2015 IEEE 39th Annual International Computers, Software & Applications Conference

Replay Debugging of Real-Time Vxworks Applications†

Chunlei Ma, Xiang Long
School of Computer Science and Engineering
Beihang Univeristy
Beijing, China
{machunlei, long}@buaa.edu.cn

Bo Jiang‡
School of Computer Science and Engineering
Beihang Univeristy
State Key Laboratory of Computer Science
Institute of Software,
Chinese Academy of Sciences
Beijing, China
jiangbo@buaa.edu.cn

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software,
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

***Abstract*—Debugging multi-task real-time VxWorks applications is tedious and time-consuming for developers. The non-determinism within the application execution makes the developers hard to reproduce a failure. As a result, the developers cannot perform cyclic debugging easily on these real-time applications. Replay debugging techniques can help developers to replay the failure scenario with determinism. In this paper, we propose an approach to replaying real-time VxWorks applications. We make use of dynamic binary instrumentation to record data flow non-determinism. Furthermore, we instrument the VxWorks kernel to record control flow non-determinism. Finally, the recorded information together with a modified VxWorks kernel is used by our replayer to support effective replaying. Our evaluation results show that our approach can effectively replay real-time multi-task VxWorks applications with low overhead.**

Keywords-execution replay, real-time, vxworks, debugging

I. INTRODUCTION

Program debugging is a tedious and time-consuming work. It greatly reduces the development productivity of the developers. How to design automatic or semi-automatic debugging tools to help the developers improve their debugging effectiveness has been the focus the software engineering research for many years.

The most frequently used debugging approach by developers is the cyclic debugging. In cyclic debugging, the developers execute a faulty program with the same input repeatedly to analyze same failure scenario in detail. In each repeated execution cycle, the developer will set break points, analyze the state of program variables and gradually narrow down the code to inspect. Finally, the developer will locate the fault in the program after this repeated debugging cycle. The delta-debugging approach [5] in essence automates this cyclic debugging strategy with software tools.

The cyclic debugging approach assumes repeated execution with the same input will trigger the same fault and lead to the same failure. However, this assumption may not always hold, which is caused by the non-determinism faced by the program under execution. There are many sources of non-determinism faced by a program in addition to input: the non-deterministic API calls, the data received from network, the thread scheduling choices made by the operating system, and interleaved access of shared memory access from different threads. Thus, to enable effective cyclic debugging of these non-deterministic programs, replaying the failed execution deterministically is necessary.

In this work, we focus on the replay debugging of real-time applications on VxWorks [23] real-time operating system, which is the most popular real-time operating system in the avionics industry. Compared with replaying traditional applications, the replay debugging of VxWorks application has several challenges.

First, the replay of real-time VxWorks applications requires ensuring the same timing of non-deterministic events while the replay of traditional applications mainly focuses on ensuring the same order of non-deterministic events. The real-time feature of VxWorks is mainly supported by its interrupt handling mechanism. For a real-time operating system like Vxworks, an interrupt can happen any time during program execution and must be handled in real-time to support safety-critical tasks. As discussed by Lamport [10], a correct reproduction of the temporal behavior of an execution implies a correct reproduction of the ordering of events in the execution. However, the reverse is not true. Thus, to replay real-time VxWorks real-time applications, it is important to record and replay of the precise timing of the non-deterministic events.

Second, the real-time VxWorks applications are usually safety-critical applications. Thus, for a practical replay-debugging tool, the overhead of non-deterministic events logging (or recorder modules) must small to minimize the impact of probe effect.

† This research is supported in part by the National Natural Science Foundation of China (project no. 61202077), the Basic Research Funding of Central Universities, the National Key Basic Research Program of China (project no 2014CB340702), the National Natural Science Foundation of China (project no 61379045), and the National Science and Technology Major Project of China (grant no. 2012ZX01039-004).

‡ Correspondence author

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.270

In previous work, Patel [14] have proposed a solution to replaying the real-time VxWorks applications, but it only supports the applications written by Ada language.

Our solution to replaying real-time VxWorks applications contains the recording phase and the replaying phase. In the recording phase, we systematically log the nondeterministic data flow inputs, synchronous events and interrupt events. In the replaying phases, we modify the VxWorks operating system source code to guide the re-execution of applications using the recorded non-deterministic choices. Our evaluation on some VxWorks applications demonstrates the feasibility of our approach.

