**Deterministic Replay for Multi-core VxWork Applications＊**

|  |  |
| --- | --- |
| Junjie Liu, Xiaopeng Gao, Bo Jiang**†**  School of Computer Science and Engineering  Beihang Univeristy  Beijing, China  {liujunjie, gxp, 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 is an important and yet expensive activity in software development. With the adoption of multi-core computing platform, the debugging of multi-core bugs on real-time platform becomes challenging. Those multi-core concurrency bugs are often called Heisenbugs, because they manifest randomly during replay. As a result, it would be difficult for a programmer to use cyclic debugging to locate the fault. In this work, we present a deterministic replay tool for multi-core real-time VxWorks applications. We combine static analysis and source code instrumentation to help record interleaved access to shared memory for multi-tasking VxWorks applications. Our experimental results show that our tool can effectively reply multi-core real-time VxWorks applications with reasonable overhead.

Keywords- deterministic replay, VxWorks, multi-core debugging

# Introduction

As the Moore’s law ceases to hold in recent years, engineers try to increase the number of cores on chip instead of increasing the clock speed of a single core. Multi-core computing is gaining popularity on desktop, server and embedded applications. However, for the real-time safety-critical systems (e.g., avionics applications on VxWorks), the adoption of multi-core platform is much slower. A major concern is its potential threat to safety. To harness the processing power of multi-core hardware, engineers usually use parallel programming to run different tasks (threads) on different cores. However, parallel programming is prone to Heisenbugs, which manifest randomly due to interleaved shared memory access, task scheduling orders, and other non-deterministic factors.

Attracted by the high performance of multi-core hardware, safety-critical real-time system has started supporting them cautiously. Starting from VxWorks 6.6, engineers can build multi-core applications running different tasks. However, when such multi-core real-time applications fail, debugging the concurrency bugs is challenging. Deterministic replay techniques can help alleviate this problem. Deterministic replay tool records the non-deterministic choices made during program execution, generates logs, and replays the execution based on the logs. Combined with the traditional cyclic debugging tools (e.g., GDB), the deterministic replay tool can help developer step into the failed execution and locate the fault.

There are many deterministic replay techniques proposed in previous work. In general, the deterministic replay tool usually contains two parts: the recorder and the replayer. The recorder can be realized at software level through system or application instrumentation. For multi-core debugging on real-time system, recording interleaved access to shared memories from different tasks can lead to significant performance degradation. To minimize the recording overhead, recorder may also adopt hardware implementation. However, hardware solutions are limited in that they need special hardware to work. Different deterministic replay tools adopt different solutions to record nondeterministic choices. Scribe[11] relies on page protection mechanism to perform recording. PinPlay[22] and ODR[1] use Pin to perform dynamic instrumentation. ThreadSanitize[23][31] relies on static binary instrumentation and is integrated with gcc 4.8. These tools can perform recording with high or reasonable cost, but they are designed for Linux or Windows rather than real-time platform.

There are also a few solutions designed for VxWorks. Daniel Sundmark[24] proposed a solution for early versions of single-core VxWorks. In our previous work[14], we have also proposed a deterministic replay tool for single-core VxWorks with operating system level instrumentation. However, starting from VxWorks 6.6, multi-core hardware is supported. None of the above works solves the problem of deterministic replay for multi-core VxWorks application. More specifically, we are mainly focusing on the problem of how to automatically and effectively record the interleaved memory accesses with moderate cost.

The main contributions of this work are as follows. First, we proposed an automatic solution for recording interleaved shared memory access for VxWorks multi-core applications. Second, we have built a deterministic replay tool for multi-core VxWorks applications on VxWorks 6.9.

\* This research is supported in part by NSFC (project no. 61772056), the Research Fund of the MIIT of China (project no. MJ-Y-2012-07), and the research fund of the State Key Laboratory of Virtual Reality Technology and Systems.

† Correspondence author

The organization of the paper is as follows. In section II, we will provide the background information on debugging for multi-core VxWorks. In Section III, we will present the design and key implementation considerations of our multi-core deterministic replay tool. In Section IV, we will perform an experimental evaluation on the effectiveness and efficiency of our tool. Finally, we discuss related work in Section V and conclude the work in Section VI.

