InSight: A Framework for Application Diagnosis using Virtual Machine Record and Replay

Senthilkumaran R and Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{kumaran, puru}@cse.iitb.ac.in

ABSTRACT

Non-deterministic execution poses several challenges toward diagnosis—debugging, profiling and execution state mining, of software systems (user-level applications and operating systems). While several techniques using modified libraries, library wrappers, binary instrumentation and memory shadowing techniques exist, we aim to exploit the record and replay technique enabled by virtualization to provide a generalized diagnosis framework. Our solution, is motivated by the requirements of reproducibility of execution, no application source code modification and no or minimal binary instrumentation overheads—all of which are seldom provided by existing techniques. `InSight` , our diagnosis framework, consists of two stages—the first which records execution state of applications and the virtual machine and the second that replays and analyses the recorded execution. We implement `InSight` on the Linux kernel-based virtual machine (KVM) platform and as contributions implement an efficient record and replay substrate for KVM and a diagnosis framework using this substrate. This paper describes the design and implementation of these components and develops a set of diagnosis tools—potential deadlock detection, lock usage profiling and function profiling. We also present experiments to demonstrate correctness, the low overheads of `InSight` and the related diagnosis outcomes.

Keywords

Virtual machine record and replay, application diagnosis

1. INTRODUCTION

An important activity for correct and efficient working of software systems—user-space applications and the operating system, is diagnosis. Diagnosis of different varieties—debugging, profiling and execution state mining, is often used for this purpose. Diagnosis services come with different levels of benefits (in terms of information for diagnosis) and costs incurred for the same.

Tools like `gdb` [1] and `gprof` [2] require binaries to be compiled with the debug information option for stateful stepped execution. Other tools like `ftrace` [3], `systemtap` [4] and `PinPlay` [5] require enabler scripts for extract information on desired events. Diagnosis tools like `valgrind` [6] use heavy-weight instrumentation to extract information and incur high overhead because of online diagnosis. Further, profiling tools like `OProfile` [7] exploit hardware counters to sample runtime execution state and need to be executed along with applications of interest. Each tool provides a subset of diagnosis functionalities and requires different degrees of execution overhead and user interventions (in terms of setting up the diagnosis environment). The "finer" the diagnosis, higher are the costs (in terms of execution time). For example, consider a write-protected memory region used to track all writes from processes/threads. Since, each memory write is trapped, applications can seldom execute at native speeds. Where as, for coarser diagnosis setups all desired information cannot be retrieved, e.g., profiling tools only provide quantitative analysis of execution state, call graphs etc. Another diagnosis requirement is to track multiple entities (process/threads) to capture a chronology of interactions between them, e.g., inter-process calls etc. Such requirements are usually met using fine grained trapped execution of all entities involved and potentially increasing the execution overheads.

Further, diagnosis tools are often required when the source of trouble is not known, e.g., bugs are non-deterministic, deadlock is dependent on inter-leaving order of threads etc. Runtime diagnosis using profiling and debuggers cannot capture conditions that do not occur. A useful requirement is to be able to log execution state of the system and, either periodically or when events of interest occur, treat it with varied diagnosis tools to uncover required information.

As part of this work, we aim to provide a framework for diagnosis of applications with the following aims—reproducibility of non-deterministic execution state, minimize execution overheads and instrumentation requirements. `InSight` , our diagnosis-as-a-service framework, exploits the record and replay feature available in virtualized environments. The framework provides a generic and quickly configurable service for diagnosis while minimizing interference to application execution. The virtual machine record feature, enables recording of the non-deterministic state of virtual machines—guest operating system and applications, to be replayed in a deterministic manner. The execution can be replayed several times for different types of diagnosis requirements. A sim-

ilar approach is used in Crosscut [8], where multi-stage replay is used to extract execution state for focussed time intervals and for selected processes. The extracted state can be replayed on a diagnosis tool valgrind or on a perl execution environment.

As part of the diagnosis framework, we instrument the guest operating system to coordinate with the host machine (the hypervisor) to exchange state information regarding events. An advantage of the modified guest is that the applications to be diagnosed need not worry about the basic instrumentation related to state collection and with a small amount of configuration can quickly use the diagnosis service. While guest instrumentation is required to provide event state information (e.g., instruction pointer to process mapping, symbol tables etc.), the generation of events itself relies on existing operating system facilities.

An important requirement of providing this service is minimal distortion—low overhead state collection and faithful reproduction of execution state for diagnosis. We exploit the record and replay facility with instrumentation to meet the above requirements and via InSight , our diagnosis-as-a-service framework, make the following contributions:

- Design and implement an optimized record and replay substrate for the Linux-based Kernel Virtual Machine (KVM) platform. Our substrate supports network traffic replay and always DMA-based disk accesses.
- Instrument the record and replay system to obtain reproducible state for diagnosis.
- Empirically validate the correctness and overheads of the diagnosis framework.
- Present a set of diagnosis tools, lock contention analyzing, potential deadlock detection and function-level profiling, to demonstrate the applicability and usability of InSight .

2. THE INSIGHT ARCHITECTURE AND IMPLEMENTATION

InSight consists of various components to achieve execution trace collection with reproducibility and negligible overhead/distortion. In this section, we describe various components of InSight and how these components work together. This section also describes the implementation of InSight components.

2.1 Architecture

As shown in Figure 1, InSight consists of two major components: (i) a record and replay enabled hypervisor, and (ii) a diagnosis subsystem. In coordination with the hypervisor, the virtual machine acts as a platform to execute applications and facilitates extraction of execution event trace. The diagnosis subsystem uses the event trace stored in an event database to answer interesting queries about the execution.

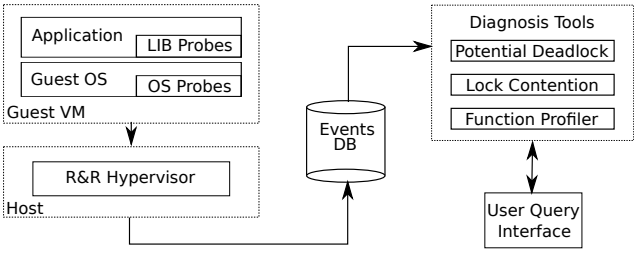

Figure 1: InSight architecture and it's components.

In the InSight framework, guest VMs execute in record mode—the application execution stage. In this stage no diagnosis information is collected, only information regarding external (non-deterministic) events which are necessary for reproducibility are recorded. At a later point in time, the recorded state of the virtual machine (and it's applications) can be replayed for diagnosis—the diagnosis stage. During replay, all external events are injected correctly for identical execution the applications and the OS. Further, event probes notify the InSight system about the occurrence of events The set of event probes we implemented is described in Section 3.

The InSight framework uses a instrumented guest virtual machine, i.e., the guest virtual machine stores and exposes state information regarding events to the hypervisor. For example, on a lock acquire event the virtual machine reveals the thread that acquired the lock. Since InSight and probes are aware of the pre-defined structure for information exchange, InSight collects the information prepared by event probes and stores them in an event database for diagnosis.

