A Record-and-Replay Fault Tolerant System for Multithreading Applications

Cristescu Ioana
Faculty of Computer Science
Technical University of Cluj Napoca
Email: Domnina.Cristescu@gmail.com

***Abstract*—This paper describes a tool used as a fault tolerant system for generic Linux programs. It is based on the record and replay technique and it uses the Utrace framework. Build on a single process debugger, the system provides fault tolerance in case of failure by stopping, for multithreading applications. While both the technique and the framework are specialized for debugging, the article shows their suitability for assuring failure protection to a large range of applications. The tool is design in order to supply continuous availability for servers.**

The main design of the system and some of the encountered technical problems are presented, also a set of examples and explanations is provided.

***Index Terms*—record and replay, fault-tolerance, multithreading applications**

I. Introduction

The tool presented was built as a fault tolerance system for multithreading user space applications. It is implemented on the record and replay model. At its basis it was a single process debugger constructed on the Utrace framework. From there it was developed to support multiple processes and to allow the programs to continue execution once brought to their previous failure point. It is addressed mostly, to projects that run for long period of time or those that create, during their execution, complex internal states that are difficult to replicate. It handles generic Linux programs, without requiring a specific programming language.

The main difficulty in tracing the programs is their nondeterministic execution. In order to record and deterministically replay the execution, the main approach is to consider the interaction between the programs and the kernel. This is done by capturing and recording the system calls. At recording, the traced application generates a sequence of events which must be followed at replay. Nevertheless, the replay phase is executed considerably faster since it only simulates the system call, which execution is stopped once it reaches the kernel space. Just as important is respecting the scheduling order during replay. The framework used offers access to the developer in several points during the thread's execution such that threads can be controlled. Therefore one can impose during replay the order in which threads will run.

We start by discussing similar tools available. In section 3 a short description of the chosen implementation is presented. A better understanding of the tool and of several design decisions is achieved through examples in section 4. As a conclusion there is a discussion of whether such a tool is necessary and the further development.

II. Related work

There are several abstract models theorized for detecting failure [1]. The current system is applicable on the fail-stop fault model, representing systems that fail by stopping. If current implementation is used in a distributed system, the model implies that the crashing nodes would stop producing output or communicating with the other nodes. In case of the fail-stop fault model, there are several techniques used. Checkpoint and restart [12] [13] is based on

periodically saving the internal state of the system. Another widely used solution is record and replay, the one used in the current implementation. Also called log-based rollback, it creates a structured view of the processes execution. The record and replay approach has several implementations, most of which have as purpose deterministic replay debugging.

Bugnet [2] records the registers content periodically and in between writes logs of the system call. It was one of the first tools using record and replay but it requires working on a special API. Our system, in comparison with Bugnet is application transparent and does not require rewriting the code. Jockey [3] was built as a debugger for distributed programs. It supports system calls and CPU operations, and it was implemented as a user space library making it safer, but less applicable, unlike the system presented, built as a kernel module and addressing a wider range of applications. Both these tools are used strictly from a debugger point of view. While they are discussed here for their proposed solutions, constructing a debugger was not our purpose.

Flashback [4] works at the kernel level by intercepting all interactions between the user space programs and the kernel such as system calls and memory accessing. It uses in its implementation the system call tables routines called by the kernel. Its main drawback is that it cant offer support for signals. Our tool does not neither support signals, but it is build on a framework which allows this improvement to be made.

Echo [5] is the most closely related tool to the current one. It is a kernel extension, it intercepts the system calls and supports multithreading, but it still remains a debugging tool solely. It logs the return value and auxiliary data when a system call is executed. Each thread has a performance counter based on

which the context switch is supported. At replay the processes are forced to stick with the recorded schedulers order. The reason Echo is the most closely related tool to our solution is that it uses the same mechanisms and techniques, but it remains a debbugger.

The R&R kernel module [8] is a tool developed by Lucian Cocan, at Technical University of Cluj Napoca. The fault tolerant system presented can be seen as an enhancement of the R&R tool. The driver is a debugging tool offering services similar to the Echo system. Its main disadvantage is that it only supports single processes application.

