On the Existence of Probe Effect in Multi-threaded Embedded Programs

Young Wn Song and Yann-Hang Lee

Computer Science and Engineering
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, U.S.A.
Tempe, AZ

***Abstract* —** Software instrumentation has been a convenient and portable approach for dynamic analysis, debugging, or profiling of program execution. Unfortunately, instrumentation may change the temporal behavior of multi-threaded program execution and result in different ordering of thread operations, which is called probe effect. While the approaches to reduce instrumentation overhead, to enable reproducible execution, and to enforce deterministic threading have been studied, no research has yet answered if an instrumented execution has the same behavior as the program execution without any instrumentation and how the execution gets changed if there were any. In this paper, we propose a simulation-based analysis to detect the changes of execution event ordering that are induced by instrumentation operations. The execution model of a program is constructed from the trace of instrumented program execution and is used in a simulation analysis where instrumentation overhead is removed. As a consequence, we can infer the ordering of events in the original program execution and verify the existence of probe effect resulted from instrumentation.

***Keywords* -** *multi-threaded program; event ordering; reproducible execution; probe effect; profiling; simulation.*

1 INTRODUCTION

In real-time embedded systems, application tasks usually run in concurrent threads. Threads may interrelate with each other as they share resource and data. They also interact with the external environment to receive sensor data and to control actuators. While the threads are running, any instrumentation to observe program execution behavior will introduce extra overhead to the execution. Instrumentation overhead, no matter how small it is, may intrude and change the execution behavior of the program and, consequently, introduce probe effect [1][2]. Hence the observed behavior through instrumentation is not guaranteed to represent the original program behavior.

Instrumentation operations can perturb program execution in two ways. First, the occurrence of an execution event is delayed by the amount of time spent on running instrumented code. This can change the time of interacting with other threads and external environment (e.g. reading an input). Second, due to the changes of the timing of invoking guarded resources and critical sections, scheduling decision can be different. This, in turn, can lead to the variations in the sequential order of accessing shared resources. Therefore, the timing perturbation by instrumented code can result in a different happen-before ordering of events [3] and possibly a different program execution path from the original program. The other related issue is that we may not be able to know whether there is any change on program execution paths caused by the timing perturbation.

To obtain a proper observation of multi-threaded program execution through instrumentation, it is critical to know the effect of timing perturbation. However, unless we adopt hardware based monitoring, it might not be possible to know the exact execution behavior of a program. It may be argued that a comparison of computation results from instrumented and un-instrumented programs can reveal any different behavior of program execution. Note that some benign program behavior may not affect the final computation results, for instance, a branch decision can be caused by different conditions in a compound conditional expression. In addition, for embedded systems, it may be tricky to manage identical external inputs arriving at the precise instants of the execution. Another approximation is to measure the overhead of instrumentation, calculate the execution time by removing the overhead, and infer the real execution. In a single thread program, this would be feasible. However, in multi-threaded programs it is extremely difficult to consider all thread interactions when thread execution time is changed. Moreover, it is problematic to take into account kernel states (e.g. run queue state) which may affect scheduling decision.

There have been several approaches to recover the performance of parallel programs by compensating the instrumentation overhead [4][5], but they do not examine the ordering of the program execution. On the other hand, deterministic replays [6][7][8] provide reproducible execution that guarantees the same execution ordering as the one observed in a recording operation. It is perceivable that any recording operation should incur some instrumentation overhead since the execution of the recording itself would have caused perturbation to the original execution. Nonetheless no research on deterministic replay examines the issue of any changes to the original program execution behavior caused by the recording operation.

In this paper, we propose a simulation-based analysis for embedded software to detect any variations of event ordering caused by instrumentation overhead. It is assumed that application tasks are performed by concurrent threads in a priority-based preemptive scheduling system. The analysis starts with an instrumented program (e.g., for dynamic program analysis) from which we want to analyze the impact of instrumentation overhead to the program execution. The instrumented program is again instrumented to obtain traces including thread interaction events and the overhead of instrumentation code. Kernel's activities as well as external inputs are also recorded. From the traces, we construct a simulation of program execution where the instrumentation overhead is removed. Timing of thread events is calculated which decides the ordering of events in the simulated

run. Then the ordering information of the original program execution with no instrumentation overhead is projected. The contributions of this work are:

1. It provides a novel way of detecting changes in event ordering due to probe effect.
2. Accurate timing of event occurrence is simulated with the consideration of all factors affecting the ordering of program execution, including kernel activities, external inputs, as well as the thread execution time.
3. It provides an analysis of simulation results for inferring the ordering of the original program execution.

The rest of paper is organized as follows. In Section 2, a concise survey of related works is presented. Section 3 describes the design of the simulation analysis. The implementation details are explained in Section 4. Section 5 shows experimental results. The paper is concluded in Section 6.

2 RELATED WORKS

Malony et al. presented the instrumentation uncertainty principle [4] suggesting that the accuracy of execution performance is degraded as the degree of instrumentation increases. Performance perturbation models were proposed to calculate the true performance from instrumented parallel programs. The models were further refined in [5]. In the models, perturbations trigger a change in event execution time and event ordering is represented by time-based and event-based perturbation models. In the time-based model, the thread events are independent while the event-based model considers the dependency between events for recovering the true performance. The dependency considered is performance degradation as arrival time and resource state change. However, the approach assumes the program execution is fixed no matter there is any instrumentation overhead or not. Hence, it does not consider how the program behavior may differ from a un-instrumented program.

When instrumentation perturbation causes different thread interleaving, we will be concerned with the potential problems of data race and execution non-determinism. Data races can result in arbitrary failures and do not help increase scalability [9]. Several efficient dynamic data race detection algorithms [8][11] have been proposed and race detection tools [12][13] are widely used in practice. In general, the approaches are based on the monitoring of read/write operations to shared variables among concurrent threads. However, the delay caused by monitoring operations and any possible probe effect have not been addressed. Deterministic multi-threading techniques [14] provide deterministic event ordering for parallel program execution. In Kendo [15], a thread's progress is represented with a logical clock. It is a thread's turn when its logical clock is the global minimum and a thread can take a lock only during its turn. In [16] and [17], thread's shared memory is isolated from other threads during a parallel phase. During a serial phase, the memory updates to shared variables are applied and locks are taken in a deterministic order. Regardless their overheads, the approaches don't consider any external input events and time-based operations, and cannot be applicable to embedded software.