All state information is stored in an event specific pre-defined structure to be consumed by the hypervisor. During diagnosis, the hypervisor progressively captures state information of all events of interest and stores them in a events database. The events database is further consumed by diagnosis tools for analysis of the applications execution.

The InSight architecture allows multiple diagnosis stages to be executed in parallel enabling efficient analysis of long test scenarios. For example, data race analysis can be done in parallel with potential deadlock analysis.

2.2 Record and replay implementation

The first of the two major components of InSight is the virtual machine record and replay substrate. The record and replay feature enables reproducibility of a virtual machine's execution. InSight's virtual machine record and replay is designed to enable faithful play back and minimize logging overheads. The record and replay system follows standard implementation techniques of non-deterministic external event recording [9, 10]. As part of InSight we designed and implemented record and replay of network events and DMA-based disk IO. Based on our literature survey, we have not come across documentation of record and replay which considers both these factors.

Efficient record and replay implementation of DMA-based IO and network events is important for faithful execution of applications which are IO bound (both disk and network).
The `InSight` record and replay substrate is implemented on the Linux Kernel Virtual Machine (KVM). The substrate records non-deterministic external events such as data read from external devices (keyboard, mouse, disks), external interrupts injected to the virtual machine and data copied to VM's memory by external devices (DMA read, network packet receive). Replay is achieved by re-injecting these events to the guest virtual machine at the same instance at which the events occurred during record stage. A time stamp which serves as a reference for event injection is recorded with external events. We use the `<branch counter, instruction pointer, ECX value>` tuple as timestamp which identifies any point in execution as discussed in [9].
Next, we briefly describe our main contributions (as part of `InSight`) to the record and replay technique in the following sections.

2.2.1 Deterministic replay of network events

In order to enable stand-alone[1] record and replay for virtual machines with network connections, we record state regarding the incoming packets. The challenge in replaying arrival of network packets is that the packet consumption by guest is non-deterministic. A guest device driver operating in polling mode can consume a received packet any time after a complete-bit is set. The complete-bit indicates whether a packet is received (present) in the receive buffer.
To record state of the packets, we modified the Intel `eepro100` network device emulation in QEMU [11] The emulation is implemented based on the device specification provided in the [12].
During replay, to ensure that packets are read at the same time instance as during record, we associate setting of the complete bit (of a received packet) with a well-defined event of the guest. For example, setting the complete bit of a received packet is delayed by QEMU until a timestamped event (interrupts) or a guest VM event (port/mmio based IO event) is raised. Instead of setting the complete bit at packet reception, if the complete bit is set when a well-defined event occurs, we ensure that guest reads of received packets are associated with a recorded event.
The example in Figure 2 shows arrival of packets $\texttt{pkt}_1$, $\texttt{pkt}_2$, $\texttt{pkt}_3$ and $\texttt{pkt}_4$ during guest execution. But the complete bit of these packets is not set till a well defined event E. This technique creates a *happened-before* relationship between the well defined event and the set of packets received. In our example, the guest VM can see the packets only after it receives the event E (so the packets are indirectly tied to event E). During recording, every time a well defined event occurs, all buffered packets (i.e., packets received by the device but whose complete bit is not yet set) are recorded in a log file and their corresponding complete bit is set. In figure 2, contents of buffered packets $pkt_{1..4}$ are recorded during event E and the log record of E will have list of buffered packets—$pkt_{1..4}$—when E occurred. During replay, before injecting a well defined event, we check if any packets have been recorded along with that event, if any, we deliver those packets by copying them to receive buffers and setting their corresponding complete bits. In Figure2, when guest execution reaches time i in replay, the hypervisor should inject event E. Before injecting event E, hypervisor finds packets $\texttt{pkt}_{1..4}$ were recorded with event E and copies the recorded packet content to receive buffers and sets the complete bits before resuming guest execution. All three operations of saving/restoring packet content, setting complete bit and recording/injecting the well defined event occur as an atomic operation from the guest's perspective—no intermediate state is visible to guest.

[1] A virtual machine to be replayed does not require a network connection for application traffic

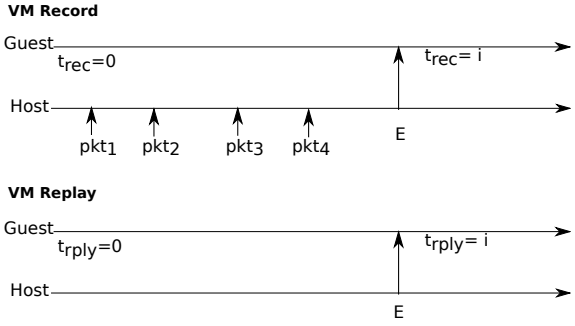

Figure 2: Enabling correct record and replay of network packets.

2.2.2 Replay of Direct Memory Access(DMA)-based disk I/O

Implementing efficient record and replay of disk accesses has two main challenges, (i) maintaining consistent disk state for replay, and (ii) supporting different methods of access, port-based or DMA-based IO. To address the first problem, we use a copy-on-write (COW) based disk format provided by QEMU. Recording is done using copy-on-write disk image and writes generated by the guest are redirected to a temporary disk file (preserving the state of the initial disk image). To replay, we use the same copy-on-write disk image which has the initial state of the disk and again redirect writes to a temporary a disk file. In both record and replay, reads after write are served from the temporary disk file. This method allows the initial disk state to be intact (both during record and during replay) and can be used for any number of replay executions.
To execute IO intensive applications we provide DMA-based disk access (as opposed to port based IO), and record and replay for the same. Our initial experiments showed that port based IO is considerably slow compared to DMA based IO and can not handle even moderate IO intensive applications. DMA-based disk accesses are asynchronous, i.e., disk

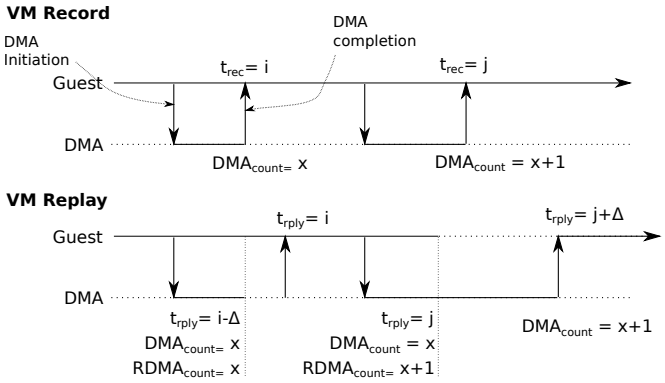

Figure 3: Record and replay implementation of DMA

access is initiated by the guest and completion is notified through an interrupt from the disk drive. In a record and replay scenario, DMA completion time— duration from DMA initiation to DMA completion interrupt arrival—should be identical in both the record and replay stages. If the completion times of DMA-based IO are not preserved, the replay will be unfaithful.