In a different approach, instead of tracing the system calls, the memory accesses are recorded [3]. While this method offers support for more generic applications, it requires a special compiler and considerably large memory.

The Linux OS supports debugging through the Ptrace mechanism [11]. This tool enables a parent thread to trace and control the thread process execution. While it is integrated in the OS and widely used, Ptrace has some disadvantages, mainly it does not have a consistent support for multithreading, which makes it unsuitable for our purposes. Using the same interface and an easy integration in the operating system, the Utrace framework [7] is a higher quality debugging tool and it was chosen for the current implementation.

III. The Fault Tolerant System

The system is build such that the recording phase interferes as little as possible with the programs execution. The only moment it does take control of the execution flow is when it sets up the memory.

A. Record and Replay using the Utrace Framework

Linux has a built in aid for debuggers, called Ptrace [6]. It is used by tools such as *gdb* and *strace*, but it is considered not suitable for multithreading applications because of the high overhead associated with a context switch. Other limitations are its lack of documentation, of supportability (it behaves different according to the operating system), and also the existence of an unjustified overhead for accessing the internal structure of the Ptrace client.

The Utrace framework [6] [7]is an enhancement over Ptrace and while it applies to the same interface, it replaces Ptrace entirely. The framework offers to its client a tracing engine which is associated with one or more threads. It is the tracing engine that is responsible for calling event triggered functions. These functions, the callbacks, cover a range of points in the execution of the thread, where the debugger would like to interfere. Such callbacks occur in safe places, when the system call either enters or exits the kernel, where no locks are held and the threads internal structure can be accessed. Furthermore, the framework allows for direct access and modification of the registers and of the internal structures associated with the threads. From within a callback the utrace client is also able to control the thread execution by injecting signals.

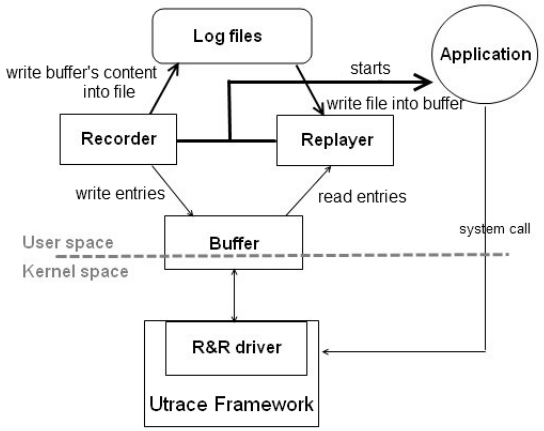

Fig. 1. Single Process Debugger

The Utrace clients run in the kernel, as a kernel module inserted before the tracee's execution. It offers better performance than Ptrace and support for multithreading applications.

B. Single process debugger

For the constructing of our fault tolerant system we started from [8] and extended it. It was constructed over an utrace client single process debugger. In order for a process to be traced, the system's main process has to become its parent. The tool consists of three components illustrated in Figure 1. These components are user space applications (the recorder and the replayer), and a kernel module (R&R driver). Their functionality is as follows:

- *The record* sets up a tracing engine, allocates the needed memory, creates the logs and starts the application.
- *The R&R driver* is the actual utrace client and the primary actor at both record and replay. In the first phase it observes the traced process and writes in the logs any useful information. At replay it controls the processes execution according to the previously written data.
- *The replayer* lets the driver know the replay phase has started and runs the tracee application.

While the tool at this step of development is strictly a debugger it presents one characteristic that makes it suitable as a basis for a fault tolerant tool. In case of a crash the kernel module would not be able to keep this data neither would the memory used by the kernel during runtime. Therefore this data has to be written into a permanent storage. Once the application has finished execution, for diverse reasons, the recorder saves the logs into a permanent file. From there, the replayer will write them back into the drivers memory. These components can be seen on Figure 1, where the buffer represent the memory written during runtime and the logs the permanent storage files. Managing the logs is vital to the correct functioning of the system and it implies the described behavior.