The main contribution of our work is as follows: (1) We proposed a new approach for replay debugging of real-time VxWorks applications. (2) We have performed a systematic experiment on real-time Vxworks applications to evaluate the effectiveness of our approach.

The following sections of the paper are organized as follows. In section II, we discuss some background knowledge on replay debugging. Then we present our replay debugging approach in section III, followed by our empirical study in section IV. Finally, section V presents related work and section VI concludes the paper.

II. Background on VxWorks Applications

The VxWorks operating system is designed for executing hard real-time applications by ensuring determinism. VxWorks real-time applications (processes) provide the means for executing applications in user mode. Each process has its own address space, which contains the executable program, the program's data, stacks for each task, the heap, and other resources. Many VxWorks processes may be present in memory at once, and each process may contain more than one task (equivalent as a thread in other operating systems).

The VxWorks process realized the real-time requirements in various ways. First, VxWorks process (application) are not scheduled, the tasks are scheduled throughout the system with preemptive, priority-based algorithm. Second, a task can be preempted not only in user mode but also in kernel mode. Thus, the strong preemptibility of VxWorks kernel ensures the highest priority task in the system that is ready to run will execute. This also ensures the real-time interrupt events are handled in a timely manner.

III. Replay Debugging of Real-Time Vxworks Applications

In this section, we present our approach to replaying real-time VxWorks applications.

A. The Overall Architecture

The overall framework of our replay system is shown in Figure 1.

We divide the execution process of our replay system into 2 steps: the recording phase (recorder) and the replaying phase (replayer).

In general, the non-deterministic information faced by an application execution includes 3 types: the data flow information, synchronous events and asynchronous events. The data flow consists of the data read from external devices and data received from other tasks. Synchronous events caused by synchronous API operations, such as semTake, semGive, msgQSend. The asynchronous events mainly consist of interrupt events.

In the recording phase, we need to record the data flow information and control flow information. To record the data flow information, we use the Valgrind tool to instrument application binary. To record the control flow information, we manually instrument the VxWork source code to record the task switch information, interrupt events, and API call events. We also make use of the *WindView* [24] tool to facilitate information collection.

Then, we use the *upload* task to send logs recorded back to the host. On the host machine, we parse the log file to filter out the unnecessary information, such as those information generated by the other applications and system tasks.

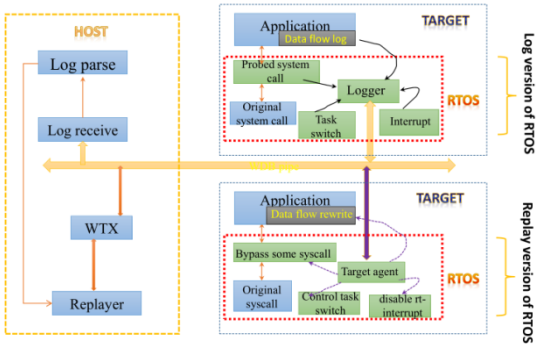

Figure 1. The overall framework of replay system

In the replay phase, we aim at replaying the application execution with those recorded information. We use a replay version of Vxworks kernel where we have revised the VxWorks source code to disable the real-time timer and replace the task scheduler with our customized one. To control the program execution, we bypass the system calls, take control of task scheduling, and disable real-time interrupt. We also make use of the WTX (Wind River Tool eXchange protocol) interfaces within the Tornado IDE to add/delete breakpoints, resume task execution, and to read/write some global variables, etc.

We will detail our design of the recorder and replayer in the sections below.

B. The Recorder

1) Recording Data Flow infornmation

To record data flow information within the application, we use the Valgrind [22] tool to perform binary instrumentation. When the task reads some data from external devices or communicates with other tasks through network, we log the data. If the data read from external device can be large in size (e.g., from a file or database), we will NOT log the data received. Instead, we just ensure the tasks read the same data by carefully preparing the data sources. Note we do NOT record the data written to external devices or sent to other tasks since they will not affect program execution state. In this way, we can reduce data-flow instrumentation overhead significantly.

2) Recording Synchronous Events

The Synchronization events of interest to replay debugging mainly consists of the system calls that can cause task reschedule, such as semTake, semGive, msgQSend, msgQReceive and so on.

