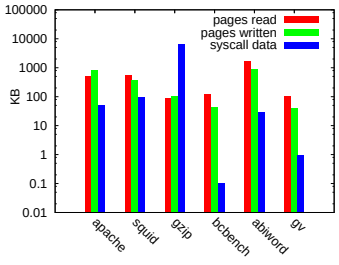

Figure 2: Space Breakdown (1 s)

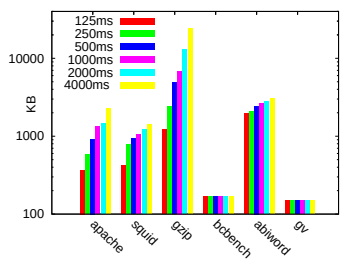

Figure 3: Recording Storage

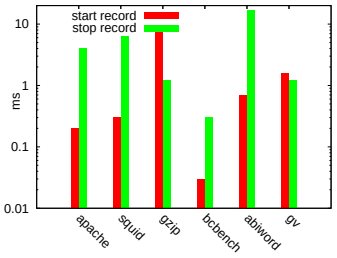

Figure 4: Recording Latency (1 s)

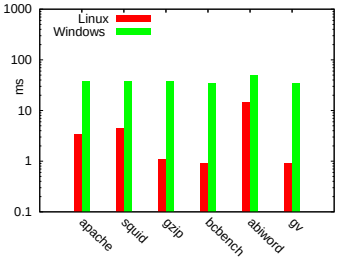

Figure 5: Resume Time

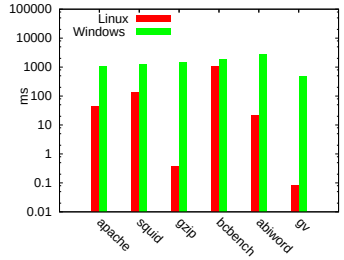

Figure 6: Reexecute Time

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 1: Application scenarios used for experiments

Figure 2 shows the storage space occupied by different constituents of a partial checkpoint 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 Transplay which enables the applications to be replayed without an equivalent install base at the target site. The large system call log shown by `gzip` accounts for the contents of the compressed file 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 3 shows the rate at which the total size of a partial checkpoint grows with the length of the recording interval.

Figure 4 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 Transplay 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 Transplay agent memory region. However, Transplay doesn't scan its own memory region when it checks the accessed and dirty bits, making `stop` relatively lighter.

Figures 5 and 6 show the times taken by resume and reexecute phases of the replay operation respectively on Fedora and Windows systems. As expected, the resume and reexecute times are far greater on Windows than on Linux. Transplay uses its native instrumentation mechanism to replay on Linux, which intercepts and replays the system calls efficiently. The large replay times on Windows is due to Pin's instruction-level instrumentation. We observe that the reexecute times for applications such as `apache` and `squid` were much smaller compared to their recording times. A partial checkpoint of a one second interval could be replayed in a few tens of milliseconds. This speedup is due to the fact that server applications spend most of their time in `poll` and `select` system calls. During replay, Transplay readily returns from these system calls without the wait.

9 Related Work

While interactive debugging tools [1] are helpful for analyzing bugs that can be easily reproduced, they do not assist with reproducing bugs. Techniques for

compile-time static checking [27, 12] and runtime dynamic checking [19, 4] 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.

Checkpointing has been a focus of extensive study. Checkpointing systems [34, 37] allow application state to be rolled back to a point in the past. Some of them [25, 39] 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 [32, 16, 21] 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 [16] and kernel intrusive [29], 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 to 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.

Optimizations such as incremental checkpointing [35] significantly improve checkpointing performance. However, maintaining a long series of incremental checkpoints corresponding to a lengthy failure-free operation can be expensive and unnecessary. Flashback [38] 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. 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. Triage [40] proposes a diagnosis protocol to automatically determine the root cause of a software failure in production. They repeatedly reexecute the failure triggering code to gain insight into the nature of the bug. While such a technique 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 generally not possible to automatically generate.

Execution replay systems [18, 36, 25, 33, 39] 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. Typically, replay is restricted to identically configured systems running the same operating system and they cannot handle discrepancies in the application environment in general. Due to high frequency of nondeterministic events, they produce large amounts of data, especially for long application runs. Some [13, 33] address shared resource nondeterminism by capturing the interactions among processes and replaying them. They require cooperation from the application and are nontransparent. R2 [18] requires the programmer to choose a high-level subroutine that completely encloses the program nondeterminism so that it can be used as a point of interception.