It may be argued that instrumentation can be done during program replay if a reproducible execution can be constructed. Instant Replay [6] is one of the earliest works that allows cyclic debugging for parallel programs by tracing and replaying relative order of events for each shared object in the program. RecPlay [7][8] based on Lamport's happens-before relations [3] records and replays synchronization operations. In the approach, data races can be detected by checking all shared memory references so that the program is free of data race before record/replay operations. Replay Debugger [18] uses a similar method as RecPlay but it focuses on debugging techniques on embedded software. It is integrated with GDB and can supply additional debugging functionalities for users to control thread execution. To reduce overhead of record and replay, speculative execution and external deterministic replay are used in Respec [19] that is capable of online replaying on multiprocessor systems even with data races. Using speculative execution, the recording process can continue to execute speculatively instead of being blocked until the corresponding replay finishes.

Most profiling tools adopt instrumentation approaches. There have been research efforts to reduce profiling overhead caused by instrumentation. Froyd et al. proposed a call-path profiler based on stack sampling [20]. The profiler, called *csprof*, provides an efficient way of constructing the calling context tree without instrumenting every procedure's call. Zhuang et al. introduced the adaptive busting approach [21] to build calling context tree with a reduced overhead while preserving profiling accuracy. In their approach, unnecessary profiling is avoided by disabling redundant stack-walking with a history-based predictor. The profiling overhead has been further alleviated by taking advantage of multi-core systems. In shadow profiling [22], shadow processes are periodically created for running instrumented code while the original process is running on a different core with minimal overhead. PiPA [23] exploits parallelism by forming a pipeline to collect and process profiles. Application execution and profiling operation are divided into stages that are pipelined and performed in multiple cores. Kim et al. proposed a scalable data dependence profiling to reduce runtime and memory overhead [24] by storing memory references as compressed formats and using pipelining and data level parallelism for the data dependence profiling.

Although there are research works on the overhead compensation for performance measurement and the reduction of instrumentation overhead for reproducible execution and profiling, no work has been proposed to reveal the possibility of execution deviation caused by instrumentation overhead. In this paper, the focused analysis is to verify if the recorded or observed execution is a true representation of the original program execution and if any instrumentation may alter the event ordering of multi-thread embedded programs as to cause any changes of the intended program behavior.

3 DESIGN AND ANALYSIS

3.1 Multi-threaded Program Execution

Multi-threaded program execution can consist of a set of thread interaction events. Such an event, as a sequence of instructions that the program executes, defines a particular action (e.g. a system call) to interact with other threads, internal and external environment. Examples include synchronization, and communication operations, and IO read/write calls. The events can be totally ordered by the timestamps at which the events take place. A partial order can also be defined among the events based on logical dependencies, i.e., happened-before relation [3].

Let's consider multiple runs of a program and the ordering of interaction events from each run. If the execution events happen at different instants, the happened-before ordering of the events may be different from one run to the other. This may lead to a change

of program execution behavior. For instance, in the sleeping barber problem, customers may be served in different orders when they arrive at different instants in separate runs. On the other hand, a particular customer may find out the waiting room is full and miss the service in one run. In the other run, if the barber cuts hair quickly, the customer can find a seat in the waiting room and receive the service eventually. In this case, the execution path of at least one thread is changed that results in a different timestamp-based ordering of events at thread level.

It is a well-known principle used in deterministic replays [7][8] that, for two runs of a data race free program, if we supply the same input data and have the same happened-before ordering of events, then the two runs must result in the same behavior. Conversely, if we observe two distinguished runs of a program, then either inputs and/or the happened-before ordering of one run must be different from those of the other run.

During an execution of a program, a happened-before ordering of program events can be dependent upon the external inputs it receives, including input data value and the timing that new data arrives. If we have the same inputs, the happened-before ordering of the program events is decided by the execution instants of threads events on shared resources and communication. For example, when two threads compete for a resource, the happened-before ordering of the locking events will be decided the instants that the two threads issue their resource requests. The choice on who is going to take the resource first may also depend upon the kernel's scheduling policy, e.g. priority based or FIFO, as well as the kernel's internal states, including run queue state and other tasks running in the system including interrupt handlers. So, when a program is instrumented, we can expect that more CPU time is spent to execute instrumentation code and the execution time of each thread becomes longer. This may have a ripple effect on the instants that program events may take place, and the happened-before ordering of the events.

3.2 Model of Multi-threaded Program Execution

To model multi-threaded program execution, we assume there are n concurrent threads, T_i for $i=1,\ldots,n$, in an embedded program. The threads are data race and exception free and are scheduled preemptively according to their priorities. The system state is the collection of thread local states and a shared global state. The interactions among the threads and with the external environment are done through the operations on the global state, which are represented by interaction events. An interaction event (abbreviated as event), e, can be a lock/unlock, semaphore give/take, message send/receive, or input/output operation that is performed when a thread invokes an event function f. The notation $Ex{:}f \rightarrow e$ is used to indicate that an event e is generated during the execution of the event function f . Apparently, the resulting event of an invocation of f depends upon the local state of the calling thread, as well as the shared global state. For instance, a non-blocking read from a device can succeed or fail depending on the availability of input data when the function is invoked.

There are two important incidents during the execution of an event function f by a threat T. "*T **enters** f*" occurs when the processing begins to take place globally. The entrance gives an important timing information since it decides a logical order between events, as well as the possible resultant event to be generated by the function. For instance, when two threads request a semaphore concurrently, the moments that they enter *sem_wait* function provide an order of the requests and can determine which thread can take the semaphore successfully and the consequent global state. The other important incident is when "***e happens***". It symbolizes the result of execution, as event e, is posted and is available to subsequent execution. Obviously, the invocation of event functions by a thread form a sequence and an event happened previously can causally affect any subsequent events [3].

To include OS and device activities in the model, a system thread, T_0, is added. The events that occur in T_0 consist of interrupts, OS scheduling, and the arrivals and updates of device input data. Thus, an interrupt event of T_0 may set a semaphore and wake up a waiting application thread. Also, an application thread can read in the latest sensor data if the read operation happens after a data update event in T_0.