Next we will use the function semTake as an example to describe our recording strategy. The modified version of *semTake()* is described in Figure 2, where the code in the red rectangle is the instrumentation code added by us.

```
STATUS semTake(SEM_ID semId, int timeout)
{
    SM_SEM_ID  smObjSemId;      /* shared semaphore ID */
    unsigned int pc = getpc();  /* get the current pc */
    (taskIdCurrent->syncCounter) += 1;/*use the extended space to
        store the counter*/
#ifdef  WV_INSTRUMENTATION
    EVT_ARG_4(EVENT_MYSEMTAKE, semId, taskIdCurrent->syncCounter,
      pc, (int)taskIdCurrent)
#endif
        /* check if semaphore is global (shared) */
        if (ID_IS_SHARED (semId))
        {
            smObjSemId = (SM_SEM_ID) SM_OBJ_ID_TO_ADRS(semId);
            return ((* semTakeTbl[ntohl (smObjSemId->objType) & SEM_TYPE_MASK ])
                    (smObjSemId, timeout));
        }
    /* otherwise sempahore is local */
#ifdef  WV_INSTRUMENTATION
    /* windview - level 1 event logging */
    EVT_OBJ_3 (OBJ, semId, semClassId, EVENT_SEMTAKE, semId,
            semId->state, semId->recurse);
#endif
    return ((* (semTakeTbl[semId->semType & SEM_TYPE_MASK])) (semId, timeout));
}
```

Figure 2 The modified version of semTake

In the modified version of semTake, we add a few recording statements. First, we call the *getpc* to get the program counter, named *pc*, and we increase the synchronous events number, named *syncCounter*, which is added for each task. Then we call the Macro *EVT_ARG_4* provided by VxWorks to log the information which contains of the type of the event (e.g. *EVENT_MYSEMTAKE*), the id of the event, the synchronous event counter, *pc* and the current task id.

Meanwhile, we also need to record the task scheduling information. So we modify the reschedule function, which provides the VxWorks task scheduler implementation (reschedule). The modified version is shown in Figure 3, where the code in the red rectangle is our added instrumentation code.

Whenever there is a task switch, we must record the precise locatoin where one task releases the CPU. We invoke the MACRO *EVT_CTX_TASKCONTEXT* defined by us to record the information when task switching occurrs. We record the type of the event (e.g. *EVENT_TASK_CONTEXT_OLD*), information of the task to swap out and the id of the task to swap in. The information about the task to swap out contains the values of 32 general registers and the checksum of task stack, which together represent the context information to identify a program execution point uniquely. We get the value of general registers from the TCB (Task Control Block) of the task and the stack checksum with function *CHECKSUM*. Vxworks use the TCB address of the task as its identification.

```
void reschedule (void)
{
        ......
unlucky:
    taskIdPrevious = taskIdCurrent;    /* remember old task */
    ......
    if (!workQIsEmpty)
    {
        intUnlock (oldLevel);    /* UNLOCK INTERRUPTS */
        goto unlucky;            /* take it from the top... */
    }
    else
    {
        ......
    taskIdPrevious->syncCounter = 0;
#ifdef WV_INSTRUMENTATION
    EVT_CTX_TASKCONTEXT(EVENT_TASK_CONTEXT_OLD, (int) taskIdPrevious,
        taskIdPrevious->regs,
        CHECKSUM(taskIdPrevious->pStackBase, taskIdPrevious->stackSize),
        taskIdCurrent);
#endif
        kernelState = FALSE;            /* KERNEL EXIT */
        windLoadContext ();            /* dispatch the selected task */
    }
}
```

Figure 3. The modified version of reschedule

A synchronous event may lead to a task switch when the calling task is blocked. When a synchronous event happens in a loop, the program counter is insufficient to pinpoint which invocation of the synchronous event lead to a task switch. Since our replayer will be responsible for task scheduling in the replay phase, we must differentiate this. So we use a synchronous event counter (syncCounter) for each task to differentiate each invocations in a loop. The counter will increment itself on each synchronized event invocation and will reset upon task scheduling. In the replay phase, our replayer will decide when to trigger a rescheduling based on the counter.

3) Recording Interrupt Contexts

When executing a loop, if interrupt occurs, the recording of program execution location calls for a special treatment. As shown in Figure 4, an interrupt happens when executing a loop, leading to a task switch. The program counter (PC) value is 0x2458. Since this interrupt happens within the loop, the same PC will be revisited several times during execution. Therefore, the program counter is insufficient to identify the precise program execution.

```
for(i = 0; i < 10; i ++)
{
    a = a + i;
----------------------PC = 0x2458
    b = a * 2 + i;
}
```

Figure 4. Interrupt occurs while in a loop

To solve this problem, the program counter needs to be extended with a unique marker to differentiate between loop iterations, subroutine calls and recursive calls. The content of this marker should be able to uniquely define the state of the program upon interrupt. Since the task registers and task stack values change across loop iterations, we can use them as the unique marker. Since the register values and stack information is too large to record efficiently, we use their MD5 hash as unique identification.