# Background

In this section, we present background information on VxWorks and discuss challenges for performing deterministic replay on multi-core VxWorks applications.

VxWorks is a popular real-time operating system wildly used in aerospace, defense, medical, industrial, robotics, automotive, and consumer electronic domain. It is also frequently used in safety critical areas where completion of tasks before deadline is crucial.

The VxWorks 5.5 only supports single-core processor with a flat memory space. The user tasks in VxWorks 5.5 run as kernel module and make system calls without address translation. Furthermore, there is no concept of process and memory space isolation in VxWorks 5.5. Thus, one user application may accidentally modify a variable of another application.

With the popularity of multi-core platform, VxWorks supports Real Time Process (RTP) in version 6.0 and supports multicore platform starting from version 6.6. The support of multicore processor is comprehensive, which includes asymmetric multiprocessing (AMP), symmetric multiprocessing (SMP), and mixed AMP and SMP mode.

To systematically record the nondeterministic choices made during program execution, the recorder must record both the control and data flow choices. For control flow choices, we can instrument the operating system. However, for data flow information, especially the information on interleaved access to shared memory; system level implementation using memory page can be expensive as all tasks running in the OS may be affected. So in this work, we resort to source code instrumentation to address the problem.

Furthermore, there are some challenges to design a multi-core replay tool working on VxWorks. The recorder should reduce the amount of code changes in kernel, reduce the number synchronization primitives used when generating log, and reduce the log size to save storage space on board.

# Our Multi-core Deterministic Replay Tool

In this section, we present the workflow and the design of our multi-core deterministic replay tool.

## The Overall Workflow of the Replay Tool

The overall workflow of our deterministic replay tool is as follows. Our tool first performs static analysis on the application to locate shared global variables accessed by more than one task. Then it instruments the application under test to generate a instrumented version such that order of interleaved access to shared memories can be logged during the recording phase. We also make use of the hook functions to capture control flow information. During the recording phase, the user executes the application to generate execution logs within the hook functions. Finally, the replayer will analyze the execution logs, add breakpoints, and replay the logged execution through task scheduling and data rewriting.

In the following sections, we will present the design of the automatic source code instrumentation tool, the recorder, and the replayer in detail.

## The Automatic Source Code Instrumentation Tool

The source code instrumentation tool helps to automatically record the order of the interleaved access to shared global variables. Our tool first performs static analysis on the application source code to identify shared global variables accessed by more than one task. Then the tool instruments the instructions corresponding to the interleaved shared memory accesses with a wrapper function. The wrapper function in turn invokes a system call, which triggers the hook function to record the memory access orders into the log. Our instrumentation tool is based on the C Development Tool (CDT) plugin integrated with VxWorks development environment.

### Static analysis

In the static analysis phase, our tool scans the source code to locate shared global variables accessed by more than one task. The algorithm is shown as Table 1:

Table 1 Static Analysis Algorithm to Identify Shared Global Variables

|  |
| --- |
| **Input**: the source files of the application  **Output**: a set of shared global variables  **Preprocessing**: search the Abstract Syntax Tree of the application to get the maps below:  write2Entry a map where keys are global variables and values are the list of task entries writing them  read2Entrya map where keys are global variables and values are the list of task entries read them  entryCntmap where keys are entry functions, and values are the numbers of tasks using the entry |
| 1: **Set** sharedSet = 0; |
| 2: **for** **each** variable v **in** write2Entry.keys **then** |
| 3: writeEntrySet write2Entry [v] |
| 4: readEntrySet read2Entry [v] |
| 5: **if** readEntrySet is empty **then**  6: **skip** //at least one entry should read v |
| 7: **if** there is only one entry in both set **and** it is the same entry **and** entryCnt[v] == 1 **then**  8: **skip** //filter if only one task entry access it |
| 9: sharedSet.Add(v)  //add the variable to the shared set |
| 10: **return** sharedSet |

The algorithm can be divided into 4 steps:

1. In the pre-treatment step, our algorithm search *taskSpawn* within the source code to build the map ***entryCnt***.

2. Recursively build the map of global variables read (***read2Entry***) and written (***write2Entry***) in each task entry.