The logs are structured in entries containing the system call number, the returned value and other auxiliary data if

necessary.

C. Multithreading Fault Tolerant System

There are two directions which the previously presented debugger must take in order to become a valid candidate for a fault tolerance tool. First, it has to allow the application to continue its execution after a crash while persisting to offer its protection services. That is even if an application fails several times it still can be replayed up to the failure point. The second improvement is to make the debugger suitable for multithreading applications.

For the first requirement, the replayer has to be able to change its function once the replay phase finished. Since that is a difficult task, to announce a user space process of a kernel event, a workaround is possible. The replayer writes all the data necessary for the driver, it changes to record mode, and then it starts the application. Nevertheless, as it was presented above, the functioning in the record mode is not entirely appropriate. The recorder component proposed in the previous section only writes the logs into a permanent storage when the application finished execution. The recorder needs to be notified by the childs death and it has to write data to the files, but in the case of a system failure, it might not be possible. Proposed solutions for this issue are either an optimistic one [9], the logs are periodically saved into a permanent storage or a pessimistic one [1], in which the data is written directly into the file. The current implementation writes often, twice every system call, therefore the optimistic approach was chosen.

In order to offer support for multithreading, we have to keep the order in which threads run during the record phase. Knowing the order is a necessary step in respecting that order during replay. Our solution is to write in a log for each system call the identifier of the process which calls it. The log used by the R&R driver will be structured as presented in Figure 2. During the replay phase the threads read the logs as means to determine the execution order. Individually, each thread reads ahead in the log, the entry associated with the following system call. If the identifier written is associated with its own identifier then its his turn execute, hence to call the next system call. This association is necessary because the thread's identifier is allocated by the operating system and changes from the recorder's and replayer's execution. If, however, the identifier of the running thread is not the one expected then a context switch has to be induced. The technique used is to put the thread to sleep, which results in relinquishing the CPU. When it wakes up, it checks again if the identifier of the current log entry corresponds to its own and until his turn has come it will continue to sleep. Given the nondeterminism of multithreading application, it can be assumed that the entries in the log are not consistent. That implies that entries of different system call can interfere. One entry in the log is written in two steps, each step in a different callback, as shown in Figure 4. The case in which those entries interfere is exemplified in the following section. However interference could appear at the level of each field. The Utrace framework is the one assuring that this does

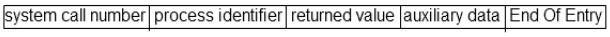

Fig. 2. One entry of the logs

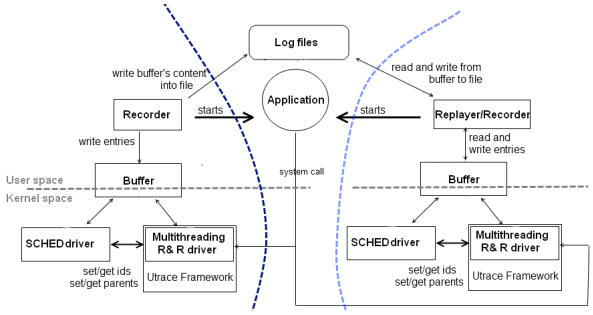

Fig. 3. Multithreading Fault Tolerance System

not happen by executing each callback atomically. Along with the identifier of all traced threads, other information has to be kept such as the parent-child relationships. While this is easy to maintain in the kernel module, just like the system calls buffer, it has to be written to a file in order to make it available to the replayer. Using the same file is not an option since it is written continuously by recorder. Consequently a new driver is needed that manages the scheduling information. We will call it the SCHED driver. It keeps in its buffer data about each process: identifier, parent and its creation order. The new system's architecture is shown in Figure 3.

After we had a closer look at how multithreading was implemented other issues arise. They will be presented in the following section since it was considered that associating those problems with an example would give a clear perception of the matter.

D. Problems encountered and solutions

One of our goals is for the tool to be transparent to the application. That is mostly accomplished with one exception. Due to an optimization in the glibc library the *getpid* system call does not get executed in the child thread. Given that glibc is implemented at the user space level, access to it is not possible. The solution is to call from the application the system call without allowing the glibc to apply a wrapper.