To define the occurrence instants, we adopt two types of clock. Clock $\boldsymbol{C}$ indicates the CPU cycles used globally. Starting from 0, it is advanced for all activities consuming CPU cycles, including thread execution, O/S activities such as interrupt handler and scheduler, and idle process. In addition, $\boldsymbol{C_i}$ is the thread local clock for thread T_i and is advanced only during the execution of T_i. Hence $\boldsymbol{C_i}$ represents the CPU time spent on the execution of T_i. With the clocks, we define:

$\boldsymbol{ht(f)}$ and $\boldsymbol{ht(e)}$ as the C clock values when f is invoked and e happens, respectively. This is the global timestamp for a call to function f and for event e.

$\boldsymbol{CT_i(f)}$ and $\boldsymbol{CT_i(e)}$ as the $\boldsymbol{C_i}$ values when T_i enters f_i and when e_i happens as a result of T_i's execution, respectively.

Based on $\boldsymbol{CT_i(f)}$ and $\boldsymbol{CT_i(e)}$, we can compute $ct(f_{i,k})$ which is thread T_i's execution time between the instants that the $(k\text{-}1)$-th event $e_{i,k-1}$ happens and that the subsequent function invocation $f_{i,k}$ is entered. Similarly, $ct(e_{i,k})$ can be obtained as the processing time from entering $f_{i,k}$ to posting $e_{i,k}$. As shown in Figure 1, an example execution of the interacting events performed by threads $T1$ and $T2$ and event timestamps are depicted. In addition to the thread local clocks for $T1$ and $T2$, thread events are aligned with the global clock and the kernel scheduling and interrupt service events are included.

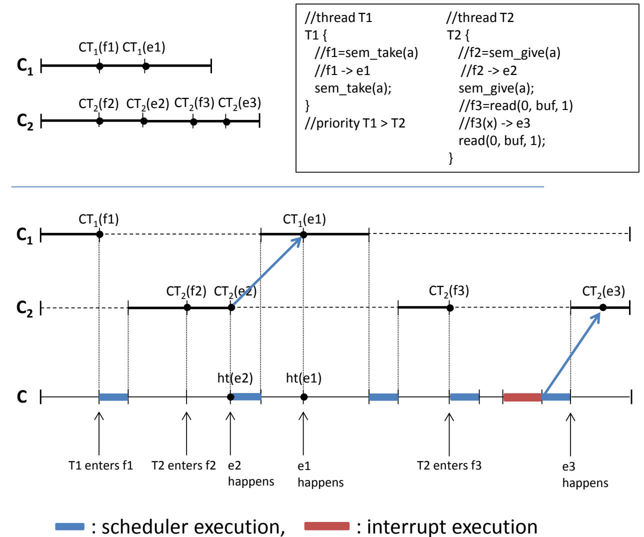

Figure 1. An example global/local clocks and event execution for two interacting threads.

For the events generated from program thread execution and system thread, an event graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ can be constructed where $\boldsymbol{V}$ is a set of events and edge $(e_a, e_b) \in \boldsymbol{E}$ if $e_a \in \boldsymbol{V}$ and $e_b \in \boldsymbol{V}$ and e_b is logically dependent on e_a. Basically, $\boldsymbol{G}$ is a partially ordered graph representing the "happened-before" relation among events [3]. An example of event graphs is shown in Figure 2. In the following sections, we use $\boldsymbol{G^I} = (\boldsymbol{V^I}, \boldsymbol{E^I})$ and $\boldsymbol{G^U} = (\boldsymbol{V^U}, \boldsymbol{E^U})$ to represent the event graphs of the instrumented program P^I and the original program without the instrumentation P^U, respectively. To determine whether there is a probe effect caused by instrumentation, it is assume that the initial states and the external events, including interrupts and the arrivals of input data, are identical in the execution of P^I and P^U. We will then need to compare G^I and G^U or at least find a way to check whether G^I differs from G^U.

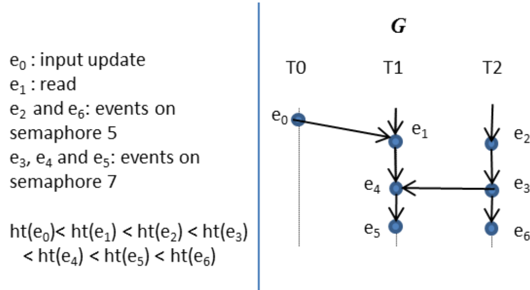

Figure 2. An example partial order graph $\boldsymbol{G}$ with seven events from program execution

3.3 Simulated Program Execution

We consider the execution of instrumented and un-instrumented programs with the same input data. In an instrumented program, extra code is inserted to record program execution behavior that we are interested in. For instance, to measure branch/decision coverage, instrumented code should log the conditions used at each decision point. Furthermore, additional instrumentation is added to obtain the trace of thread interaction events, thread execution time, and the overhead of instrumentation code. Thus, the event graphs $\boldsymbol{G^I}$ can be constructed from the observed event trace. However, $\boldsymbol{G^U}$ of P^U is not observable directly.

With the event functions and the events collected from the execution of P^I, an event-driven simulation can be conducted to analyze the execution behavior of P^U. Since there is no instrumentation in P^U, any instrumentation overhead should be removed from the thread execution time of P^I. As shown in Figure 3, the execution time to be considered in the simulation analysis, $\boldsymbol{CT_i^S}(f)$ and $\boldsymbol{CT_i^S}(e)$, can be obtained from the measured $\boldsymbol{CT_i^I}(f)$ and $\boldsymbol{CT_i^I}(e)$ of instrumented program P^I. As a consequence, if P^U invokes an identical event function and results in the same event as in the execution of P^I, the event may occur at an early instant. The change in execution time can also vary the relative order of program and system events. For instance, an event e of P^I that is invoked after an interrupt may happen before the same interrupt in P^U. This alteration may have a ripple effect. For instance, if the interrupt service routine signals a ready status, the running thread that invokes a function call to read the status in P^U would not see the ready status immediately and may be blocked until the interrupt arrives. The example suggests that, in the simulation analysis, kernel resource and scheduling operations must be incorporated to determine the event ordering.