3. Search the entry point to build the set of global variables that may be accessed by more than one task (line 2 to 8).

4. Add the identified global variable ***v*** to the ***sharedSet*** (line 9).

### Automatic Instrumentation

At the application level, the shared memory access can be logged by either source code instrumentation or binary instrumentation. However, on VxWorks platform, binary instrumentation is hard to realize, because there is no readily available binary instrumentation tools. For portability, we build a source code level instrumentation tool to record shared memory access.

Automatic instrument tool instruments the read and write operations in source code related to the shared global variable to record the interleaved access to shared memories. And the instrumentation algorithm works as in Table 2.

Table 2 Algorithm of Instrumentation Tool

|  |
| --- |
| **Input**: the application source code and shared global variables ***G*** identified through static analysis  **Output**: the instrumented source code  1: **for** **each** C file **in** project **then** |
| 2: backup C file |
| 3: ***ast*** get abstract syntax tree of the C file |
| 4: Insert the ***include*** statement for the header file into the ***ast*** |
| 5: ***change***logrecording changes in ***ast***  6: ***sgv***shared global variable from static analysis |
| 7: **for** **each** statement **in** ***ast*** **then** |
| 8: **for** **each** node **in** statement **then** |
| 9: **if** the node is in ***G*** **then**  10: Wrapping the node and record in ***change log*** |
| 11: **if** ***main*** function is in ast **then** |
| 12: insert ***initInstrument()*** in the ***main*** function |
| 13: write back the ***change log*** to modify the C file  14: **return** the files changed |

The instrumentation algorithm work as follows:

First, it finds all C files in the application project and backs up them. Then it checks the including header files of every C file, and adds the instrumentation header file into them if it is nonexistent. After that, the algorithm iterates each statement of the AST tree for each file to instrument (wrapping) each read and write statement and generate a change log. The CDT will use the change log to update the source files. After that, the algorithm inserts the ***initInstrument()*** as the first statement of ***main*** function. The ***initInstrument()*** prepares the instrumentation log files Finally, the algorithm updates the AST (Abstract Syntax Tree) and the source code file based on the change log.

An example demonstrating the code before and after instrumentation is shown in Figure 1. The recording and the assignment are performed in our customized system call ***sharedMemSyscall***.

/\*wrapper function\*/

int lesReadInstr(int \*origin){

int target;

sharedMemSyscall(GLOBAL\_READ, origin, &target);

return target;

}

void lesWriteInstr(int origin, int \*target){

sharedMemSyscall(GLOBAL\_WRITE, &origin, target);

}

void sharedMemSyscall (int type, int \*from, int \*to){

int scn = SYSCALL\_NUMBER(2, 0);

return syscall(x, from, to, 0, 0, 0, 0, 0, scn);

}

/\*global is a shared global variable. The code before wrapping \*/

global = global +1;

printf("%d\n",global);

/\*The code after wrapping\*/

lesWriteInstr(lesReadInstr (& global) + 1, &global);

printf("%d\n", lesReadInstr (&global));

Figure 1 Example of Source Code Instrumentation

## The Design of the Recorder

As discussed in previous sections, the static analyzer first analyzes the original application to find all shared global variables. Then, the automatic instrumentation tool running in the host machine transforms the original application into an instrumented application by instrumenting the memory accesses to those shared variables.

C:\Users\liu\AppData\Local\Microsoft\Windows\INetCache\Content.Word\recording phase.emf

Figure 2 The Recording Phase

As shown in Figure 2, during the recording phase, the recorder uses hooks to log non-deterministic choices made during the execution of the instrumented application. We deploy a recording task for each core such that parallel memory accesses can be recorded simultaneously.

Instead of directly modifying the kernel source code, we use syscallEntryHook and syscallExitHook to record system API invocation information. We have also extended the arguments of syscallExitHook to get result information of system invocation such as the returned buffer of a ***read*** API call. Because system call may cause task context switches such that the sequence of returns does not match the sequence of entries, we record the synchronization events twice, once before the system call and once after the system call.