There are two possible scenarios during replay, the DMA-based disk access finishes earlier during replay than during record and vice-versa. To maintain identical ordering of events during replay, we delay the execution of the guest till the corresponding DMA interrupt arrives. The two scenarios are showed in Figure 3,

1. The DMA which completed at time i during record, completes earlier at time $i-\Delta$ during replay. In this case, the DMA completion interrupt is delayed till the replaying guest execution reaches time i.

2. The DMA completed at time j during record, but when the guest execution reaches time j during replay, the DMA is not yet complete. We pause the guest execution till the DMA completes (at time $j+\Delta$).

`InSight` relies on a DMA interrupt counter to achieve this. The counter represents the number DMA interrupts received so far. We maintain two counters, recorded count RDMA_{count} and actual count ADMA_{count}. Actual count, a VM state variable, is incremented whenever a DMA interrupt is raised by the emulated disk drive. Recorded count is a field in the log record. During record stage, every time a DMA interrupt is recorded, the recorded count for that DMA interrupt record is set to the current value of the actual count. During replay, before injecting a DMA interrupt from the log file, actual count and corresponding (interrupt record's) recorded count values are compared. If the actual count is less than the recorded count, then DMA is yet to complete. In this scenario, the guest VM process is added to a event interruptible wait queue by the host. When the counters match the guest process is removed from the wait queue and resumes execution.

The other possible case where actual count ADMA_{count} is greater than recorded count RDMA_{count} does not occur because the replay maintains determinism—no extra DMA's can be initiated.

3. THE INSIGHT DIAGNOSIS FRAMEWORK

.

In the `InSight` framework, application diagnosis is a two step process. In this section (and the next) we present design and implementation details and a set of use cases of `InSight` , the diagnosis-as-a-service framework. The framework itself is generic and can be applied to several diagnostic questions. By diagnosis we imply collection of state information during application execution to be used for debugging, profiling and execution analysis.

3.1 InSight events

Diagnosing tools of `InSight` work offline and rely on the event traces for analyzing applications. Event traces are generated by collecting key events about the application execution. Events are classified based on whether the information to be collected requires instrumentation for event generation and event information.

Explicit events need event probes, a code snippet which aids in notifying the event occurrence and event information extraction. For example, events such as lock acquisition, lock release require special instrumentation to generate events from the application's context. Event probes are a part of an application's execution. On the other hand implicit events can be extracted without any instrumentation of applications and their execution. Extraction of implicit events are facilitated by the virtualization layer (software and hardware). For example, occurrence of a memory access event on a specific memory address can be extracted using the additional level of page tables introduced by virtualization. Every event has two important requirements, instrumentation for event notification and information to be extracted on the event. For example, extracting lock events involves getting notifications during a lock event and extracting information related to address of the lock and identity of the thread acquiring the lock.

3.1.1 Event probes

Explicit events are those which capture specific events within the context of an applications execution. Event probes are inserted in application execution path to generate explicit events.

Our first method for event probes is based on Pin tools [13]. We used the Pin API to insert necessary event probes in application's executable. Pin does not require source code modifications but it uses JIT recompilation and runs the application in a virtual environment. Existing work [14] reports an average overhead of 20-30% for running applications in the Pin virtual environment (without event probes and related actions).

The second method to insert event probes works in cases

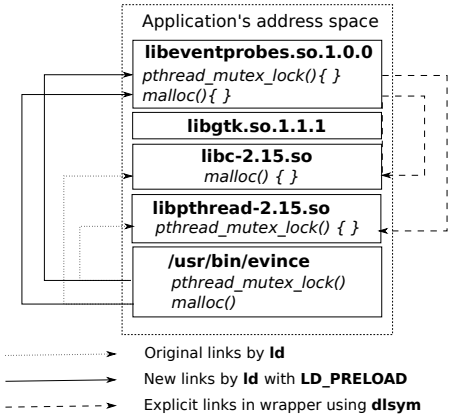

Figure 4: Wrappers built using LD_PRELOAD.

where events are to be generated from standard library functions and are defined in shared libraries. The basic idea is to create wrappers around the required library functions. Wrappers are created by using the LD_PRELOAD environment variable. This variable is used to specify a library to be loaded when an application is executed. The LD_PRELOAD library is load before any other shared library is loaded in the application's address space. Conventionally, the dynamic linker links a library function call by searching sequentially in all the loaded libraries and the first instance of the function definition found is linked. A wrapper for a library function can be implemented by using *chained linking*. To understand this, consider the scenario in Figure 3.1.1. Without the LD_PRELOAD wrapper, the `pthread_mutex_lock()` and `malloc()` calls inside the `evince` applications are linked to `libpthread-2.15.so` and `libc-2.15.so` respectively. With a LD_PRELOAD wrapper, these calls are linked to the wrapper functions defined in the `libeventprobes.so` library. Subsequently (after logging required event information), the wrapper function switches control to the intended function call. The address of the actual function definition is obtained using `dlsym()` function — given a function name returns the function definition address — of dynamic loader `ld`'s API.

Initial usecases of `InSight` use the LD_PRELOAD-based method for event probe insertion, but our framework is generic enough to accommodate other methods to capture user-level functions (albeit at greater instrumentation and runtime overheads). Our current implementation uses modified guest kernel for operating system related events. `systemtap` [4] can also be used to insert kernel-level event probes. `systemtap` allows embedding `C` code through guru mode[2], which can be used for event generation.

3.1.2 Event information and event buffer

Information about events is exchanged from event probes to the diagnosis system by using `event_buffer` in the guest application's address space. Event probes prepare required event information whenever an event occurs. By preparing primitive information about the events, overhead is kept in check. For example, a lock event consists of lock address, task's pid, lwpid (thread id) and timestamp of the lock request. These details can be extracted without incurring high overhead. To minimize overhead further, frequently used and non-changing information (pid, lwpid, process name) are cached in memory.

There are other subtle challenges in making event information available to the diagnosis system. The event information is in application's address space and addressed using guest virtual addresses (gva). Whereas the diagnosis system, which accesses the event information executes in the host address space and uses host virtual addresses (hva). Further, if the `event_buffer` (memory used for event information) spans across multiple pages, we need to do more than one address translations from `gva` to `hva`. In order to ensure that always one translation is sufficient, we use a page-aligned `event_buffer` in the guest (with size less than a page). In multi-threaded applications, events may occur in context of different threads. With a global event buffer, it possible for a thread to overwrite an existing event information prepared by another thread. To avoid such situations, we use thread specific `event_buffers` for event information.

3.1.3 Event notifications

The diagnosis system identifies event occurrences by receiving event notifications from event probes. To generate an `InSight` event , an event probe does following:

1. Prepare necessary information about the event in the `event_buffer`. The information consists of generic fields and event-specific fields.
2. Copy guest virtual address (`gva`) of the `event_buffer` to the `EAX` register.
3. Store the `InSight` event notification key EVENTNOTIFY in the `EBX` register.
4. Raise debug exception using `int3`.