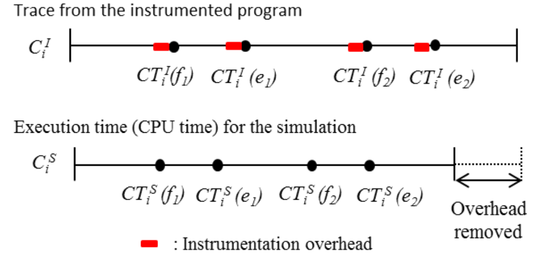

Figure 3. Execution time with no overhead in simulation analysis where C_i^I and C_i^S are thread T_i's local clock in P^I and in simulation.

The goal of the simulation is to generate an event graph $\boldsymbol{G^S}$ to represent the un-instrumented execution of all thread and system events observed in the execution of P^I. In the simulation, a thread T_i will execute a sequence of function invocations and events ($f_{i,k}$ $e_{i,k}$) in timestamp ordering that are collected from P^I. The inter-event execution times of $ct^S(f_{i,k})$ and $ct^S(e_{i,k})$, are calculated by subtracting any instrumentation overhead from $ct^I(f_{i,k})$ and $ct^I(e_{i,k})$. The simulation is done with a global clock C^S and the current simulation time is denoted as *cur-time*, which is advanced based on thread events and operating system activities. At each simulation epoch, the ready thread with the highest priority is chosen as the running thread. The running thread T_i can proceed to perform its next activity $a_{i,k}$, where $a_{i,k}$ is either $f_{i,k}$ or $e_{i,k}$, if there is no system event in [*cur-time*, *cur-time*+$ct^S(a_{i,k})$]. The global clock C^S is then advanced to the next simulation epoch, i.e. *cur-time*+$ct^S(a_{i,k})$. Otherwise, the running thread is preempted by the arriving interrupt irq_j at *cur-time*=$CT_0(irq_j)$. After the un-instrumented execution time of irq_j, T_i may continue its execution or a context switch may occur if there is a thread of higher priority waked up by the interrupt.

As the simulation proceeds from one event to the other, the graph $\boldsymbol{G^S}$ is constructed by adding edges for logical dependence among thread and system events. For instance, a happened-before relation is added from a message send event to a subsequent message receive event. Similar, an interrupt is happened before the thread's *sem_wait* event which gets completed due to a *sem_post* event issued by the interrupt. In Section 4.3, the simulation algorithm implemented in VxWorks is described.

3.4 Analysis of Simulation Results

Once $\boldsymbol{G^S}$ is constructed, we will be interested in the execution behavior of P^U that may be inferred based on $\boldsymbol{G^S}$. Note that $\boldsymbol{G^S}$ does not always represent $\boldsymbol{G^U}$, since if the event ordering of P^U is different from that of P^I (due to the instrumentation overhead), then the execution paths of P^U and P^I might be different. Thus, there might be an event e such that $e \in \boldsymbol{V^I}$ but $e \notin \boldsymbol{V^U}$. Given that the simulation is based on the events in P^I, we have $e \in \boldsymbol{V^S}$. This leads to $\boldsymbol{G^U} \neq \boldsymbol{G^S}$. However, a positive result can be established in the following theorem when $\boldsymbol{G^I}$ and $\boldsymbol{G^S}$ are equivalent.

Theorem 1: If and only if $\boldsymbol{G^S} = \boldsymbol{G^I}$, then $\boldsymbol{G^U} = \boldsymbol{G^I}$.

Proof: To shown the "if" part, let's assume $\boldsymbol{G^U} \neq \boldsymbol{G^I}$. Then, there should be a thread that generates different events or experiences different happened-before relations in the execution of P^U and P^I. Let $e_i^I(k)$ be the first such event of P^I (in terms of the global lock

C^I) that $e_i^I(k) \neq e_i^U(k)$, where $e_i^X(k)$ represents the k-th event performed by thread T_i of program P^X. All events of P^I that are priori to $e_i^I(k)$ have the identical happened-before relations as their equivalent events in P^U. For thread T_i, it must has executed the same first (k-1) functions and generated the same (k-1) events, i.e., $e_i^I(l)=e_i^U(l)$ for $l=1,\ldots,k$-1. Thus, it should use the same function in its k-th invocation, i.e. $f_i^I(k)=f_i^U(k)$.

To have $e_i^I(k) \neq e_i^U(k)$, the global states of P^I and P^U that are used in the processing of $f_i^I(k)$ and $f_i^U(k)$, should be different. This suggests that at least one extra update is inserted before the invocations of $f_i^U(k)$ in the execution of P^U. Let this update event is performed by a different thread T_j and denoted as $e_j^U(k')$. As the instrumentation overhead is removed in P^U, this update event is brought forward and occurs before the invocations of $f_i^U(k)$ in the execution of P^U, even if $e_j^I(k')$ occurs after the invocations of $f_i^I(k)$ in the execution of P^I. This change in event ordering should be observed in the simulation analysis, i.e., $e_j^S(k')$ occurs before the invocations of $f_i^S(k)$ since the same instrumentation overhead is deducted and the program behavior has not been changed before T_i makes its k-th invocation. The addition of the happened-before dependency, $e_j^S(k') \rightarrow e_i^S(k)$, in $\boldsymbol{G^S}$ results in $\boldsymbol{G^S} \neq \boldsymbol{G^I}$.

For the "only if" part, the equivalency of $\boldsymbol{G^U}$ and $\boldsymbol{G^I}$ indicates that thread T_i would invoke the same event function $f_i^I(k)$ and generate the same event $e_i^I(k)$ as in the execution of P^I, even if the instrumentation overhead is removed. Thus, the simulated execution of $f_i^I(k)$ will generate the same event $e_i^I(k)$ which has the same happened-before relations with preceding events. This implies $\boldsymbol{G^S} = \boldsymbol{G^I}$. ■

The theorem implies that, if the logical order of thread events built in the simulation analysis is as same as the one in the execution of the instrumented program P^I, then $\boldsymbol{G^I}$ is the true representation of $\boldsymbol{G^U}$. On the other hand, if $\boldsymbol{G^S} \neq \boldsymbol{G^I}$, the simulation failed in a sense that when the partial order begins to be different, the execution path may also have changed too. This suggests that the instrumented program may have started to take a different execution path. Since the simulation uses the same execution path as the instrumented program, the simulated execution is no longer a representation of the un-instrumented program. However, we can find out when the execution begins to change and how it changes. Let e_d be the very first event of $\boldsymbol{G^S}$ that has a different partial order in $\boldsymbol{G^I}$. Then, the partial graph of $\boldsymbol{G^S}$ for all events priori to e_d is the true representation of the same events in the execution P^U. A follow-up investigation on the trace can be considered to find out how the instrumentation changes the program execution.