C. The Replayer

1) Replaying Data Flow infornmation

At the replay stage, we must ensure the data read from the external device or received from other tasks will be as same as the record phase. For large size of data from the files or databases, we just carefully prepare the execution environment to ensure the same data is consumed by the application. For the other data inputs, the replayer uses the Valgrind dynamic binary instrumentation tool to bypass the corresponding execution and directly feed the recorded data to the application.

2) Replaying Synchronous Events

Since the replayer is responsible for task scheduling in replay phase, the replayer must first filters out the records of synchronous events that do not trigger the task switch. Then we add the breakpoint at the *pc* value corresponding to a synchronous event logged in record phase with the VxWorks debugger.

In replay phase, we must guarantee that only one task of our replayed application is in ready or running state, the others should be in suspended state.

When the synchronous operation is called, the breakpoint will be hit and the current running task will suspend at the *pc* value. Then we will check whether its synchronous event counter is equal to that of the recorded one. If so, a task switch must be performed. Then we will schedule another task to run according to the order of task executions in recorded phase. Otherwise, we will continue the suspended task. Finally, when some tasks release the semaphore, and if it is time for scheduling it based on the recorded information, we will schedule it for running.

3) Replaying Interrupt Events

The interrupt is a kind of asynchronous event, and it can happen in any position on the execution path of the application as long as it is enabled. Remember we have logged the pc and unique marker (the hash of the register values and stack information) in the recoding phase. To replay an interrupt, we first add breakpoint at the corresponding program counter recorded when interrupt happens. If the breakpoint is hit, it means that the current running task may be interrupted by interrupts in recording phase, which we will verify with the unique marker. Specifically, we compare the register and stack hash of current task with that of the recorded one. If they are not equal, the current task should continue to run. If they are equal, it means that the current task reaches the location where the interrupt occurred in its record phase. Since that interrupt causes the task switch in the recorded execution, we delete the breakpoint set by program counter of current task and then we schedule another task to run according to the recorded order of task executions. If this new task has been interrupted by interrupts during this time slice, a new breakpoint should be added like similarly.

IV. EVALUATION

A. Evaluation Goal

In this section, we want to evaluate the effectiveness and overhead of our proposed replay debugging approach. We use 3 sample VxWorks application to perform the replay debugging experiment.

B. Experiment Setup

In our experiment, we used the VxWorks 5.5 and Tornado development environment. This environment is provided by Wind River. It is running on Microsoft Windows XP with the Intel Core I5-4430 and 1G memory. We used the VxSim tool to simulate the Vxworks target board.

We use 3 real-time multi-task VxWorks applicatoins for evaluation as shown in Table 1. In each row, we list the ID, the number of lines of source code, the number of tasks, and a brief description of the application

Table 1. Subject Programs for Evaluation

Application ID	LOC	No. of Tasks	Description
A1	50	2	The implementation of producer and consumer problem
A2	70	2	Calculating and printing primes
A3	150	3	Random number generation and sorting

The first application (A1) simulates the producer-consumer problem, which contains frequent semaphore operations. The second program (A2) spawns two tasks to calculate the prime number, one is responsible for calculating and printing primes from 1 to 20000 while the other is responsible for the prime numbers from 20000 to 40000. The third program (A3) spawn three tasks, the first one is responsible randomly generating 20000 integer numbers from -1000 to 1000 and sending data to other tasks, the second is responsible for sorting the 10000 integers received from the first task and the last task works similar to the second one.

To evaluate the effectiveness of our replay debugging approach, we performed replay debugging on the 3 subject applications with our approach and checked whether the replay is successful.

To evaluate the overhead of our approach, we executed the application with and without enabling our recording mechanism and compared the results. Since the scheduling time slice have an impact on the frequency of task schedule (The shorter the slice, the more frequent the schedule), we evaluate two settings of the task schedule slice: 10ms and 100ms. We made the setting through the interface.