The first two steps prepare event information while the last two steps generate the notification. Events are notified using a debug exception which is trapped by the hypervisor and processed by the diagnosis system. To distinguish event notification exceptions from other exceptions a special value EVENTNOTIFY is stored in the `EBX` register. Hypervisor handles event notifications differently during application execution stage and diagnosis stage.

3.2 Stage 1: Application execution

Application execution is the first stage of our `InSight` framework. An application executes within a record and replay enabled virtual machine. The hypervisor is configured to

[2]The guru mode is set by invoking `systemtap` with the **-g** option. When in guru mode, C code snippets enclosed between {% and %} are accepted and added to the probe by the `systemtap` translator.

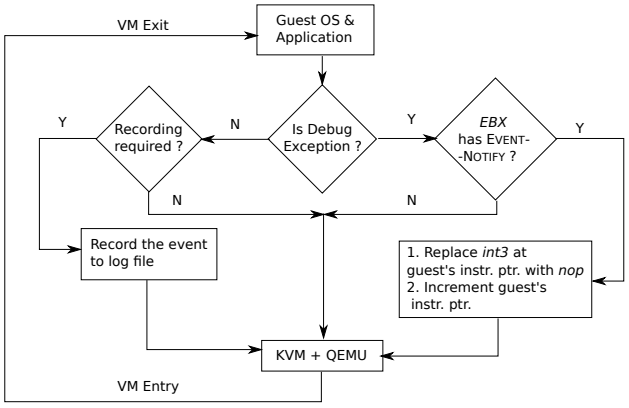

Figure 5: Events notification during VM Recording

operate the virtual machine in record mode, which enable recording all events of interest.

Application execution begins after inserting necessary event probes (both operating system and user-level probes). Figure 5 shows the VM recording process along with event notifications. During the application execution (VM recording) stage, event information and event notifications are not necessary, and application execution can make do without this overhead by avoiding raising these exceptions. To avoid trapping of event notifications, we could use either use hardware support to disable trapping all debug exceptions or use software instrumentation to disable trapping only debug exceptions which are raised for event notifications. We choose the later method, so that the normal debug exception handling is not changed. Figure 5 shows the method on a VM EXIT caused by a debug exception. The method avoids trapping event notifications by replacing the `int3` instruction used by the event probes with a `nop` instruction. The first step in replacing is to identify the address of the `int3` instructions which caused the debug exception. We allow the debug exceptions to occur for the first time. Every time a debug exception occurs, if the virtual machine is in record mode (application execution stage) and `EBX` register has EVENT-NOTIFY, we replace the instruction pointed by current instruction pointer with a `nop` instruction.

3.3 Stage 2: Application diagnosis

The diagnosis stage starts with VM replay, which replays previously recorded application execution. The deterministic replay feature of the VM ensures that the event probes will prepare the same event information and will raise the same set of explicit event notifications as in the record stage. Additionally, the diagnosis stage can generate implicit events while the VM replays the recorded execution. In contrast to application execution stage which ignored the event information and event notification, the diagnosis stage logs all event information. The only difference between application execution stage and application diagnosis stage is in the event notification method. During the diagnosis stage, `int3` instructions will not be replaced with `nop` instructions. Every time a debug exception occurs during replay, diagnosis system handles it as follows:

1. Check for `InSight` event identification key EVENT-NOTIFY in the `EBX` register.
2. If EVENTNOTIFY is not present in `EBX` register, follow the default handling of debug exception.
3. Else, convert the guest virtual address (`gva`) of `event_buffer` present in `EAX` to host virtual address.
4. Read the event information from the translated `hva` address and store the information in the event database.

The diagnosis system collects and saves the information in event database, and diagnosis tools can use the events stored for analysis. Our current implementation uses flat file as event database. We explain extraction of implicit events in function profiler's implementation section (see Section 4.4).

4. USECASES

In this section, we explain the prototype tools created for demonstrating the usefulness of our framework.

4.1 Events used by diagnosis tools

To demonstrate the usefulness of `InSight` we implemented tools which analyses for lock usage related issues and profiles application execution. Here we discuss information fields for the set of events generated to implement these tools. Each event type includes information specific to the event type. Along with event specific fields, for every event common fields such as `pid`, `lwpid`, process name and timestamp are extracted.

LOCK, TRYLOCK, TIMEDLOCK, UNLOCK, LOCKINIT events are generated from the corresponding `pthread_mutex_*` wrapper functions and the lock address represents the event-specific information. Similarly, {LOCK, TRYLOCK, TIMEDLOCK, UNLOCK}RET events are generated from the wrapper functions after the actual definition returns. In this case, the return value and lock address are the event-specific information fields. THREADCREATE and FORK events are generated from `pthread_create` and `fork` function wrappers respectively. THREADCREATE is generated from the newly created thread before executing the start routine. FORK event is generated from the child process created. Both these events include parent's `lwpid` information. CONTEXTSWITCH is generated from the kernel and includes `lwpid`'s of from and to processes. EXIT is generated either during `exit, _Exit, _exit` calls or from the wrapper library destructor —invoked while unloading libraries from the application address space. In the former case, EXIT includes return value of `exit` functions and 0 as return value in the later case. SYMBOL events are generated once per execution and are done so before the first explicit event. Event information in this case includes the address of `symbol_buffer` (see Section 4.1.1).

The SAMPLE (implicit) event is generated during application execution at every sample period `sample_interval` (see Section 4.4) and includes instruction pointer value as event information.

4.1.1 Symbol table extraction

For efficient diagnosis, results produced by the diagnosis system should include symbols from source code of the application being diagnosed. This helps user to map the issues found in application execution to application source code. To achieve this, the diagnosis system should be able to map raw addresses present in the stack and instruction pointers to symbols. Mapping addresses to symbols inside event probes while generating events incurs high overhead to application execution. We postpone this mapping till the diagnosis stage. During application execution stage, we generate a SYMBOL event which exposes symbol tables of the application. This event consists of a 4KB page boundary aligned `symbols_buffer`. The symbol tables are exposed by iteratively raising SYMBOL events and each event exposes part of the symbol table by copying it to `symbols_buffer`. Every time the `symbols_buffer` becomes full, a SYMBOL event is raised. Like other explicit events, information present in `symbols_buffer` is extracted and used only during diagnosis stage.

4.2 Lock contention analysis

Lock contention is an important performance metric in multi-threaded applications. Poorly designed thread co-ordination might result in huge loose of concurrency because of lock contention. As a result, it is useful to analyze the level of lock contention present in applications for different workloads characteristics. We implemented a Lock Contention Analyzer which uses event traces generated by `InSight` framework to analyze lock usage in a multi-thread application.

The tool's lock contention output is inspired by the `mutrace` tool [15], which provides *online* light weight lock contention analysis for pthread locks. Our lock contention analyzer operates on the events trace generated instead of running inside the application (as with `mutrace`). This has the advantage of incurring lesser overheads as well the same lock events could be used for analyzing other issues such as potential deadlocks.