4 Implementation

4.1 Execution Environment

To implement the proposed approach of detecting probe effect, an execution environment is set up on a single core of a 1.6 GHz Intel Atom processor running VxWorks 6.8 [25]. The VxWorks' priority-based preemptive scheduler is configured. Two IRQs are available during the execution, a 60Hz timer IRQ and a PS2 keyboard IRQ. The queuing mechanism for tasks blocked on a semaphore is based on task priority.

We consider a simplified system in which three kinds of tasks and system activities can affect the timings of event occurrences and must be traced in the execution of instrumented program: application tasks, interrupt handlers, and the scheduler. All application tasks and kernel operations are run in a flat memory space and there is no page-fault exception. We also assume there are no exceptions from the running applications. The priorities of application tasks are set to be higher than the priorities of any other system background tasks. Thus, the execution of background tasks will not affect the analysis of probe effect. The trace data are sent to the host at the end of the application execution to avoid any activities including file IO during application execution.

4.2 Execution Trace and Measurements

To trace the invocation of event functions and the occurrence of events, we adopt the instrumentation mechanism of the record framework of Replay Debugger [18]. Since the record framework already supports the wrappers for tracing event execution, we added 1) the measurement code for CPU time spent for each thread and 2) the overhead measurement for the instrumentation code. The existing interrupt handlers for timer and keyboard are instrumented to collect the timestamp when an interrupt arrives and to measure the execution time of interrupt handler. In addition, the keyboard interrupt handler is customized to directly communicate with the keyboard driver. For our Atom-based target processor, x86's RDTSC (Read Time-Stamp Counter) instruction is used to collect timestamps.

Task execution time is measured in scheduler hook routines that are invoked for every context switch. For each task, we keep the timestamp that the task is switched in and when the task is switched out. The difference between the current timestamp and the switched-in timestamp is accumulated to the task CPU time. The execution time of scheduling operation and context switching is measured offline using two tasks, task 1 and task 2, where task2 has a higher priority than task 1. First, we let task 2 be blocked on a semaphore and when task1 posts the semaphore, task 2 becomes running. Then, we remove task 2 and just run the *sem_post* operation by task 1. The intervals from the invocation of *sem_post* to the completion of the call are measured for the two cases. The difference is considered as the measured execution time of scheduling operation and context switch.

4.3 Simulation Analysis Algorithm

Using the execution trace from the instrumented program execution and the measurements of the execution environment, the simulation is performed as shown in Figures 6 and 7. It is governed by the invocation to event functions and the occurrence of events and interrupts. The simulation maintains a global clock C which advances when the running task, event function, or interrupt service routing is executed. Note that thread T_0 is used to represent system's external activities and runs concurrently with the scheduled application thread. It consists of interrupt and input data change events. The execution time of interrupt events is measured from ISR routine, whereas the execution time for input data change event is set to 0.

When a thread is scheduled to run, the simulation clock is adjusted to the instant of the next interrupt or the subsequent event function call, whichever comes first. When the global clock C is equal to the arrival time of an interrupt, the time spent on the interrupt is added to C. If there is a thread pending for the arrival of the interrupt, its state is changed to *ready* once the interrupt is processed. Then, the highest priority ready thread is scheduled for execution. If an event function invocation takes place, the resource required by the function is evaluated. The call can lead a return with error, a blocked thread, or the execution of event function. A simulated event happens when an event function is completed. System state may be updated (e.g., a message is de-queued) and blocked tasks may be waked up as the consequence of the

```
1:   enter_func(func: f, thread: T, event e)
2:       if f is resource release function
3:           return
4:       if resource not available
5:           if f is synchronous function
6:               set T to pending
7:           else
8:               mark e as "fail to acquire the resource"
9:       else
10:          mark e as "succeed to acquire the resource"
11:
12:  execute_event(event: e, thread: T)
13:      if e is resource releasing event
14:          if any tasks pending for the resource
15:              select a task and set it to ready
16:          else the resource is released
17:      e_L = event that e logically depends on
18:      V^S = V^S ∪ e
19:      E^S = E^S ∪ (e_L, e)
```

Figure 6. Sub-routines for the the simlulation algorithm

```
1:   do_simulation()
2:     C_cur = 0          // the current global clock
3:     C_i = 0  for all T_i    // C_i is the thread clock for T_i
4:     T_r = the highest priority ready thread
5:     next_act = the earliest action of T_0 and T_r
6:     while (true) {
7:          Δ=the instant of cur_act – C_cur
8:          cur_act = next_act
9:          C_cur is advanced to the instant of cur_act
10:         if (cur_act == ISR completion)
11:             set any tasks pending for the ISR to ready
12:         else {
13:             if (T_r ≠ null), C_r is advanced by Δ
14:             if (cur_act == IRQ arrival at T_0)
15:                 next_act=ISR completion
16:             else if (cur_act==input data change)
17:                 mark data input event
18:             else if (cur_act==function f invocation)
19:                 enter_func(f, T_r, e) // Ex:f→e
20:             else if (cur_act==event e completion)
21:                 execute_event(e, T_r)
22:         }
23:         if any change in task state
24:             T_r = the highest priority ready thread
25:         if (cur_act ≠ IRQ arrival at T_0)
26:             next_act = the earliest action of T_0 and T_r
27:         if all events are executed
28:             break
29:     }
```

Figure 7. The simulation algorithm

happened event. Whenever needed, the scheduler's execution time is added to C to simulate the scheduling operation. When there is no ready thread, the idle process is simulated by advancing C to the next interrupt. Note that, interrupts are accepted during task execution and are delay if they arrive during the execution of event functions or scheduling operation.