C. *Results and Analysis*

We have successfully replayed all three subject programs with our proposed approach, which demonstrate the effectiveness of our technique.

Then we compare the performance of the application before and after instrumentation on the 2 time slice settings.

The results are shown in Figure 5. We use "L" to represent long time slice of system scheduling (i.e., 100ms) and "S" to represent short time slice of system scheduling (i.e., 10 ms). The x-axis represents the different combination of application and time slice setting. The y-axis represents the execution time in seconds. We can see that for both settings, although the instrumentation incurred overhead for each application, the overhead is relative small. A detailed analysis show that the overhead of our instrumentation is less than 5%. Furthermore, the comparison of corresponding applications in the two different time slice setting ("L" and "S") are also small, which shows that our recording overhead within the scheduler is also low, such that more frequent task switches will not impact performance significantly.

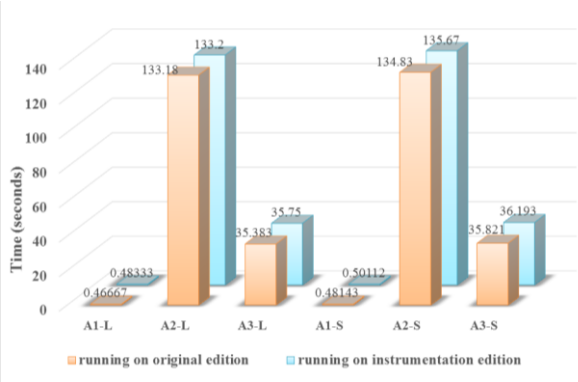

Figure 5 Comparison of Instrumentation Overhead

In summary, the evaluation results show that our technique can effectively replay real-time VxWorks applications with low overhead.

V. RELATED WORK

In this section, we review the closely related work on deterministic replay debugging.

In recent years, Researchers have done a lot of research on the deterministic replay and proposed some useful replay solutions. They can be based on hardware support or software or both.

Bacon [3] et al. first proposed a replay scheme that has the hardware support, and they present a hardware/software design that allows the order of memory references in a parallel program to be logged efficiently.

LeBlanc and Mellor-Crummey [11] make a contribution to software-only replay. Their method is denoted Instant Replay and focuses on logging the sequence of accesses to shared objects in parallel executions.

Some subsequent proposals have been extensions to the work of LeBlanc and Mellor-Crummey. For instance, Audenaert and Levrouw have proposed a method for minimizing recordings [2] of shared object accesses. And Chyassinde Kergommeaux and Fagot included support for threads [5] in a procedural programming extension of the Instant Replay method.

In addition, CLAP [9] is based on LLVM compiler and symbolic-execute tool KLEE and is also a software-only replay. CLAP uses LLVM to achieve source code instrumentation, and records the program execution path information in symbolic execution. CLAP could be divided into three stages: record, inference, and replay. In record phase, CLAP records the program execution path rather than recording the data race and synchronous instrumentation. In inference stage, CLAP uses a symbolic execution tool KLEE. The records of the program execution path information are mainly used for accelerating the inference, and CLAP only needs to find one path of execution in inference stage. In the process of symbolic execution, CLAP collects the information about the program execution, such as the shared read/write operations, the read/write operation sequence of each thread, synchronous operations and path conditions (as all bound symbolic branch selected set) and the forth. When symbolic execution ends, CLAP will translate all the information collected in symbolic execution phase into a global equation, and the solution of the equation represents the thread scheduling scheme. Then the step comes into the constraint solving phase, and CLAP enters the replay stage when finding a thread scheduling scheme at the end of symbolic execution phase. CLAP does not need to record shared memory dependencies and program state, and reduces the overhead of the record stage.

Petel [14] et al describe a language-based framework for tracing and replay of real-time concurrent programs in Vxworks. Their framework consists of wrappers for Vxworks synchronization constructs like semaphores, message queues and threads. This framework supports two modes of execution, namely, trace and replay. In the trace mode, important synchronization events, with necessary debugging information, are recorded into a trace file. In the replay mode, a trace of synchronization events is read and used to control the behavior of threads so that these events are exercised in the same order as they were recorded. But their framework only be used for the application written by Ada language.

Sundmark [19] proposed a method to replay debugging of embedded real-time system. He use the task stack and context information to pinpoint the location of a program execution point. However, it relies on manual instrumentation on the application to record data-flow information, which makes their approach hard to automate.