To record the parallel access to the shared memory, the recorder maintains a buffer and a task for each core to reduce the synchronization code between cores to only one atomic statement, which is getting the sequence id of the event. At the replay phase, we can perform merge sort on logs of different cores to order the events by sequence id.

The Recorder uses filters to reduce the log size. Firstly, we utilize an unused variable in TCB (Task Control Block) to mark the task of interest when we load the application and initialize the startup task. If the task of interest spawns a new task, we also copy the mark from the parent task to the child task. In this way, our hook will only handle the events generated by tasks with marks. This filtering mechanism helps the recorders to ignore the events from other applications as well as irrelevant system events.

Our static analyzer will filter unnecessary messages before program compilation and execution. We also design a buffer and an uploading task for every core to reduce the traffic between cores, and to filter unrelated events. In this way, we can achieve much better performance in the recording phase.

## The Design of the Replayer

C:\Users\liu\AppData\Local\Microsoft\Windows\INetCache\Content.Word\replaying phase.emf

Figure 3 The Replay Phase

The ***Replayer*** consists of the module on the target and eclipse plugin on the host. The kernel module loads the application in debug mode and registers hooks for processing events upon breakpoints. The plugin will parse the logs and control the application by setting breakpoints. We use WTX (Wind River Tools eXchange protocol) to communicate between the host and the target.

Within the hook function, our tool performs the following steps. Based on the input of the system call triggering the hook, our hook function first filter out unrelated system calls based on the argument. If the system call is about reading or writing shared data, the corresponding assignment is performed first. Then it writes task id and system call type to special global variable, which will be used by the ***Replayer***. If the hook is entry hook and the system call is to resume a task or access a file, it will modify the argument.

During the replay phase, a log parser first parses the logs generated from the recording phase. Based on the logged non-deterministic choices, the replay controller sets breakpoints in the application during replay, re-schedules tasks and writes back logged data for a faithful replay.

Table 3 The Replay Algorithm

|  |
| --- |
| **Input**: log files recorded and instrumented application  1: ***log***parse and merge the log files  2: install the replaying hook  3: set the first breakpoint at the position of first ***taskCreate***  4: load the application in debug mode  5: When first breakpoint hit, remaps taskid  5: **while** *!log.empty()* **then**  6: get next log entry  7: set breakpoint in hook based on next log  8: eventnext log’s information  9: waiting for breakpoint hitting  10: **if** event.type in <write, read, sysctl> **then**  11: write back logged parameter to API invocation  12: wakeup(event.nextTask) //schedule new task  13: **end if**  14: **end while** |

# Evaluation

## Evaluation Goal

In this section, we want to evaluate the effectiveness and overhead of our proposed deterministic replay tool for multicore debugging of VxWorks applications.

We use three sample multi-core applications for evaluation as shown in Table 4. In each row, we list the application name, the number of lines of source code, the number of tasks in the application, a description of the application and the concurrency bugs.

Table 4 Subject Applications

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Application Name | LOC | No. of Tasks | Description | Bug Type |
| rtpAccount | 80 | 3 | Two Accountant deal with the same book without protection | Data race |
| readSensor | 80 | 2 | Read sensor data and send it to net | Data race |
| router | 130 | 3 | 3-port router transmit messages | Data race |

The ***rtpAccount*** application simulates a typical data race scenario, where two tasks operate on shared global variables in parallel without proper protection of synchronization primitives. The ***readSensor*** application contains two cooperating tasks. The task1 of ***readSensor*** application reads sensor data from file, performs calculations, and puts data in the ring buffer. The task2 of ***readSensor*** application reads data from the ring buffer and sends the data to a simulated network file. The ***router*** application has 4 cooperating tasks. There are 3 tasks reading messages from 3 ports and one task transmits the message to the network or drops it.

To measure performance, we also measured the execution time and log size of each application with and without instrumentation for comparison.

## Experiment Setup

In our experiment, we used VxWorks 6.9 as the experimental platform. The host is running Microsoft Windows XP with the Intel Core i5-3470 and 4GB memory. We used the built-in simulator of workbench 3.3 to run VxWorks and the simulator is configured as a dual-core x86 target board with 1GB memory space. Note that we simulate sensor and network by files on host.