Our tool uses the LOCK, TRYLOCK, TIMEDLOCK, UNLOCK, LOCKRET, TRYLOCKRET, TIMEDLOCKRET, UNLOCKRET events for identifying lock contention. With these events, we can simulate the execution of threads. A thread's state consists of the per-lock blocking state and the list of locks held by each thread. Each lock's state consists of the thread ID holding the lock, state of the lock (free or busy), list of threads waiting for the lock and previous thread that had acquired the lock if free. Though the state maintained provides enough information about lock usage, obtaining details about blocked lock requests is not straight forward. When multiple threads try to acquire the same lock it is not clear which thread acquires the lock. We employ an indirect method to identify blocked lock requests. For example, consider first three events of event trace as (LOCK `L`, T_1), (LOCK `L`, T_2) and (LOCK `L`, T_3), where `L` is a lock, T_i is thread which the tried to acquire lock `L` Among these three executed lock requests, only one of them doesn't block (assuming lock `L` is available before these events). But among the three threads, the thread T_j which acquired the lock will be known only when (LOCK_RET `L`, T_j) event occurs. In order to count number of threads which are blocked on a lock `L` acquired by thread T_j, we maintain a count of #threads who have issued a lock request but have not acquired the lock yet. On the LOCK_RET event, this count is used to estimate the number of blocked threads on the lock.

4.3 Potential-deadlock detection

With multi-threaded applications, due to infinite possibilities of thread interleaving, it is seldom possible to identify all potential bugs. These hard-to-catch bugs are caused by specific thread interleaving which might not have occurred during testing phase—one such bug is deadlock. We implemented a potential deadlock detection tool which uses the event trace collected by `InSight` and uses an existing potential deadlock algorithm [16].

Potential deadlock detection consists of three major steps, (i) Building a lock tree, (ii) Adding interleaving edges, and (iii) Running a modified depth first search (DFS) for cycle detection.

A lock tree is build for every thread. A lock tree of thread T, consists of several paths from the root R to every node N Every path from the root to any other node in the tree represents a stack (root node at the bottom of the stack) of locks held by the thread at some point of execution. A Lock tree can be easily built with lock and unlock events generated by the `InSight` framework. In other words, lock trees represent all possible hold scenarios in hold-wait deadlock condition. Once the lock trees are built, to represent potential wait-for relationship between threads, we add interleaving edges across lock trees. Interleaving edges between N and M are added if and only if nodes N and M represent the same lock and belong to different lock trees. In Figure 6, we use notation of A_i^T to represent a lock A acquired by thread T and suffix i is added to identify various instances of lock A in the lock tree of thread X. In Figure 6, node A_1^X and node A_1^Y represent lock A and acquired by X and Y respectively, so interleaving edges are added between them (note that suffixes do not matter).

4.3.1 InSight's potential-deadlock detector

Interleaving edges are normally bidirectional. Referring to Figure 6 an interleaving edge from node B_1^X to node B_1^Y represents a wait-for scenario where X waits on Y because thread X tried to acquire lock B held by thread Y. Similarly, the other direction of the interleaving edge represents Y waiting for X. In the `pthread` library, a subtlety exists—

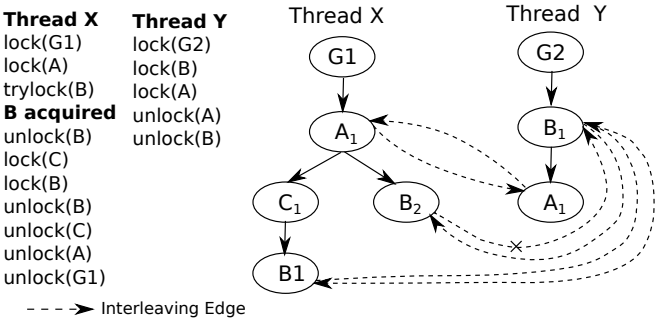

Figure 6: Wait-for graph for potential deadlock analysis

`trylock` does not block if the lock is not available, but threads can block for a lock acquired using `trylock`. Again referring to Figure 6, the edge $B_2^X \rightarrow B_1^Y$ is removed , because the node B_2^X represents a lock acquired by X using `trylock` which can't block on any thread. On the other hand, once the lock B is acquired by X using `trylock`, thread Y can block on X for lock B, so the edge $B_1^Y \rightarrow B_2^X$ is valid. Our `InSight` potential-deadlock detector is cognizant of such cases and builds lock trees in which uni-directional interleaving edges represent `trylock` based lock acquisition. Further, `pthread`'s `timedlock` is considered as a `trylock`. In case of `trylock` and `timedlock`, lock nodes are added to lock trees only if they acquire lock.

Our techniques uses the LOCK, TRYLOCK, TIMEDLOCK, LOCKRET, TRYLOCKRET, TIMEDLOCK, UNLOCK, UNLOCKRET events to build the lock tree (with interleaving edges). Next, we use a DFS-based algorithm [16] to detect valid lock cycles.

4.4 Function profiler

A function profiler provides function level statistics such as percentage of execution time spent in each function and helps in identifying hot spots in an application. We built a function profiler using the implicit event collection and address space identification features of the `InSight` framework. As mentioned earlier, implicit events are collected transparently from guest applications. In order to support implicit events, the host needs to know for which application implicit events are collected, address space information (to translate from guest virtual address to host virtual address) and symbol table details to resolve addresses. This section explains details of implicit event collection and its use for function profiling.

4.4.1 Application identification

As part of `InSight` , the host maintains a list of tasks being tracked—tasks for which profiling information is needed—in (`tracked_lwpids`) and the current task executing on the vcpu (`current_lwpid`) of interest. Each `lwpid` object in the `tracked_lwpid` consists of `lwpid`, `pid`, `counts_to_sample` (used for sampling) . A task is a process or a light weight thread (pthread) in the guest. These variables are updated when new tasks are created, when a tracked task exits and during context switch. To identify a new task creation, we use FORK and THREADCREATE events generated by a already tracked tasks. Note that tasks which are not tracked task will not generate these events. On FORK and THREADCREATE, the host updates `current_lwpid` to the `lwpid` of the current event and adds the `lwpid` to `tracked_lwpids`. This is valid because the FORK and THREADCREATE events are generated during execution of the newly created task. A task exit is identified by an EXIT event, and on this event the host removes all the tasks in the `tracket_lwpid` with `pid` as EXIT event's `pid`. Note that EXIT events are generated only from a processes inside the guest not by threads. The host also sets the `current_lwpid` to NULL. During the CONTEXTSWITCH event from task `P` to task `Q`, the host updates the `current_lwpid`. If the `Q` is in `tracked_lwpids`, then the host sets the `current_lwpid` to `Q`. Otherwise `current_lwpid` is set to NULL. The host to can now use `current_lwpid` to know which process is running on a vcpu at any time (to associate collected implicit events).