PinPlay [12][15] is a replay system that focus on the application debugging. They are based on the dynamic binary instrumentation of a captured program trace. Unfortunately, PinPlay need the support of instruction count while some commercial CPU does not support and the overhead of logging is another problem to address.

ODR [1] and PRES [4] are two novel approaches that facilitate replay debugging. But they are unable to immediately replay an application, given the log data from the logging phase. Instead, they intelligently explore the space of possible program execution path until the original output or bug is reproduced. However, they require the mechanism of recording a subset of the cache traffic between memory and the CPU's.

VI. CONCLUSION

Debugging multi-task real-time VxWorks applications is hard. The non-deterministic events occurred during application execution makes it hard to reproduce the failures. As a result, the developers cannot perform cyclic debugging on these real-time applications. Replay debugging techniques can help developers to repeat the failure with determinism. In this paper, we propose a solution of replaying real-time VxWorks application. We use dynamic binary instrumentation to record data flow information. And we instrument the VxWorks kernel to record synchronization events and interrupt events. The recorded information are used by our replayer to support effective replay debugging. Our evaluation results show that our approach can successfully replay real-time VxWorks applications with low overhead.

With the popularity of multi-core architecture, more and more real-time VxWorks applications are infact multi-task concurrent applications. Those concurrent tasks have interleaved access to shared memory, which add another source of non-determinism. In future work, we will improve our approach to support effective replay debugging of multi-core real-time VxWorks applications.

REFERENCES

[1] G. Altekar and I. Stoica. "ODR: Output-deterministic replay for multicore debugging." In *proceedings of the 22nd Symposium on Operating Systems Principles*, pages 193 – 206, 2009.

[2] K. Audenaert and L. Levrouw. Reducing the space requirements of instant replay. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 205 – 207. ACM, May 1993.

[3] D.F. Bacon, S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. ACM. 1991.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. "Framework for instruction-level tracing and analysis of program executions." In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154 – 163a,1,2,3, 2006.

[5] J. Chassin de Kergommeaux and A. Fagot. Execution Replay of Parallel Procedural Programs. *Journal of Systems Architecture*, 46(10):835 – 849, 2000.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 342-351, 2005.

[7] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In:Proc. of the 5th USENIX Symp. on Operating System Design and Implementation. New York: ACM Press, 2002. 211−224.

[8] J. Gait. A Probe Effect in Concurrent Programs. *Software-Practice and Experience*, 16(3):225 – 233, March 1986.

[9] J. Huang, C. Zhang, J. Dolby. CLAP: recording local executions to reproduce concurrency failures. PLDI. 2013: 141-152.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-565, July-1987.

[11] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE ransactions on Computers*, 36(4):471-482, April 1987.

[12] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, Pin: building customized program analysis toolswith dynamic instrumentation, *In Proceedings of the 2005 ACM SIGPLAN conference on* Programming Language Design andImplementation (PLDI '05), 190 – 200, 2005.

[13] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." *In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 89-100. 2007.

[14] D. K. Patel. Tracing And Replay Of Real-time Concurrent Programs In VxWorks. *Computer Science & Engineering*, 2007.

[15] 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 the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10)*. ACM, New York, NY, USA, 2-11, 2010.

[16] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. "PRES: Probabilistic replay with execution sketching on multiprocessors." *In proceedings of the ACM SIGOPS 22nd Symposium on Operating System Principles*, pages 177 – 192, 2009.

[17] Y. Saito. "Jockey: A user-space library for record-replay debugging." *In Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, pages 69-76, 2005.

[18] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. "Flashback: A lightweight extension for rollback and deterministic replay for software debugging." In *Proceedings of the USENIX Annual Technical Conference*, 3, 2004.

[19] D. Sundmark. Replay Debugging of Embedded Real-Time Systems: A State of the Art Report, *MRTC Report*, M¨alardalen Real-Time Research Centre, Malardalen University, February 2004.

[20] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. *In Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.

[21] M. Xu, R. Bodik, M.D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *In Proceedings of the 30th International Symposium on Computer Architecture*. New York: ACM Press, 2003. 122−135.

[22] Valgrind. http://valgrind.org/.

[23] VxWorks. http://www.windriver.com/products/vxworks/

[24] VxWorks Programmer's Guide 5.5. http://www.vxdev.com/docs/vx55man/vxworks/guide/

[25] Wind View User Guide. http://www.vxdev.com/docs/vx55man/windview/wvug/index.html