5 Experimental Results

We used two multi-threaded programs, the dining philosophers and the sleeping barber, from the LTP benchmark suite [25] in the experiments. The dining philosophers program has 5 philosopher threads with decreasing priorities from philosopher 1 to philosopher 5. Each philosopher is looping from thinking, picking up forks, and to eating. The thinking and eating activities of philosophers 1 and 2 are implemented with blocking reads for keyboard input. On the other hand, the thinking and eating activities for philosophers 3, 4, and 5 are replaced with a simulated computation. When a keyboard pressing interrupt occurs, a philosopher (1 or 2) will be waked up and may preempt another running philosopher. Thus, philosophers will be differently interleaved depending on the timing of keyboard inputs (e.g., different order of philosophers' eating). In the sleeping barber program, the waiting room has three available chairs and there are one barber thread with the highest priority and 5 customer threads with decreasing priorities from customers 1 to 5. The barber is looping from sleeping if there is no waiting customer, and to serving a customer. A customer waits in the waiting room and gets a haircut if a chair is available. He leaves without a haircut if all the three chairs are occupied. The barber's sleep is waked up by a keyboard interrupt. As a consequence, customers 4 and 5 (with lower priorities than customers 1 to 3) may not get haircuts depending on how fast the barber is waked up by keyboard interrupt.

Both programs in the experiments have relatively short execution times considering manually injected keyboard operations. A simulated computation is inserted between event functions to ensure the speed of program execution is comparative to the rate of manual keyboard entry. As we add wrappers to event functions and interrupts, instrumentation overhead is added to the program execution. In addition, extra simulated computation is inserted in thread execution to represent dynamic analysis or profiling overheads. These instrumentation overheads are removed in the following simulation analysis. Tables I and II show execution time and overhead from the average of 5 executions of each benchmark program. . In Table I, the CPU times of

TABLE I. CPU TIME FOR THE DINING PHILOSOPHERS

Thread	CPU time (cycles)	CPU time without overhead(cycles)	Overhead
Philo.1	29,964,934	20,042,918	49.5%
Philo.2	29,073,286	20,029,304	45.1%
Philo.3	161,459,044	153,060,446	5.4%
Philo.4	161,626,702	153,232,244	5.4%
Philo.5	160,931,184	153,228,482	5.0%
Average			8.6%

TABLE II. CPU TIME FOR THE SLEEPING BARBER

Thread	CPU time (cycles)	CPU time without overhead(cycles)	Overhead
Barber	7,965,912	5,024,050	58.5%
Custo.1	10,065,480	9,544,202	5.4%
Custo.2	10,046,790	9,524,506	5.4%
Custo.3	2,568,782	2,047,280	25.4%
Custo.4	1,378,142	1,029,454	33.8%
Custo.5	1,615,038	1,231,966	31.0%
Average			18.4%

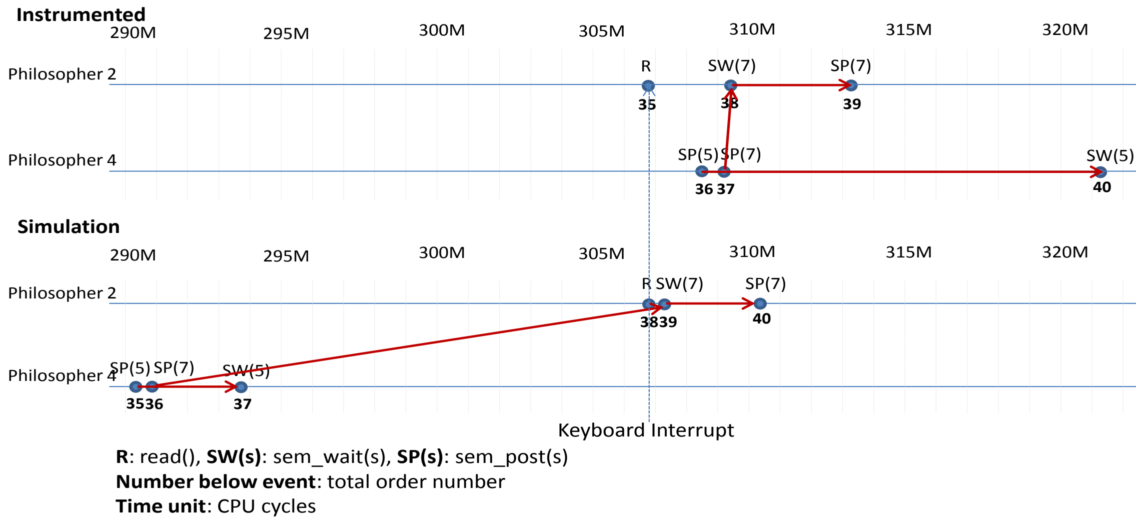

Figure 8. An example execution of dinning philosopher program where $\boldsymbol{G^I} = \boldsymbol{G^S}$ but some events are with different timestamp ordering

philosophers 3, 4, and 5 are greater than that of philosophers 1 and 2 as simulated computations are used for the thinking and eating activities instead of blocking reads. In Table II, customers 1 and 2 spend more CPU times than other customers as we inserted additional simulated computations before entering the waiting room to delay the entrances of customers 4 and 5. Thus, customers 4 and 5 would not arrive too early while the first three customers are waiting on the three chairs, and have to leave. There are some differences in the CPU times spent by customers 3, 4, and 5 as customer 3 always gets a haircut while customers 4 and 5 may get haircuts depending on input timing.

Few experiment results of the probe effect are illustrated in Figures 8 to 10. Figures 8 and 9 are the results of two different executions of the dining philosophers program with different inputs and Figure 10 is based on an execution of the sleeping barber program. In each figure, the event orderings from the instrumented program and the simulation analysis are compared. We only illustrate specific time frames with particular threads that show some variations in event ordering. The horizontal lines show the timestamps of event occurrence and arrow lines indicate the logical dependencies between events. The number below each event denotes the event sequence number in timestamp (total) ordering. Figure 8 depicts a case when $G^I = G^S$ but the timestamp orderings are different. The case appears when philosopher 2 is waiting for a keyboard input and, as soon as the input becomes available, it preempts philosopher 4. In the simulated execution, since thread execution time is reduced due to the removal of the instrumentation, the events of philosopher 4 happen while philosopher 2 is still blocked. As a result, the timestamp ordering of events in the simulation differs from that of the instrumented program. However, since $G^I = G^S$, the simulation is the true representation of the original program and we can conclude that there is no probe effect for the instrumentation overhead.