We also collect the running time and log size of the applications in recording phase with and without instrumentation to evaluate the impact of our tool on performance.

## Results and Analysis

### Replay Effectiveness

In this section, we want to evaluate whether our tool can effectively replay the multi-core VxWorks applications.

We will show the standard output (stdout) of those applications and the files generated after execution. Then we compare the results to verify whether the replay is successful or not.

*Recording phase*

start0 , 699999972 ,4

[account2] check\_result [999] yuan

[account1] withdraw [1] yuan in 33 s

[account0] deposit [2] yuan in 33 s result = 0

*Replaying phase*

start0 , 699999972 ,4

[account2] check\_result [999] yuan

[account1] withdraw [1] yuan in 33 s

[account0] deposit [2] yuan in 33 s result = 0

Figure 4 Stdout of rtpAccount

The standard outputs of the application ***rtpAccount*** in recording phase and replaying phase are shown as Figure 4. The results are the same in both the recording phase and the replaying phase, meaning that our tool has successfully replayed the multi-core application with data race.

The standard output of the application ***router*** is shown in Figure 5. We can see again that the outputs of application in the recording phase and the replay phase are the same, showing that our multi-core replay tool has successfully replayed the application.

*Recording phase*

………………………

[1] write 0 ENZybSdpQamO38ABeLvKPs9xVURqI05 with 35

[0] read msg to port 2

[0] write 2 4eqMSuYRmF6o9ywtOAI2JBl8CUgNsfZ with 35

[1] read msg to port 1

[1] read msg to port 1

[1] read msg to port 0

[1] write 0 fiYwxcXgBJhMHTP8S1bevot94jGnLQy with 35

[0] read msg to port 1

[0] write 1 j2biy9DVfB3HnOK8tYzGso14ZlgIR6X with 35

………………………

*Replaying phase*

………………………

[1] write 0 ENZybSdpQamO38ABeLvKPs9xVURqI05 with 35

[0] read msg to port 2

[0] write 2 4eqMSuYRmF6o9ywtOAI2JBl8CUgNsfZ with 35

[1] read msg to port 1

[1] read msg to port 1

[1] read msg to port 0

[1] write 0 fiYwxcXgBJhMHTP8S1bevot94jGnLQy with 35

[0] read msg to port 1

[0] write 1 j2biy9DVfB3HnOK8tYzGso14ZlgIR6X with 35

…………………………

Figure 5 Stdout of Router

The ***readSensor*** application output file is shown in Figure 6. We can see from the file contents that the results of the recording and replaying phase are exactly the same. This supports our conclusion that our tool can help replay debugging multi-core VxWorks applications successfully.

*Replaying phase*

21

89

57

47

43

57

54

1

35

64

……

*Recording phase*

21

89

57

47

43

57

54

1

35

64

……

Figure 6 Output File of readSensor

### Performance Evaluation

Now that we have successfully replayed the three applications, we want to further explore the impact on application performance of our source instrumentation. We compare the execution time in seconds with and without instrumentation for the three applications as shown in Figure 7. The x-axis shows the 3 applications while the y-axis shows the execution time in seconds. For each application, the blue bar represents the execution time without instrumentation while the red bar represents the corresponding execution time with instrumentation.

Figure 7 Impact of Instrumentation on Time

We can that for each application, the instrumentation indeed lead to additional overhead. The additional execution time for the instrumented applications is 1, 1, and 0.01 seconds, respectively. While this overhead may be insignificant for a traditional application, it may lead to changed program behavior for real-time applications. This is also frequently called the probe-effect. At one side, the results show that the overhead of our instrumentation is reasonable. At the other side, it also indicates that to use replay debugging for real-time applications with instrumentation, we must control the impact of the probe effect. For example, a common practice is to leave the probe (i.e., the instrumentation code) within the application and the system under test such that the probe effect is minimized.

Figure . Impact of Instrumentation on Log Size

To explore the impact of instrumentation on log size, we further measure the size of the log generated by our recorder for each application. As shown in Figure 8, the y-axis is the log size in KB while the x-axis is the application. We can see that the log sizes are consistently less than 1M bytes, which is not large. Although there are some variances in log size for different application, the absolute value of log size is small for most memory configurations on typical hardware. Thus, we can conclude that the log generated by our recording tool is small for practical use.

