MKtrace: An innovative debugging tool for multi-threaded programs on multiprocessor systems

Yusen Li, Feng Wang, Gang Wang, Xiaoguang Liu, Jing Liu
College of Information Technology and Science
Nankai University
liyusen007@hotmail.com

Abstract

In this paper, we propose an innovative debugging tool called MKtrace to help programmers identify bugs in multi-threaded programs on multiprocessor systems with little overhead. Unlike the traditional debugging tools, we use the trace log to analyze the cause of a crash or any abnormal behaviour. Bugs can be identified within the log file, not directly at run time. Also, a key advantage of MKtrace is its insignificant overhead.

The main idea of MKtrace is to monitor all the processes or threads from a program when they switch out in the kernel. We log the call stack of each thread, and then we analyze the log in the user space.

We implemented MKtrace on Linux AS3 with two processors and achieved promising results during our experiments.

1. Introduction

Multi-threaded programs may contain both sequential and concurrent errors, but deadlock and race conditions are specific to multi-threaded programs [13]. Because multi-threaded programs have non-repeatability and there is no synchronized global clock, there are few useful debugging tools for them. Most of them, like GDB [14], can only handle a single process. Breakpoints may be set for individual threads and the target application stops only if a particular thread encounters the breakpoint. So these debuggers allow interaction with only one thread at a time. Yet most of them have complex interfaces for users or have a high overhead. Probe effect is also typical for most of multi-threaded program debuggers.

The classic approach to debugging single-threaded programs involves stopping the program during execution, examining register values and the stack. Unfortunately, parallel programs do not always have repeatability. Even for the same inputs, the outputs will be different according to the executing environments.

According to the survey from CHARLES E. MCDOWELL and DAVID P. HELMBOLD [1], techniques for debugging concurrent systems have been organized into four groups:

1. Traditional debugging techniques applied with some success to parallel programs.
2. Event-based debuggers that view the execution of a parallel program as a sequence of events.
3. Techniques for displaying the control flow and distributed data associated with parallel programs.
4. Static analysis techniques based on dataflow analysis of parallel programs.

The traditional parallel debuggers are also called breakpoint debuggers. They are similar to a set of sequential debuggers, one per parallel process. They provide some control over program execution and provide state examination. However, these debuggers cannot tell us what happened during the interaction of several processes and have severe probe effect.

Event-based debuggers belong to the monitoring debuggers. They are often used to provide some replay tools for the multi-threaded program. This method may be more powerful than the traditional parallel debuggers, but the probe effect is a problem if the trace log is not recorded continuously and the overhead is often high [8, 9].

Static analysis tools avoid the probe effect entirely by not executing the programs. They find the potential bugs through analyzing the source code. They are powerful for data race, deadlocks and some semantic errors. However, the accuracy is not satisfactory and the computational complexity is often exponential [10].

There are four main types of techniques for displaying the control flow according to [1]: textual presentation of the data, time-process diagrams, animation of program execution and multiple windows.

MKtrace, which we will discuss in this paper, is an event-based debugger. The event of the MKtrace is the switching process of the multi-threaded program. The log we record during the event is the call stack of user space, including all the return addresses and the EIP. The main contributions of this paper are:

1. Introduction to the concept of concurrent program debugging.
2. Implementation of MKtrace on Linux AS3 with two processors.
3. Experiments applying MKtrace to various programs, achieving excellent results.

The rest of this paper is organized as follows. Section 2 describes the related work of concurrent program debugging. Section 3 describes the implementation details of MKtrace. Section 4 describes how we used MKtrace to analyze programs. We do some experiments in section 5 and conclude and present ideas for future work in section 6.

2. Related Work

As mentioned in section 1, there are three main approaches to concurrent program debugging: traditional style debuggers, monitoring systems and static analysis systems. Several thread debuggers have been developed for debugging various types of concurrent errors.

DTrace is a comprehensive dynamic tracing framework for the Solaris Operation Environment of Sun. It belongs to the traditional debuggers. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs. Tracing programs can be written using the D programming language. The language is a subset of C with additional functions and variables specific to tracing. D programs most resemble awk programs in structure. They consist of a set of actions rather than a top-down structured program. In a DTrace program, one or more probes are enabled. Whenever the condition for the probe is met, the action associated with the probe in the DTrace program is executed. So there is obvious probe effect in DTrace. There are also some famous debuggers that belong to the category of traditional debuggers like KDB [2] and ODB [4].