Figure 9 shows a case where the logical ordering (partial) is altered due to the instrumentation overhead. The timestamp ordering of events in the simulation is changed in a similar way as in Figure 8. Events of philosopher 4 occurred earlier than in the instrumented program execution while philosopher 2 was still blocking for input. Since $\boldsymbol{G^I} \neq \boldsymbol{G^S}$, the simulation is no longer a representation of the original program. We notice that the partial graphs of $\boldsymbol{G^I}$ and $\boldsymbol{G^S}$ are identical until the *37-th* event. Thus, for each thread, the next event function to be invoked immediately after the *37-th* events should be identical in the instrumented program and in the simulated execution. After the *37*-th event, the difference in the logical ordering of events is triggered by the order of the sem_wait(7) events invoked by philosopher 2 and philosopher 4. We manually inspected the source code and found out that the change in the logical ordering did not result in a change of execution path. Hence, any program dynamic analysis based on execution path is still valid. However, in general, it will be a very challenging task to find a change of execution path by manual inspection of source code.

Figure 10 demonstrates a case of a change of execution path due to instrumentation overhead in sleeping barber program. The result shows that $\boldsymbol{G^I} = \boldsymbol{G^S}$ up to the *27-th* event. However, in the *28-th* event, the sem_wait(4) is invoked much earlier in the simulation than what happens in the instrumented program. The resultant partial order graphs are drawn in Figure 11. The second line is for barber thread and the last line is for customer 5. In the simulation, the semaphore given at the 25-th event is taken by customer 5 while in the instrumented program execution the semaphore is taken by barber thread. In fact, in the instrumented execution, customer 5 arrives after the barber is waked up by a keyboard input and begins to cut hair. A chair becomes available for the arriving customer 5. On the other hand, customer 5 arrives before the keyboard input, suggesting the barber is still in sleep mode. The customer should leave as he cannot find any available chair in the waiting room, i.e., a different execution path. Since the simulation uses the same observed event from the instrumented program, the simulated execution cannot be correct after the 28-th event.

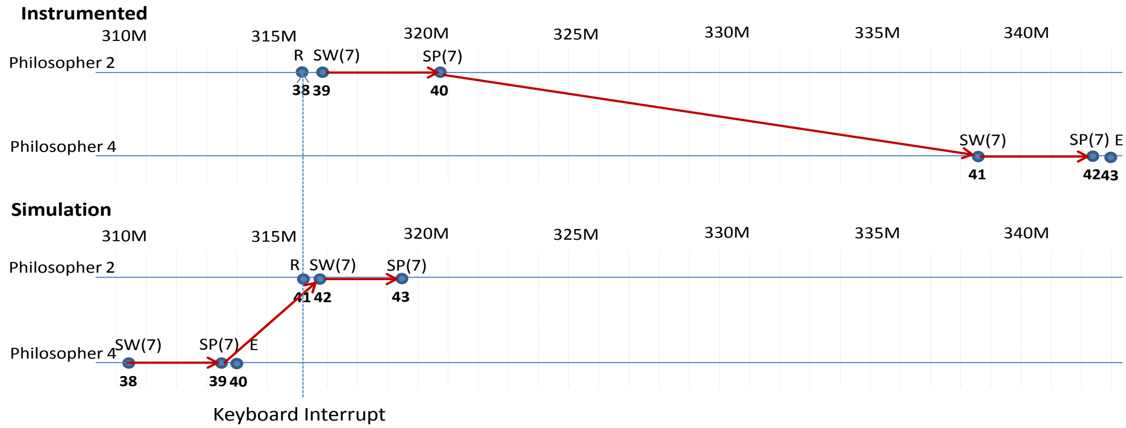

Figure 9. An example execution of dinning philosopher programin which $G^I \neq G^S$ (i.e., the partial orderings are different after the 38th event)

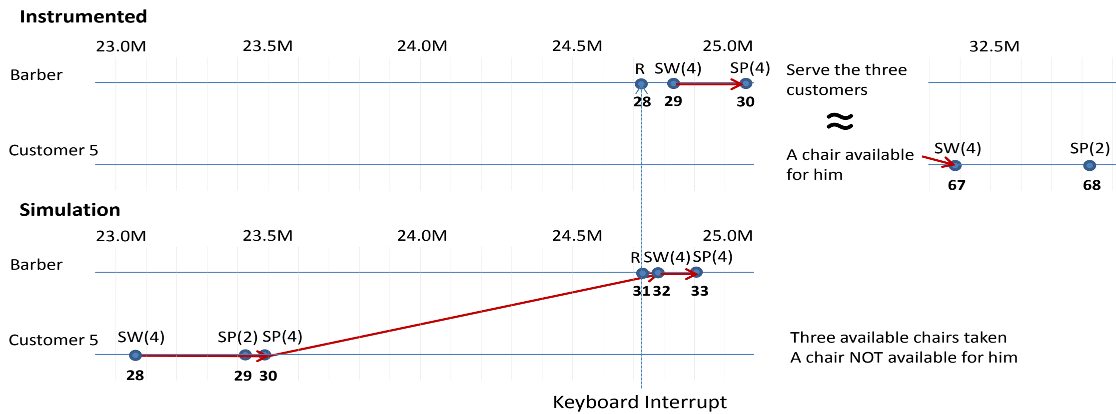

Figure 10. An example execution of sleeping barber program where $G^I \neq G^S$ after the 38th event. In the simulation, customer5 does not get haircut while he does in the instrumented program

6 Conclusions

Often the most important metric in the dynamic analysis of multithreaded programs is the overhead of instrumentation since researchers are aware of the potential probe effect caused by the overhead. However, to the best of our knowledge, no research has proposed a way to detect any changes in program execution when the programs are instrumented. In this paper, we model the execution of multi-threaded program according to the happened-before ordering of global events. Using the trace of event function invocations and OS activities, a simulation-based analysis is presented to detect if the partial order of the happened-before relation is altered by instrumentation. The experiments of two simple applications running on VxWorks demonstrate how instrumentation overhead can lead to changes in event timestamp ordering and in the partial order of happened-before relation.