# Related Work

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

Scribe[11] is a transparent and lightweight deterministic replay tool for applications on commodity multi-core operating systems. It records system call, page fault and signals to reproduce the execution of recorded application. Scribe can replay application without instrumenting application source code. Our replay tool is similar to Scribe in event type and replaying approach. The difference is that we build automatic instrumentation tool at application level to record the interleaved shared memory access instead of using page fault for the disable page protection in user space.

DORA[25] can replay modified application with the execution log recorded before modification. DORA uses lightweight operating system mechanisms similar to Scribe. However, DORA can transit a task or a group of tasks from controlled replay to normal execution. DORA record more details of a system call to check whether it is the same system call occurred in the recording phase. Otherwise, the divergence must be due to addition or deletion of event. In this way, DORA can help reproduce, diagnose, and fix software bugs by replaying a version of a recorded application that is recompiled with debugging information, or patched to fix a bug.

ODR[1] records only part of the shared memory access with the Pin instrumentation tool. Therefore, it needs to search possible paths to reproduce the result in the replaying phase. It is a novel solution to reduce the overhead at the recording phase.

A similar solution is CLAP[10], which uses LLVM to perform the instrumentation and record non-deterministic choices. CLAP is a more portable solution due to its use of intermediate program representation. If VxWorks provides support for the clang complier in future versions, it may be a more suitable solution. In contrast, Castor[16] directly modifies the LLVM compiler to instrument the application when building a replay tool.

Lreplay[4] performs recording with the hardware support from the Godson-3 platform. The solution only costs 0.85 byte per 1000 instructions and takes 1.3% area of chip. It is an ideal solution to reduce time and space cost, but we cannot get the source of commodity CPU. On the other hand, we do not want to bind our solution to a hardware model, because we want our solution to be as general as possible.

Emulator and VM based solutions are also popular techniques for recording nondeterministic choices made during program execution. ReTrace[27] provides a solution based on VMware hypervisor. Karma[3] uses the GEMS[15] full system simulator to record conflicts by modifying the cache coherence protocol. And DeLorean[17] uses SESC[21] to for recording non-deterministic choices, which is similar to *Strata.*

PinPlay[22] builds their solution based on the dynamic binary instrumentation tool Pin[13]. Pinplay records the non-deterministic choices and writes them back in replaying phase. The space overhead of the log is one major problem of the PinPlay. LEAP[8] records the sequence of accesses to shared objects by instrumenting the java bytecode with Soot [12].

# Conclusion

Developing a deterministic replay tool on multi-core VxWorks is a challenging problem. We propose a feasible solution to replay multicore VxWorks applications on VxWorks 6.9 with our deterministic replay tool. We adopt static analysis, source code instrumentation and system level instrumentation to realize our tool. Our experimental results on 3 VxWorks multi-core applications show that we can successfully replay them with reasonable time and space cost.