The most important type of event-based debuggers is replaying debuggers. The approach requires tracing some events during execution. One approach is to record the order in which processes interact [14]. Each process logs the order of shared memory or the order of synchronization operations [15]. There is also an approach that traces the data readings from every shared-memory location, but too much data is traced and the overhead is really high [16]. Linux Trace Toolkit is also an excellent event based debugging tool, but it can be only used on a single processor system.

Static analysis is the only approach that has no probe effect and it is usually used to detect two classes of errors in concurrent programs: synchronization errors and data race errors. The most famous analysis of concurrent programs is the one of Taylor and Osterweil [17]. Callahan and Subhlok [18] present another approach for determining which data dependencies observed in a sequential execution of a program are preserved in a parallel execution of the program. The static analysis is also used for intrusion detection [18]. It is shown how static analysis may be used to automatically derive a model of application behavior.

The usage of call stack information during debugging is discussed by Feng. H.H et al. [12]. They collect information from the stack at every system call, and draw a VTPath of the execution of the program. However it is only for single-threaded program and the overhead is high because it will hook all the system calls.

MKtrace is an event-based debugging tool. Unlike the debuggers we mentioned, MKtrace is very small but still very efficient. The “event” of MKtrace is the switching moment of threads and the trace log is the call stack information. We record the call stack information when the monitored thread switches out and analyze the trace log after execution. The overhead of MKtrace is light and the probe effect is also ignored. There are three parts in MKtrace: kernel engine, chardev and log analyzer. We implement MKtrace on Linux AS3 and prove its effectiveness through experiments.

3. MKtrace Implementation

The main idea of MKtrace is that we monitor all the processes or threads from one program when they switch out in the kernel. We record the call stack of the process or thread, and then we analyze the log in the user space. As we know, two threads may switch out in the same time on multiprocessor system, so there are some differences in the implementation between single-processor and multi-processor systems. We will illustrate the differences in every part of MKtrace.

3.1. Kernel Engine

Our purpose is to log the call stack when the process or thread is switching out. We do this in the kernel space. When switching out, the process will call *context_switch*, we can get the user space ESP and EBP from the kernel stack as shown in Figure 1. So we can trace the user space stack according to the EBP, and get all the functions' return addresses of the call stack. The relationship between the EBP and the call stack is shown in the Figure 2. We also record the EIP of user space according to which we can get the line number where the execution stopped.

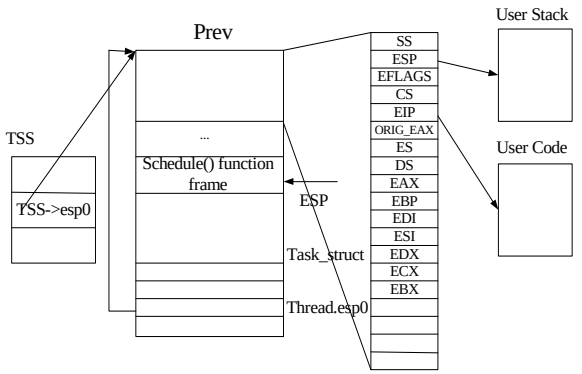

Figure 1. **Kernel stack after any system call, before context switch**

In order to monitor all the threads of one program, we first mark the main thread, and if the main thread forks out other threads, we can recognize the marked thread and mark the new thread.

To log the stack information during switch, we use buffer to cache the log and then we fetch the log from user space by a chardev. In a single processor system, we can use one circular buffer to cache the log and the log must be sequential. But in a multiprocessor system, two processors can call the schedule separately, so two threads can switch out at the same time. If we also use one buffer to cache the log, we should lock the buffer when we write it, so the schedule becomes sequential and the performance may drop significantly. In order to improve the performance, we provide one buffer for each processor. Each processor can write the log into its own buffer, but in order to analyze the log, we use a global clock to record the sequence of the log. We have to add a lock when we access the clock, so there will be probe effect. For example, two threads on different processors switch at the same time, but only one of them can get the lock of the clock, so the execution sequence will be changed. But in fact it only has little impact and will not affect the correctness of the program.

When each processor starts to write the trace log, it should increase the global clock first, and then write the log with the clock to its own buffer. As in the single processor system, we also write all the logs into a user space log file, but before we analyze it, we should resort the log by the clock in the multiprocessor system.