4.4.2 Sampling application execution

We sample an application's execution at regular intervals and collect the instruction pointer value at each sample. These samples are approximate measure of what part of code contributes how much to the application execution. A sample is recorded every SAMPLEINTERVAL CPU cycles. Note that this must consider only the CPU cycles spent by the corresponding tracked task. We use hardware performance counters and performance counter overflow for sampling [17]. The host maintains a variable `counts_to_sample`—number of CPU cycles to be executed before sampling the currently running task. This variable is updated during VM ENTRY, VM EXIT. Since we may be sampling more than one task at a time, the `counts_to_sample` variable has to be saved and restored during during CONTEXTSWITCH events. The following steps explain the sampling procedure.

During VM ENTRY,

- If `current_lwpid` is NULL, then we need not to sample the current task.
- If `current_lwpid` is not NULL, CPU cycles need to be counted. The performance counter is preset with a value such that it overflows after `counts_to_sample` CPU cycles. Counting after VM ENTRY is enabled in hardware.

During VM EXIT,

- If the counter has overflowed, create a SAMPLE event with with event information as current instruction pointer value and reset the `counts_to_sample` value to SAMPLEINTERVAL.
- Else, update the `counts_to_sample` by deducting the number of CPU cycles between VM ENTRY and VM EXIT.

Additionally,

- During FORK or THREADCREATE events, to start the first sampling of the new thread, the `counts_to_sample` variable is set to SAMPLEINTERVAL.

- During CONTEXTSWITCH event from task `P` to task `Q`, if `P` is in `tracked_lwpids`, then save the current value of `counts_to_sample` in `P`'s lwpid object, and if `Q` is in `tracked_lwpids`, then restore the value from `Q`'s lwpid object to `counts_to_lwpid`

Once samples are collected, a tool aggregates the collected samples for corresponding functions. The results are presented after mapping raw addresses to function names.

5. EXPERIMENTAL EVALUATION

In this section, we explain various experiments that we performed to evaluate the efficiency of our framework for application diagnosis. We focus on three major aspects as part of the evaluation, (i) empirically verify the correctness of record and replay substrate (ii) quantification of overheads, due to the record and replay substrate and event notifications and logging, and (iii) demonstration of usefulness of our framework and tools.

5.1 Record and replay correctness

An important feature of `InSight` is faithful re-execution of virtual machines to exploit the execution reproducibility feature. To verify correctness of `InSight` we performed the following tests:

- We ran the `top` command during the record session. During replay when the `top` command was re-executed, we compared the two outputs. All fields displayed as part of the `top` output were identical.

- We executed a CPU intensive workload (**Syn_cont**, described in Section 5.3) and compared various parameters of execution— execution time, number of context switches, minor and major page faults, number of IO waits and number of swapped pages. All parameters during record and replay stage of the application were identical.

- We also verified the correctness of `InSight` 's network packet recording and DMA recording implementation. For testing correct network replay, we setup two UDP clients sending packets to a receiver executing inside a virtual machine (which was recorded for replay). The receiver recorded all packets in to a file, and we verified that the checksum of the file is same during record and replay. The reason behind choosing UDP clients is to have non-deterministic mix of packets. For verifying DMA implementation, we created a file using `dd` command with input file as `/dev/urandom`. The checksum of the file was same during record and replay stage. This test also shows that the random number generator of the system is deterministic during replay.

Test Cases	**CPU Stress (W1)**		**RDTSC Stress (W2)**		**N/W Stress (W3)**	
	Time (secs)	Std. Dev.	Time (secs)	Std. Dev.	B/W (Mbps)	Std. Dev.
VM w/o Record	8.66	0.16	0.08	0.00	225.6	5.95
VM with Record	8.77	0.38	11.81	0.15	184.16	5.08

Table 1: Impact of virtual machine recording on performance.

5.2 Overheads related to recording

One of the design goals of `InSight` is to introduce minimize overheads during the record stage, a property ensures that record stage execution resembles real world execution. We conducted experiments to determine the overheads during the record stage. The system used for these experiments had the following characteristics: Intel Core i5 processor (2.8 GHz with 4 cores, 4 GB RAM). The virtual machine was allocated 1 vcpu and 1.5 GB of RAM. The guest and host kernels were identical—Linux 2.6.38.8 and the host used QEMU 0.14.0 for device emulation.

Our experiments were focused on measuring completion time of benchmarks and impact on network utilization. We tried to measure the impact on disk IO, but due to multiple levels of page cache (at host and at guest) we could not get any conclusive results.

Three different workloads were used to estimate the overheads, (i) **W1:** 10 iterations of a program with CPU intensive computation (computing arctangent value with scale of 3000 using *bc* utility), (ii) **W2:** 10 iterations of a program which executes 10 million `rdtsc` instructions, and (iii) **W3:** 5 iterations of a program which transfers 100 MB from host to guest.

Table 1 reports results of our experiments. For the CPU intensive workload **W1** the recording overheads are negligible (less than 1%), all executions finish in a duration of close to 8.7 seconds. We use software emulation for **rdtsc** instructions for enabling record and replay. Overheads introduced by this trap-and-emulate model is evaluated by the **W2** workload. Over 10 million such calls, the overhead of execution shows a drastic slow down (by about 150x). Our workload is an extreme scenario and in reality this overhead is amortized over a long period of time (usually 1 million **rdtsc** instructions are spread across 10 minutes of execution). Workload **W3** is used to quantify the impact on network utilization. The decrease in achieved network bandwidth is around 22%. The decrease is due to recording packet contents (for replay) which requires multiple file writes every second. Also, for scenarios where end points of network

connections are across machines (or networks) we envision the impact of record overheads to be lesser.

5.3 Diagnosis overheads

We used following workloads to evaluate the `InSight` framework. The first two workloads are `pthread`-library based and the rest have a single thread of execution.

- **RUBiS** is an auction site prototype which uses a web server and a database server. We used Apache the web server with the thread based multi-processing module (`httpd.worker`). Threads-based Apache was chosen to demonstrate the overhead characteristics of an application involving several thread-based events. The web server executed inside a virtual machine and all processes of the web server were diagnosed. The database server and the client executed on different physical machines.
- **Syn_cont** is a synthetic workload to simulate lock contention and had sixteen threads, each executing the following code.

```
void start_thread(int id) {
  for(i=0;i<1000;i++) {
    comp(); lock(level1[id/2]); comp()
        ; unlock(level1[id/2]);
    comp(); lock(level2[id/8]); comp()
        ; unlock(level2[id/8]);
  }
}
```

  `comp()` is CPU intensive function which simulates per-thread activity. The **Syn_cont** workload had two levels of lock contention, `level1` locks are shared between two threads and `level2` locks are shared between eight threads.
- The **sort** workload lexicographically sorts lines of 750 MB ASCII file. The input file is stored in the guest's local storage.
- The **awk** workload uses an associative array to count number of records in the above mentioned file.
- The **sed** workload substituted a specific pattern of string in the above mention file. String matching is done using regular expression.

The following results to show how application characteristics affect the replay speed and the events database size. Table 2 presents the results of execution time and event database size of the workloads under different execution scenarios—without recording, with recording and replay with explicit and implicit events capture. The implicit sampling event, generate a `SAMPLE` event every X CPU cycles (denoted as Replay$X$ in Table 2). The execution time was measured from VM boot to VM shutdown. The **sort** and **Syn_cont** workloads repeated execution five times each before VM shutdown and the **RUBiS** workload served 10500 requests before shutdown.