##### References

1. G. Altekar, I. Stoica, ODR: output-deterministic replay for multicore debugging, in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09) . ACM, pp. 193-206, October. 2009.
2. D.F. Bacon, S. C. Goldstein. Hardware-assisted replay of multiprocessor programs, ACM, pp. 194-206, 1991.
3. A. Basu, J. Bobba, M. D. Hill, Karma: scalable deterministic record-replay, in International Conference on Supercomputing(ICS '11), 2011.
4. Y. J. Chen, W. W. Hu, T. S. Chen and R. Y. Wu, LReplay: A pending period based deterministic replay scheme, in ACM SIGARCH Computer Architecture News(CAN '10), pp 187-197, 2010.
5. J. Devietti, B. Lucia, L. Ceze, M. Oskin, Dmp: Deterministic shared-memory multiprocessing, in Proceedings of the 14nd Int’l Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, pp. 85-96, 2009.
6. G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. Chen, ReVirt: Enabling intrusion analysis through virtual-machine logging and replay, ACM SIGOPS Operating Systems Review, 36(SI):211-224, 2002.
7. Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, and Z. Zhang, R2: An application-level kernel for record and replay, in Proceedings of the 8th USENIX conference on Operating systems design and implementation. USENIX Association, pp. 193-208, 2008.
8. J. Huang, P. Liu, C. Zhang, LEAP: lightweight detlrepalerministic multi-processor replay of concurrent java programs, in Eighteenth ACM Sigsoft International Symposium on Foundations of Software Engineering(ISFSE '10), pp. 207-216, 2010
9. J. Huang, C. Zhang, LEAN: simplifying concurrency bug reproduction via replay-supported execution reduction, in ACM SIGPLAN Notices, 47(10):451-466, 2012.
10. J. Huang, C. Zhang, J. Dolby, CLAP: recording local executions to reproduce concurrency failures, in PLDI'03, pp. 141-152, 2013.
11. O. Laadan, N. Viennot, J. Nieh, Transparent, lightweight application execution replay on commodity multiprocessor operating systems. ACM Sigmetrics Performance Evaluation Review, 38(1):155-166, June. 2010.
12. P. Lam, E. Bodden, O. Lhoták, L. Hendren, Soot-a Java bytecode optimization framework, in Cetus Users and Compiler Infrastructure Workshop(CUCIW '11), pp. 1-11, October. 2011
13. 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), pp. 190-200, 2005.
14. C. Ma, X Long, B. Jiang, Z. Zhang. Replay Debugging of Real-Time Vxworks Applications, in Computer Software and Applications Conference(COMPSAC '15) , pp. 568-573, 2015.
15. M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ACM SIGARCH Computer Architecture News, 33(4):92-99, 2005.
16. A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, M. Rosenblum, Towards Practical Default-On Multi-Core Record/Replay, in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS '17), 2017.
17. P. Montesinos, L. Ceze, J. Torrellas, DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently, in International Symposium on Computer Architecture. IEEE Computer Society(ISCA '08), pp. 289-300, 2008.
18. S. Narayanasamy, C. Pereira, B. Calder, Recording shared memory dependencies using strata, ACM SIGARCH Computer Architecture News, 40(5):229-240, October. 2006.
19. S. Narayanasamy, G. Pokam, B. Calder, BugNet:Continuously Recording Program Execution for Deterministic Replay Debugging. ACM Sigarch Computer Architecture News, 33(2):284-295, June. 2005.
20. N. Nethercote, 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, pp. 89-100. 2007.
21. P. M. Ortego, P. Sack, SESC: SuperESCalar Simulator, January 2005. <http://sesc.sourceforge.net.>
22. 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, pp. 2-11, 2010.
23. K. Serebryany, T. Iskhodzhanov, ThreadSanitizer: data race detection in practice, in The Workshop on Binary Instrumentation and Applications(WBIA '09). ACM, pp. 62-71, 2009.
24. D. Sundmark, Deterministic Replay Debugging of Embedded Real-Time Systems using Standard Components, in Mälardalen RealTime Research Centre(MRTC '04), 2004.
25. N. Viennot, S. Nair, J. Nieh, Transparent mutable replay for multicore debugging and patch validation. ACM Sigarch Computer Architecture News, 41(1):127-138, March. 2013.
26. 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(ISCA '03), New York: ACM Press, pp. 122-135, 2003.
27. M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, Retrace: Collecting execution trace with virtual machine deterministic replay, in Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation(MoBS '07) 2007
28. 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, pp. 342-351, 2005.
29. PIN home page,: <http://rogue.colorado.edu/Pin/>.
30. Valgrind. <http://valgrind.org/>.
31. Sanitizers. <https://github.com/google/sanitizers>
32. Vxworks Application Programmers Guide 6.9. <https://zh.scribd.com/document/272059631/Vxworks-Application-Programmers-Guide-6-9>
33. Vxworks Kernel Programmers Guide 6.9.<https://zh.scribd.com/document/92980101/Vxworks-Kernel-Programmers-Guide-6-9>
34. Using CDT APIs to programmatically introspect C/C++ code <http://wiki.eclipse.org/images/c/c7/CDT_APIs_for_code_introspection.pdf>