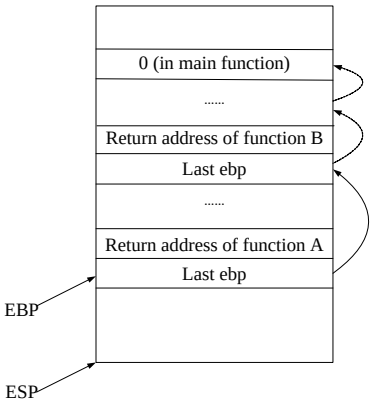

Figure 2. **User space call stack**

3.2. Chardev

The Chardev plays the role of interface between the user space and the kernel.

First, when we start using MKtrace, we should notice the kernel which thread should be monitored. We implement it by the function *device_ioctl* in chardev. We get the *task_struct* structure of the thread to be monitored and mark it. We test the flag of the *task_struct* structure in the *copy_process* function, if the parent thread forks out another thread, we also mark the new thread.

Then, once started, buffers in the kernel will be filled with logs. We should fetch the logs from the kernel and write them into a log file in the user space. We implement it by the function *device_read* in chardev.

The chardev is only an interface between the user space and the kernel. We should also provide a user interface by which users can use MKtrace. We implement it by a user space program called *mktracer*. When we want to use MKtrace, we can use the command: *mktracer program*. There are two threads in *mktracer*. One of them notices the kernel which thread we will monitor by the *device_ioctl* of chardev. Another thread will call the *device_read* of chardev to get the log repeatedly.

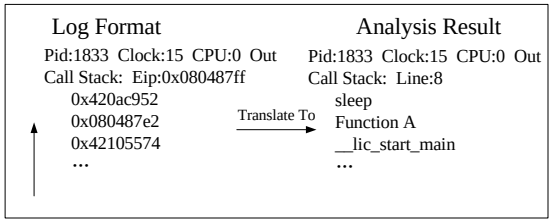

Figure 3. **Formats of log and analysis result**

3.3 Log Analyzer

The logs we get from the kernel are actually the return addresses of functions. The format of the log is shown in Figure 3. "Pid" is the process ID being monitored, "Clock" is the global clock value, "CPU" represents which processor is working and "Out" represents the process is switching out. "Eip" is the current IPC, and the addresses listed below are the call stack.

The logs come from different processors but they have a global clock.

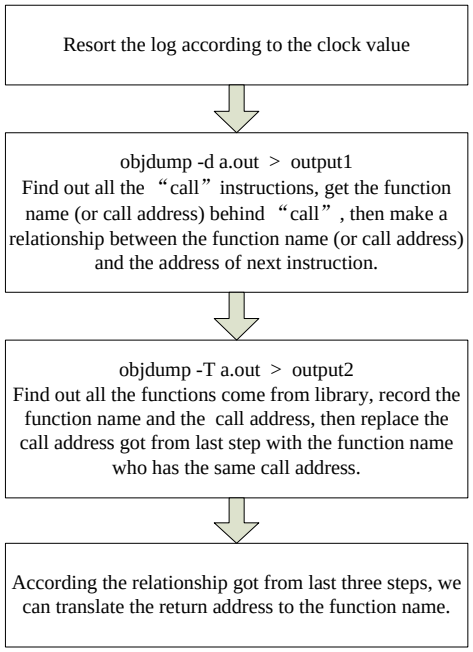

Figure 4. **The flow of log analyzer**

First we should sort the logs according to the clock. However, we cannot get the function name because we only know the return addresses. To solve this problem, we analyze the executed file (ELF) with *nm* and *objdump* commands. We can disassemble the executive code, and get all the "call" instructions and record the function names. We also record the next instruction of the "call", which is just the return address of the called function. According to this information, we create a map that records the relations between return address and the function name. If a program compiled with *gcc –g* option, we can also get the line number from the result of command "*objdump –d –l*" according to the EIP address.

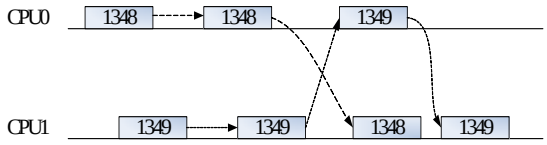

Figure 5. **Animation of the program execution**

4. What can MKtrace do?

Though MKtrace is a simple debugging tool, it can help programmers to identify various bugs quickly from the log. We will illustrate some usages of MKtrace by some examples. MKtrace cannot only find some traditional single process bugs but also identify some familiar concurrent errors. We do some experiments in next section.