Workload	Scenario	Execution Time (secs)	Events DB Size (MB)
sort	W/o Record	1346	NA
	W/ Record	1477	NA
	Replay (∞)	1401	5.8
	Replay (800K)	1551	165
	Replay (400K)	1635	324
	Replay (100K)	2394	1279
RUBiS	W/o Record	476	NA
	W/ Record	481	NA
	Replay (∞)	235	38
	Replay (800K)	244	39
	Replay (400K)	274	40
	Replay(100K)	249	46
Syn_cont	W/o Record	886	NA
	W/ Record	883	NA
	Replay (∞)	941	31
	Replay (800K)	998	120
	Replay (400K)	1070	208
	Replay (100K)	1321	737

Table 2: Replay speed and events database size for different workloads and event types. Replay (X) denotes, an implicit SAMPLE event once every X CPU cycles.

The results verify that replay speed is inversely proportional to sample rate and events database size is dominated by SAMPLE events. The **sort** workload is CPU and IO intensive and did not generate any lock events. A high implicit events sampling rate (1 sample per 100K cycles) generated a 1279 MB event database—99% of which were theSAMPLE events. When the implicit sample events were disabled, size of the events database was only 5.8 MB. Varying the sampling interval from 100K to 800K cycles per sample, varied the replay time of the **sort** workload reduced from ≈40 mins to ≈26 mins. The **Syn_cont** workload is CPU intensive and similar results as the **sort** workload were observed.

When the workload is not CPU intensive, the replay speed is higher than the record speed. This is due to skipping of the `halt` instructions during replay—CPU no longer waits for external interrupts as they are served from replay log. The **RUBiS** workload had only 2% of CPU utilization and mostly waited for database replies. Since the database replies are served from replay log, the replay speed was faster by a factor of 2. The **RUBiS** workloads CPU utilization was very low, hence the number of SAMPLE events generated was lower as well—32.5% of 46 MB—and increasing the sample interval had very negligible impact on replay speed and the events database size.

5.4 Diagnosis results

In this section, we present the results which showcase the usefulness and correctness of our diagnosing tools.

5.4.1 Lock usage analysis

To evaluate the usefulness of our lock usage tools, we analyzed the event traces of **Sys_cont** and the **RUBiS** workload. Table 3 summarizes the results achieved. For **Syn_cont**, the values are average of 5 runs and in each run the values are

Workload / Lock Description	Total reqs.	% of blocks	% of owners--hip changes
Syn_cont / Level 1 locks	2000	6.32%	71.46 %
Syn_cont / Level 2 locks	8000	73.66 %	78.85 %
RUBiS / queue->one_big_mutex	126	0 %	97.46 %
RUBiS /apr_pool_t->apr_thread_mutex_t	129	0 %	21.66 %

Table 3: Lock usage analysis with test workloads.

average of locks of same level. For **RUBiS**, the values are average of 5 runs and in each run the values are average of 5 thread groups created by Apache.

As expected, the lock usage tool identified the two level-2 locks present in the **Syn_cont** workload as most contented (measured by number of blocks and % of blocks) and most shared (measured by number of ownership change between two consecutive lock acquisition). With the **Syn_cont** workload, contention and ownership changes of the level 2 locks was higher, 10x more blocking on the two locks. We verified this by using the same workload with the **mutrace** tool [15], which reported similar numbers. For the **RUBiS** workload, there were no lock contentions but only ownership changes. We matched the lock addresses with the source code of Apache 2.4 and located the top two locks (in terms of number of ownership changes).

5.4.2 Function profiler

This section presents the results of the function profiling diagnosis tools. We compared our results with the `perf` [18] tool for validation. We execute a sample set of applications inside the `InSight` framework for profiling. `perf` was executed with CPU cycles as event on the host machine with the same applications (the KVM version we used for implementation does not support virtualized of hardware counters). The results shown in Table 4 are a consolidated view of five different runs with different applications. We present the rank range and average of % of sample for the top 4 functions with `InSight` based and `perf`-based function profiling. Our results are very consistent with `perf` results. The accuracy is achieved by having far more samples than the default configuration of `perf`—`InSight` collected approximately 14 to 34 times more samples with 100K CPU cycles as sample interval. `InSight` profiler has reported per-thread level statistics for **Syn_cont** and **RUBiS**. These per-thread level reports are more useful compared to `perf`'s reports which are consolidated over all threads.

5.4.3 Potential deadlock analysis

This sections presents the experiments done to demonstrate use of the potential deadlock analysis tool. Finding a potential deadlock in a matured program is a difficult task. We tried various pthread-based applications (evince, gedit, vlc, and httpd), but we could not find any potential deadlock. We developed the following sample applications with various potential deadlock scenarios to validate the results from our tool. These sample applications have the potential for deadlock but deadlocks do not occur during execution (we introduced sleep between locking events to serialize the threads).

```
Application 1: Trylock
T1(){
        if(trylock(L1) == acquired){
        lock(L2); unlock(L2); unlock(L1);
            } }
T2() { lock(L2); lock(L1); unlock(L1);
    unlock(L2); }

Application 2: Gatelock
T1() {  lock(Gate); lock(L1); lock(L2);
        unlock(L2); unlock(L1); unlock(
            Gate); }
T2() {  lock(Gate); lock(L2); lock(L1);
        unlock(L1); unlock(L2); unlock(
            Gate); }
```

For these two test application scenarios, our tool reported correct results. The potential deadlock tool identified deadlocks in the `Trylock` and `Transitive` applications. The tool did not report any deadlock in `Gatelock` application, but identified that a potential deadlock involving lock `L1` and `L2` was protected by lock `Gate`. The tool reported following cycles (names are changed to match the above examples.

- $L2_{T2} \rightarrow L1_{T2} \rightarrow L1_{T1} \rightarrow L2_{T1} \rightarrow L2_{T2}$, this cycle is reported for `Trylock` example. In this case we made sure that the `trylock` succeeds to ensure the potential deadlock. When the `trylock` call failed, no deadlock was reported.
- For the `Gatelock` application, the tool reported that the potential cycle $L2_{T2} \rightarrow L1_{T2} \rightarrow L1_{T1} \rightarrow L2_{T1} \rightarrow L2_{T2}$ as protected by lock `Gate`.

The potential deadlock analysis of these application is based on the events related to locks that were captured during the diagnosis stage and the above experiment demonstrates their usefulness.

6. RELATED WORK

In this section, we briefly discuss work related to deterministic replay and work which leverages deterministic replay for efficient application debugging and analysis. Various virtual machine record and replay solutions have been discussed in [9, 10, 19]. `InSight`'s record and replay implementation improves on these to provide efficient DMA and network traffic handling. These solutions, and `InSight` , are applicable in uniprocessor scenarios whereas solutions such as DoublePlay[20] and [21] have extended record and replay to multi-processor scenarios.