In this paper, we only consider single core machines with priority-based preemptive scheduling. As a further step, our analysis can be extended for multi-core systems in which thread migration and multiprocessor scheduling should be considered. Also it would be interesting to attempt a hardware-assisted online detection mechanism for potential probe effect. Then, remedy actions, such as synchronization operations, can be inserted to ensure deterministic event ordering

Acknowledgment

This work was supported partially by a grant from the NSF Industry/University Cooperative Research Center (I/UCRC) on Embedded Systems at Arizona State University.

References

[1] J. Gait, "A probe effect in concurrent programs," *Software Practice and Experience,* 16(3), pp: 225-233, 1986.

[2] D. Kranzlmüller, R. Reussner, and C. Schaubschläger. "Monitor Overhead Measurement of MPI Applications with SKaMPI". *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pp: 43-50, 1999.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM,* 21(7), pp: 558-565, 1978.

[4] A.D. Malony, D.A. Reed and H.A.G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis", *IEEE Transactions on Parallel and Distributed Systems*, 3(4), pp: 433-450, 1992.

[5] F. Wolf, A.D. Malony, S. Shende and A. Morris, "Trace-Based Parallel Performance Overhead Compensation", *Proceedings of the International Conference on High*

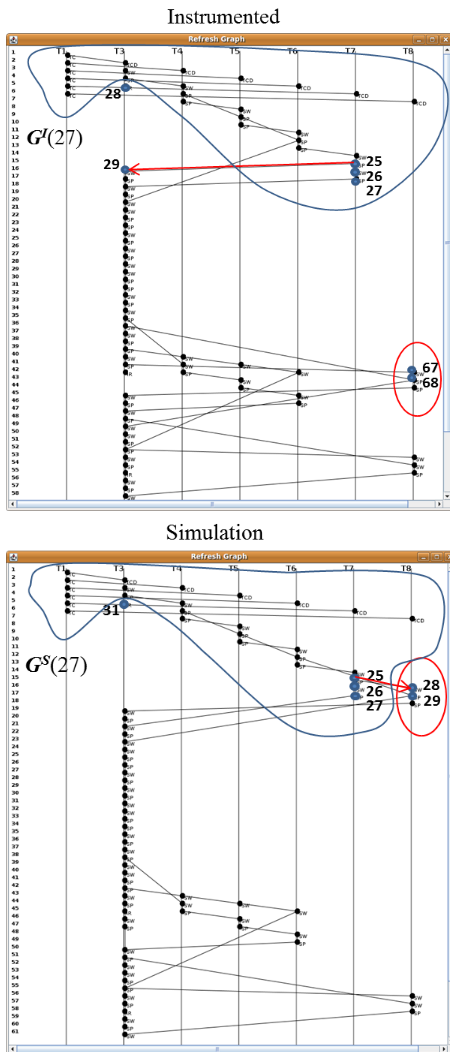

Figure 11. Examples of the partial order graphs from the execution of Sleeping Barber program

Performance Computing and Communications, pp: 617-628, 2005.

[6] T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging parallel programs with instant replay,” *IEEE Transactions on Computers*, 36(4), pp: 471-482, 1987.

[7] K. D. Bosschere, M. Ronsse, and M. Christiaens, “Debugging shared memory parallel programs using record/replay,” *ACM Future Generation Computer Systems,* 19(5), pp: 679-687, 2003.

[8] M. Ronsse, M. Christiaens, and K.D. Bosschere, “Cyclic debugging using execution replay”, *Proceedings of the International Conference on Computational Science*, pp: 851-860, 2001.

[9] S. V. Adve, “Data races are evil with no exceptions: technical perspective,” *Communication of the ACM*, 53(11), pp: 84, 2010.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs,” *ACM Transactions on Computer Systems (TOCS)*, 15(4), pp: 391–411, 1997.

[11] C. Flanagan and S. N. Freund. “FastTrack: efficient and precise dynamic race detection,” *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pp: 121-133, 2009.

[12] DRD, Valgrind-3.8.1. http://valgrind.org/.

[13] Intel Inspector XE 2013. http://software.intel.com/en-us/intel-inspector-xe.

[14] T. Bergan, J. Devietti, N. Hunt, and L. Ceze, “The deterministic execution hammer: How well does it actually pound nails?” *The Second Workshop on Determinism and Correctness in Parallel Programming (WODET)*, 2011.

[15] M. Olszewski, J. Ansel, and S. Amarasinghe, “Kendo: Efficient Deterministic Multithreading in Software,” *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pp: 97-108, 2009.

[16] T. Liu, C. Curtsinger, and E. D. Berger, “DTHREADS: Efficient Deterministic Multithreading,” *The 22nd ACM Symposium on Operating Systems Principles,* pp: 327-336, 2011.

[17] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang, “Efficient Deterministic Multithreading Without Global Barriers”, *The 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pp: 287—300, 2014.

[18] Y. Lee, Y. Song, R. Girme, S. Zaveri, Y. Chen, “Replay Debugging for Multi-threaded Embedded Software”, *Proceedings of IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing*, pp: 15-22, 2010.

[19] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, Satish, P. Chen, et al, “Respec: efficient online multiprocessor replay via speculation and external determinism”, *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pp: 77-90, 2010.

[20] N. Froyd, J. Mellor-Crummey, and R. Fowler, “Low-overhead call path profiling of unmodified, optimized code”, *Proceedings of the 19th annual international conference on Supercomputing*, pp: 81-90, 2005.

[21] X. Zhuang, M. Serrano, H. Cain, and J. Choi, “Accurate, efficient, and adaptive calling context profiling”, *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp: 263-271, 2006.

[22] T. Moseley, A. Shye, V. Reddi, D. Grunwald, Dirk and R. Peri, “Shadow Profiling: Hiding Instrumentation Costs with Parallelism”, *Proceedings of the International Symposium on Code Generation and Optimization*, pp: 198-208, 2007.

[23] Q. Zhao, I Cutcutache, and W. Wong, “PiPA: Pipelined profiling and analysis on multicore systems”, *ACM Transactions on Architecture and Code Optimization*, 7(3), pp: 13:1-13:29, 2010.

[24] M. Kim, H. Kim, and C. Luk, “SD3: A Scalable Approach to Dynamic Data-Dependence Profiling”, *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp: 535-546, 2010.

[25] VxWorks Kernel Programmer's Guide 6.8.

[26] Linux Test Project, http://ltp.sourceforge.net/.