4.1 Execution Animation

If we both monitor the switching out and switching in process of a thread, we can get the execution animation of the multi-threaded program. For example, Figure 5 shows a program with a two thread execution animation on a dual-core system. In Figure 5, two lines represent time axis of two processors, "1348" and "1349" are tow threads. The blocks denote that threads are executing on the processor.

4.2 Dead Lock Detection

When a deadlock occurs, thread A is waiting for the resource owned by thread B, while thread B is also waiting for some resources owned by A. None of the two threads will release the resources until they get the resources from the other side.

MKtrace can tell the programmer where the two threads stopped precisely. We can find that, if there is a deadlock, the log record of the two threads will not change after a fixed time. We can also get the location of the deadlock from the line number of the source code given by the log.

4.3 Data Race Analysis

```
Thread1:
void threadfunc1 (void)
{
    if( i!=0 )
    {
            i = 1;
    }
    while(!j); //line 8
    i = 1;
}
Thread2:
void threadfunc2 (void)
{
    if( j!=0 )
    {
            i = 1;
    }
    while(!i); //line 17
    j = 1;
}
```

Figure 6. **Pseudo code of dead lock**

Unlike other data race identification tools, MKtrace cannot find the data race from the execution process directly. However, if a programmer can anticipate some potential data races and insert some asserts into the source code, MKtrace can help him identify the cause of a data race. The reason is that MKtrace can draw an execution path of all the threads. If thread B switches in after thread A switches out and illegally modifies the shared memory, then, the thread A switches in again and the assert fails. At this time, MKtrace can tell the programmer which thread has switched in during the time out of thread A. Programmer can get some illuminations from the result.

4.4 Anomaly Detection

We cannot only monitor the switching out process but also the switching in process. If there are some intrusions during the switching out time, the call stack has changed when the thread switches in again. We can analyze the differences within the stack information to identify the problem.

5. Experimental Evaluation

In order to give some representative examples, we apply some small programs with the specific bugs mentioned above to MKtrace. We will show the interesting logs in this section.

5.1 Dead Lock Detection

The program showed in Figure 6 has a deadlock. The initialized values of *i* and *j* are 0.

```
Pid:1812 Clock:1 CPU:0 Out
Call stack: Line:8
      threadfunc1
      __libc_start_main
Pid:1811 Clock:2 CPU:0 Out
Call stack: Line:17
      threadfunc2
      __libc_start_main
Pid:1812 Clock:3 CPU:1 Out
Call stack: Line:8
    …….
Pid:1811 Clock:8 CPU:0 Out
Call stack: Line:17
    ……
```

Figure 7. **Analysis result from MKtrace**

When the program runs, threads will be blocked at line 8 and line 17. At that time, we stop the program and analyze the log from MKtrace, the result is shown in Figure 7.

We can find the problem immediately, because both threads repeatedly log the same line and can not go ahead.

MKtrace is unable to distinguish the dead lock from a long time blocking, so you should stop the program personally if you doubt that there may be some problems in your program.

```
Pid:1833 Clock:4 CPU:0 Out
Call stack: EIP:0x08048780
      0x420ac952
      0x080487e2
      0x42105574
      0x08048581
      ……
Pid:1833 Clock:5 CPU:0 In
Call stack: EIP:0x08048780
      0x420ac952
      0x080487e2
      0x42105574
      0xc03059a4
```

Figure 8. **MKtrace log of intrusion**

```
int * shint; //shared memory
int main()
{
    pid_t child;
    int ret = 0, status;
    char * test;
    shint = (int *)mmap(NULL,sizeof(int),PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS,-1,0);  //shared memory
    *shint = 0;
    child = fork();//line 10
    if(child > 0){
        funcb();
        wait(&status);
    }else{
        sleep(3);//line 16
        funca(1);
    }
    return 0;
}
void funca()
{
    assert((*shint) == 0);
}
void funcb()
{
    *shint = 1;
    sleep(1);//line 28
}
```

Figure 9. **Code which has data race**

5.2 Data Race

We will give a simple example of the data race case and explain how MKtrace can help us. The program shown in Figure 9 has a data race.