Communication has to be allowed between the driver and the recorder, replayer applications, and also between the R&R driver and the sched driver. For both there are solutions offered by operating system. The first communication is done through the ioctl [10] system call. It is called from user space and it

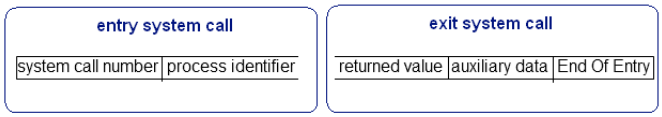

Fig. 4. Entry written in 2 steps

has as one of its parameters a command number. This allows several commands to be transmitted to the kernel which will be able to differentiate between them. The two drivers have to communicate both during record and replay by allowing the R&R driver to call functions from the sched driver. This is implemented in Linux through the symbol table [10] to which the kernel modules have access. The sched driver exports its functions into the symbol table.

General treatment for a system call at replay is to stop its execution and set the return value from the logs. The solution is applicable to all system calls in a single thread application, but for multiple threads one exception has to be made. The *clone* and similar system calls are executed and a tracing engine is attached to the child process.

IV. Testing scenarios

The following tests represents scenarios of interest from the implementation's perspective, that can occur during a multithreading application. The system calls presented are called by the user space application and controlled by the kernel drivers. Notice the use of system call such as *fork*, *vfork* and *clone* for the creation of processes, which in the Linux OS are sharing a similar implementation with threads.

The system supports failure by stopping scenarios. Our failure imitation was injecting a SIGKILL signal to the application, pressing the 'Ctrl+C' keyboard combination. Therefore in the following examples a '^ C' symbol appears standing for the SIGKILL signal occurence.

A. The Vfork System Call

The sequence tested is shown below.

```
pid=vfork();
if (pid==0){
  printf("CHILD: %d\n", syscall(SYS_getpid));
  exit(0);
}
printf("PARENT: %d\n", syscall(SYS_getpid));
```

For system calls such as *fork* and *clone* the parent is the first to exit the system call. Another system call might be executed by the parent immediately after or the child process might start running by exiting at its turn the *clone* or *fork* system call. It is easy therefore for the parent to attach a tracing engine to the child. In the case of the *vfork* system call, the parent exits the system call only after the child has finished execution. This is due to the fact that *vfork* was built as an optimizated alternative to an old implemetation of *fork*, which used to copy the entire structure of the parent process as means to create the child. *Vfork* is constructed such that both the parent and the child share the same structure and, in order of avoiding synchronization problems, the parent waits until the child finishes execution [14]. In Ptrace for instance, this implementation makes it impossible for the parent to attach a tracing engine to child. The Utrace framework resolves this issue, while previous debuggers did not, by creating a callback function for all *clone* related system calls, right before neither the parent nor the child process exits the system call. There both processes structure can be handled safely.

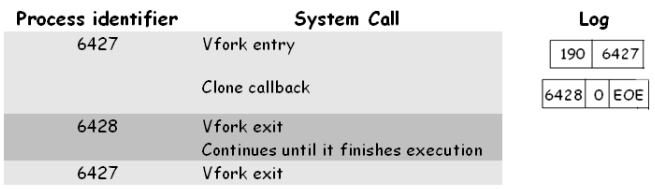

Fig. 5. Vfork example: Process 6427 writes in the log the system call number - 190 for *vfork*, its identifier, the returned value - the child's id 6428, the auxiliary data and 'End Of Entry'. In the clone callback an engine is attach to the process 6428 and afterwards both 6427 and 6428 resume execution.

An example highlighting the issues discussed is shown in Figure 5.