Virtual machines have been recently proposed [10, 23] as a debugging tool. Virtual machines, in general, provide the advantage of being operating system agnostic. However, due to the additional state introduced by the guest operating system and other processes which are not relevant to the application being debugged, virtual machine checkpointing is a relatively high overhead operation and requires large amount of storage space. Furthermore, the continuous runtime overhead imposed by virtual machines [24] may not be acceptable to some applications. Restoring a virtual machine involves restoring the complete OS state, its processes and potentially large secondary storage state. Tens of seconds or minutes may elapse between each step in an interactive debugging operation, making the process unnatural. The techniques we introduce in Transplay may be used to partially address this overhead. For instance, by applying partial checkpointing, the state accessed by the guest operating system during the last few seconds of its execution can be recorded, rather than the complete virtual machine state.

Extending legacy software through transparent instrumentation is a common approach to providing innovative and new functionality. Many methods of instrumentation have been developed. Some are implemented in the kernel [5, 20, 15], some in user space [17, 22] and some others [14] a combination of kernel and user space. User space approaches typically

intercept application's calls to the library functions. Most of them are only applicable to dynamically linked binaries and generally cannot prevent the application from bypassing the instrumentation. While kernel based instrumentation methods are more general and secure, they require significant extensions to the kernel and are often race prone.

10 Conclusions

Transplay is a software failure diagnosis tool which captures application bugs that occur in production and allows the recorded bugs to be deterministically reproduced again and again in a completely different environment, running a different operating system, without having to replicate the original setup or to do repeated testing. Transplay provides an innovative and efficient mechanism to record the complete state required to replay an application, including relevant pieces of its executable files, for a brief interval of time before its failure. 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. In order to provide this functionality, Transplay uses a novel instrumentation mechanism based on a simple kernel extension that decouples the application from its underlying operating system, the installed set of application binaries, and other CPU state which can conflict with the target system. Transplay introduces the notion of a partial checkpoint that represents the partial state of the application necessary to replay its execution for a specified interval. Partial checkpointing minimizes the amount of data to be recorded while ensuring that all information necessary to reproduce the bug is available. Transplay 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.

We demonstrate the effectiveness of Transplay approach through our prototype, which can capture partial checkpoints of unmodified Linux applications and deterministically replay them on other Linux distributions and on Windows. We have recorded several real-life software bugs using Transplay and in each case, Transplay captured the root cause of the failure and the necessary bug triggering data and events. With modest recording overhead, Transplay is able to generate partial checkpoints of several applications such as the Apache web server, and correctly replay them on Windows. Our evaluation of Transplay shows that it would be a valuable tool that can simplify the root cause analysis of production application failures.

References

[1] GDB, The GNU Debugger.

[2] http://www.yagoto-urayama.jp/~oshimaya/nbug/etc/bench/bcbench.html.

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

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

[5] Linux vserver project, linux vservers.

[6] Microsoft Corporation, Dr. Watson Overview.

[7] Mozilla.org, Quality Feedback Agent.

[8] The DWARF Debugging Standard, http://dwarfstd.org/.

[9] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] J. Chow, T. Garfinkel, and P. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[11] F. Cornelis, M. Ronsse, and K. D. Bosschere. Bosschere. tornado: A novel input replay tool. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 03), Las Vegas*, pages 1598–1604. CSREA Press, 2003.

[12] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, December 1994.

[13] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, New York, NY, USA, 1988. ACM.

[14] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *In Proc. Network and Distributed Systems Security Symposium*, pages 187–201, 2004.

[15] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. Slic: An extensibility system for commodity operating systems. In *In Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, 1998.

[16] C. Goater, D. Lezcano, C. Calmels, D. Hansen, S. Hallyn, and H. Franke. Making applications mobile under linux. In *Proceedings of 8th Linux Symposium*, 2006.

[17] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[18] 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 *In Proc. Operating Systems Development and Implementation (OSDI)*, 2008.

[19] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *The Winter Usenix*, 1992.

[20] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: A new approach to application security. In *In Proceedings of the 10th ACM SIGOPS European Workshop*. ACM, 2002.

[21] G. J. Janakiraman, J. R. 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*, pages 260–269, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M. B. Jones. Interposition agents: transparently interposing user code at the system interface. *SIGOPS Oper. Syst. Rev.*, 27(5):80–93, 1993.

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

[24] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[25] T. LeBlanc and J. MellorCrummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[26] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[27] M. Luján, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of java array bounds checks in the presence of indirection. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 76–85, New York, NY, USA, 2002. ACM.

[28] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 190–200. ACM Press, June 2005.

[29] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the transport. In *In HotNets II*, 2003.

[30] D. Mosberger and T. Jin. httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[31] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Operating System Design and Implementation*, 2002.

[33] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Not.*, 24(1):124–129, 1989.

[34] J. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, 1997.

[35] J. Plank, J. Xu, , and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing, technical report cs-95-302, 1995.

[36] Y. Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, 2005.

[37] 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 *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 300.2, Washington, DC, USA, 2005. IEEE Computer Society.

[38] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, 2004.

[39] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. *SIGSOFT Softw. Eng. Notes*, 25(5):158–167, 2000.

[40] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *Proceedings of SIGOPS symposium on operating systems principles*, 2007.