The data race is proved to happen when the assert fails and the program exits. We can see that the process with PID 20656 has switched in twice during the switching out of the process with PID 20657. So we can suppose that process with PID 20656 modified the shared variable. We can also find this by the animation of the execution of Figure 10.

5.3 Anomaly Detection and Stack Smashing

Stack smashing can also be involved in this section, because the stack is smashed when an anomaly occurs. In order to find the intrusions, we should monitor the process's switching in. We create two threads in one program. When thread A switches out, thread B plays the role of an intruder. It will modify the user space EBP of thread A, and insert another address to replace the original EBP. When thread A switches in again, it will find that the user space call stack has changed. Part of the log is shown in Figure 8.

```
Pid:20656 Clock:1 CPU:0 Out
Call stack: Line:10
    fork
    __libc_start_main
Pid:20657 Clock:2 CPU:0 Out
Call stack: Line:16
    sleep
    __libc_start_main
Pid:20656 Clock:3 CPU:1 Out
Call stack: Line:28
    sleep
    funcb
    __libc_start_main
Pid:20656 Clock:4 CPU:0 Out
Call stack: Line:13
    wait
    __libc_start_main
Pid:20657 Clock:5 CPU:0 Out//exit
Pid:20656 Clock:6 CPU:0 Out//exit
```

Figure 10. **Analysis result of Figure 9**

We will find that, there is an unexpected address in the log of switching in. The address 0xc03059a4 is the return address of intrusion code.

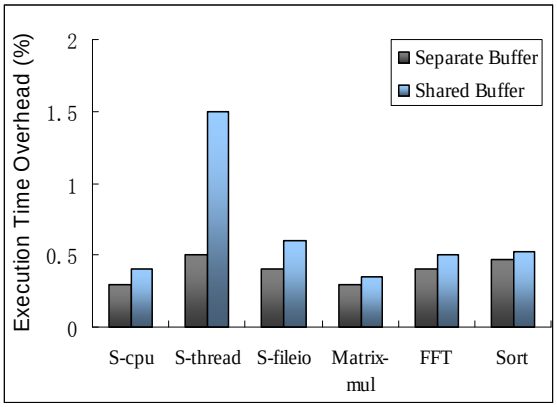

Figure 11. **Overhead of MKtrace**

The log file size may be very large, we can use a circular file to avoid this situation. If the circular file is full, we truncate the log file. Though some information

will be lost, the information has no effect on our analysis.

5.4 Neglectable Overhead

The main attraction of MKtrace is its small overhead. So many programmers flinch from the high overhead of some other debugging tools for replay or anomaly detection. MKtrace however is efficient, we only need to record the call stack information when the processes switch out. These instructions only access some memory locations and will not take much time. Figure 11 shows the overhead of the MKtrace. "S-cpu","S-thread" and "S-fileio" denote SysBench with cpu, thread and fileio test modes.

We use SysBench, Matrix Multiply program, FFT and Sort program to evaluate the overhead of MKtrace. SysBench is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load. We chose three test modes of SysBench: cpu, thread and fileio. We run these programs on the Linux AS3 OS in the virtual machine with Pentium(R) D CPU 2.66GHz and 1.00G memories. It is obvious that the overhead of the MKtrace is less than 1%. By comparison, some debugging or replay tools have 17%-88% or over 100% overhead [15].

It is interesting that the overhead of the version of MKtrace with one shared buffer is almost the same as the overhead of the version of MKtrace that uses one buffer per CPU. Instinctively, MKtrace with one shared buffer will have more overhead because many processors should contest for the lock of the shared buffer. But in fact, when the thread number is small, the event that two threads switch out at the same time will happen very rarely. So, each processor can get the lock if it needs. When thread numbers increase, the probability of two threads from one program switch out simultaneously will increase, so the lock increases the execution time. The S-thread in Figure 11 denotes the threads test mode of SysBench, and we choose 200 threads, so the overhead of shared buffer is much higher.

6. Conclusion

We discussed the problems encountered with concurrent programs debugging and existing approaches to debug multi-threaded programs. Then, we proposed an innovative debugging tool MKtrace for multi-threaded programs on multiprocessor systems. As an event-based debugging tool, we use "switch process" as the "event" and log the call stack information of the thread. MKtrace has light overhead and ignores probe effect. We implement MKtrace on Linux AS3 OS with two processors. We also discussed the implementation differences between single-processor and multi-processor systems.