Crosscut [8] uses virtual machine record and replay to extract process level replay logs which can be replayed with Valgrind [6] and other diagnosing tools. Crosscut adds reproducibility to Valgrind, but this solution fails to handle

Application	Function name	InSight ranks (min - max)	perf ranks (min - max)	InSight % usage (average)	perf % usage (average)
awk	r_interpret	1 - 1	1 - 1	22.56	24.9
	__GI____strtod_l_internal	2 - 2	2 - 2	8.75	6.99
	r_dupnode	3 - 5	5 - 8	5.06	3.76
	rs1scan	3 - 5	3 - 4	4.76	4.48
sort	strcoll_l	1 - 1	1 - 1	85.26	77.16
	__memcmp_sse2	2 - 2	2 - 2	4.81	6.67
	__strlen_sse2	3 - 3	3 - 4	1.99	2.40
	xmemcoll0	4 - 4	5 - 5	1.70	1.78
sed	re_search_internal	1 - 1	1 - 1	28.81	28.90
	mbrtowc	2 - 2	2 - 2	18.89	15.83
	__gconv_transform_utf8_internal	3 - 3	3 - 3	12.14	13.13
	builtd_wcs_buffer	4 - 4	4 - 4	8.03	8.22

Table 4: Comparison of function profiling using `InSight` and `perf`.

multi-process applications and to harness hardware counters for profiling. For example, Crosscut solution can not be used to identify potential deadlocks with process level semaphores and race condition in shared memory segments. With `InSight` , tools to diagnose these issues can be developed if semaphore and shared memory access events are generated. X-ray [14] employed a two-stage diagnosing method using the record and replay feature. X-ray aims at identifying performance issues and tracking the root causes of these issues. X-ray uses latency, CPU usage, file system usage etc, as performance metrics. X-ray relies on taint analysis for providing performance summarization on the chosen performance metric. For example, while an input request is processed by the application, X-ray assigns performance cost for each basic block of the code through which the request is processed. Later it summarizes the assigned costs to identify the dominant root cause over basic blocks. X-ray can be extended to identify lock contention and to provide profiling information, but it can not identify potential deadlocks and race conditions. A similar solution can be implemented on the `InSight` framework. For example, X-ray's solution of assigning costs to basic blocks can be built by inserting event probes for system calls and identifying basic block executions by generating implicit events.

Valgrind [6]—a heavyweight online diagnosing tool (designed for multi-threaded applications) instruments the application code and analyzes the execution online. Valgrind lacks reproducibility and incurs high overhead to application execution. Valgrind's analysis is confined to the address space of a process, it cannot debug multi-process applications. Further, Valgrind reimplements the `pthread` library to obtain control on execution of thread. This restricts the generalization of Valgrind framework, whereas in `InSight` , extending support to other multi-thread libraries is easier—a set of wrapper calls can be pre-loaded for generating events. Tralfamadore [23] collects low-level traces about operating system execution and answers interesting queries to help understand the execution flow. Tralfamadore also works online incurring high overheads and since operating system events are logged application-level diagnosis is not possible. `InSight` on the other hand is offline, is application-aware (meaning extracts process/thread context) and can replay a recorded session several times for extraction of event over several iterations. PinPlay [5] provides repayable debugging framework which uses Pin [13] instrumentation to enable record and replay facility for parallel programs. For debugging, PinPlay uses the GDB [1] interface through remote stub protocol of GDB. PinPlay provides reproducibility, but lacks support for diagnosing tools provided by `InSight` .

7. CONCLUSIONS AND FUTURE WORK

Efficient diagnosis of multi-threaded and multi-process applications are challenging and `InSight` provides a extensible framework which can be used to build diagnosis tools. We have implemented `InSight` —a lightweight offline diagnosing framework which uses record and replay technique. `InSight` framework has important properties of reproducibility and minimum overhead without any compromise in efficient diagnosing. `InSight` tools are built to demonstrate the efficiency and the usefulness of our framework. We conducted various experiments to measure the overhead introduced by `InSight` , the obtained results are accurate and showed that `InSight` incurs less overhead to application execution.

Our current `InSight` tools are designed for multi-thread related issues such as lock contention and potential deadlock detection. The principles used in these tools can be extended to multi-process related applications—especially inter-process communication related issues. `InSight` 's current implementation does not provide facilities to identify memory related issues such as races conditions, uninitialized memory reads and heap profiling which are addressed by Valgrind[6]. Techniques discussed in [24] can be applied in `InSight` framework with the help of implicit events. Implicit events related to memory accesses can be generated using additional page table present in virtualization layer. Tools can then use theses memory access events to identify race conditions in multi-thread and multi-process environment.

8. REFERENCES

[1] "GDB: The GNU Project Debugger." http://www.gnu.org/software/gdb/.
[2] "GNU gprof."
[3] "Ftrace." http://elinux.org/Ftrace.
[4] "Systemtap." http://www.redhat.com/magazine/011sep05/features/systemtap/.
[5] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in Proceedings of CGO, pp. 2–11, 2010.
[6] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in Proceedings of PLDI, pp. 89–100, 2007.
[7] "Oprofile." http://oprofile.sourceforge.net/news/.
[8] 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 VEE, pp. 13–24, 2010.
[9] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc, "Retrace: Collecting execution trace with virtual machine deterministic replay," in Proceedings of MoBS, 2007.
[10] K. KE and M. LE, "A kvm-based logging and replay system for debugging non-deterministic executions," in Proceedings of Cloud Computing and Virtualization, pp. 270–277, 2010.
[11] "Kernel Based Virtual Machine." http://www.linux-kvm.org/page/Main_Page.
[12] Intel Corporation, Intel 8255x 10/100 Mbps Ethernet Controller Family, 2006.
[13] "Pin - a dynamic binary instrumentation tool." http://www.pintool.org/.
[14] M. Attariyan, M. Chow, and J. Flinn, "X-ray: automating root-cause diagnosis of performance anomalies in production software," in Proceedings of OSDI, pp. 307–320, 2012.
[15] "Mutrace: Lock Contention ." http://0pointer.de/blog/projects/mutrace.html.
[16] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in Proceedings of PADTAD, pp. 51–60, 2006.
[17] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A,3B and 3C, August 2007.
[18] "Perf: Linux profiling ." https://perf.wiki.kernel.org/index.php/Main_Page.
[19] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in Proceedings of OSDI, pp. 211–224, 2002.
[20] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," in Proceedings of ASPLOS, pp. 15–26, 2011.
[21] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in Proceedings of VEE, pp. 121–130, 2008.
[22] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in Proceedings of HPDC, pp. 101–110, 2009.
[23] G. Lefebvre, B. Cully, C. Head, M. Spear, N. Hutchinson, M. Feeley, and A. Warfield, "Execution mining," in Proceedings of VEE, pp. 145–158, 2012.
[24] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in Proceedings of VEE, pp. 65–74, 2007.
[25] "KVM VirtIO." http://www.linux-kvm.org/page/Virtio.