We applied MKtrace to some representative programs and achieved excellent results. We can find the position of deadlock immediately from the MKtrace log. MKtrace also gives us suggestions to find data races. If there are intrusions during switching out, MKtrace can also tell us the intruder's function address. We can generally get the execution animation of multi-threaded programs.

MKtrace can be used for many thread systems as long as the thread system has the kernel structure and the kernel charges for the switching process. In the future, we will develop on some other platforms like MPI and invent more features for MKtrace in order to make it even more powerful.

7. Acknowledgements

This paper is sponsored by NSF of China (No.90612001), Science and Technology Development Plan of Tianjin , (No. 043800311, 043185111-14) and Nankai University Innovation Fund and ISC.

8. References

[1] Charles E. Mcdowell and David P. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys (CSUR)* Volume 21, Issue 4, 1989. pp. 593-622.

[2] PA Buhr, M Karsten, J Shih, "DB: a multi-threaded debugger for multi-threaded applications", *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Philadelphia, Pennsylvania, United States, 1996, pp. 80-87.

[3] Andrew P.Tolmach and Andrew W.Appel, "Debuggable concurrency extensioins for standard ML", *Technical Report CS-TR-352-91*, Princeton University, Dept. of Computer Science, 1991.

[4] ODB,
http://java.sun.com/features/2002/08/omnidebug.html

[5] Detlefs, D.L. Leino, R.M., Nelson, G., and Saxe, J.B., "Extended static checking", *Tech. Rep. Res. Rep.* 149, Systems Reaearch Center, Digital Equipment Corp, Palo Alto, Calif, 1997.

[6] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson, "Eraser: a

dynamic data race detector for multithreaded programs", *ACM Transactions on Computer Systems (TOCS) archive* Volume 15 , Issue 4 (November 1997), 1997, pp. 391-411.

[7] Ahmed K. Elmagarmid, "A survey of distributed deadlock detection algorithms", *ACM Transactions on Computer Systems (TOCS)* Volume 15, Issue 4 (November 1997), 1997, pp. 391-411.

[8] T.J. LeBlanc J.M. Mellor-Crummey, "Debugging parallel programs with instant replay", *IEEE Transactions on Computers* Volume 36 ,Issue 4 (April 1987), 1987, pp. 471-482.

[9] Michiel Ronsse, Koen De Bosschere, Jacques Chassin de Kergommeaux, "Execution replay and debugging", *Fourth International Workshop on Automated Debugging (AADebug 2000).*

[10] N Feamster, H Balakrishnan, "Detection BGP Configuration Faults with Static Analysis", *Networked Systems Design and Implementation*, 2005.

[11] Sougata Mukherjea John T. Stasko, "Applying algorithm animation techniques for program tracing, debugging, and understanding", *International Conference on Software Engineering Proceedings of the 15th international conference on Software Engineering* Baltimore, Maryland, 1993, pp. 456-465.

[12] HH Feng, OM Kolesnikov, P Fogla, W Lee, "Anomaly detection using call stack information", *Security and Privacy*, May 2003, pp. 62- 75.

[13] Farchi, E. Nir, Y. Ur, S, "Concurrent bug patterns and how to test them", *Parallel and Distributed Processing Symposium, 2003. Proceedings*, April 2003, pp. 22-26.

[14] Richard H. Carver and Kuo-Chung Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables", *6th Intl, Conf. on Distributed Computing Systems*, Boston, MA, May 1986, pp. 428-432.

[15] K. C. Tai, Richard H. Carver, and Evelyn E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution", *IEEE Trans On Software Engineering*, January 1991, pp.45-63.

[16] Douglas Z. Pan and Mark A. Linton, "Supporting Reverse Execution of Parallel Programs", *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, may 1988, pp. 124-129.

[17] Taylor, R. N, "Debugging real-time software in a host-target environment", Tech. Rep.212, Univ. of California at Irvine, 1984. [18] Waqner, D. Dean, R, "intrusion detection via static analysis", *Security and Privacy*, California Univ. Berkeley, CA, 2001.

[18] Waqner, D. Dean, R, "intrusion detection via static analysis", *Security and Privacy*, California Univ. Berkeley, CA, 2001.