B. Sleep

The sequence tested is shown below.

```
pid=vfork();
if (pid==0){
  for(i=0; i<2; i++){
    gettimeofday (&tv, NULL);
    printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
    sleep(1);
  }
  exit(0);
}
printf("PARENT: %d\n", syscall(SYS_getpid));
for(i=10; i<12; i++){
  gettimeofday (&tv, NULL);
  printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
  sleep(1);
}
```

It is one system call where a context switch is bound to happen. Therefore it can be used as a supplementary point of synchronization for the replayer. When a context switch occurs, the offset at which the process starts writing is recorded and at replay that offset is restored. Therefore each process reads from the log at the point it wrote. Another moment during the application execution that such synchronization takes place is when a process dies.

When sleep is called at replay the process has to relinquish the CPU but it wakes up earlier than at record. When it does it has to verify that it is in fact its turn. Before understanding how this verification is done one other digression is needed first. The entries in the log files are written in two steps:

- When entering the system call. It writes the system call number and the identifier of the thread;
- When exiting the system call ends. It writes the returned value and the auxiliary data.

If the two parts of the entries are separated the identifier of the owner of a system call can no longer be verified. This is what happens for a sleep system call. Moreover, when a sleep resumes it has to verify the next system calls identifier because the log for the current system call no longer has identification data. Figure 6 highlights this situation. If that happens when system calls are no longer executed, the process might continuously loop. This situation occurs when processes wake up only in order to finish execution. For the two reasons presented above the sleep and other similar system calls have their corresponding entries in the logs containing twice the process identifier.

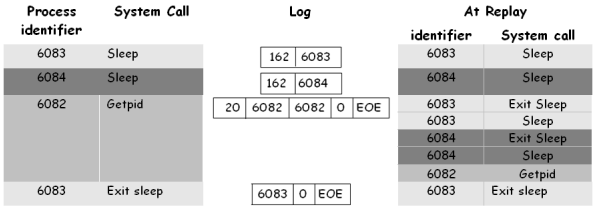

Fig. 6. Sleep example: At record, while calling *sleep*, the processes 6083, 6084, write the system call number and their identifiers. *Sleep* is then executed, allowing 6082 to write the entry corresponding to *getpid* - the system call number 20 and the identifier as the calling process and as the returned value. Once the 6083 resumes execution it completes the *sleep* entry.

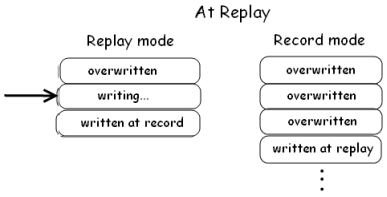

Fig. 7. Writing the Logs at Replay

C. Second Time Replay

This scenario is tested on the following code:

```
pid=vfork();
if (pid==0){
  for(i=0; i<10; i++){
    gettimeofday (&tv, NULL);
    printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
  }
  exit(0);
}
printf("PARENT: %d\n", syscall(SYS_getpid));
for(i=10; i<20; i++){
  gettimeofday (&tv, NULL);
  printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
}
```

When the replayer switches to record mode, the kernel modules also have to stop reading from the buffers and start writing. The problem is that an inconsistency appears in the logs regarding the identifiers. Additional data structure can be used to keep the new identity but it is not a scalable solution. In order to bring the pids to the same values, during the replay phase the thread will overwrite the read data, as it is illustrated in Figure 7. Also at the moment when the transition is made from replay to record mode the parent structure has to be modified. Running the scenario described above the results are as shown in Figure 8.

D. Write

Several system calls require to be executed at replay too. Such is write, which results were lost by a system crash. Since it is not traced it might seem unnecessary to create an entry for it in the buffer. Consider the following case: a thread A calls gettime and writes the result on the screen, after which thread B continues. The code is as shown below:

```
pid=vfork();
if (pid==0){
    ...
    gettimeofday (&tv, NULL);
```

```
root@ioana-desktop:/home/ioana/Desktop/New# ./recorder simpleFork_sleep
saving the buffer...
PID = 5830 time = 334968
PID = 5830 time = 335157
PID = 5831 time = 335351
PID = 5831 time = 335507
PID = 5830 time = 335418
PID = 5831 time = 335724
saving the buffer...
PID = 5830 time = 335690
PID = 5831 time = 335945
^C
root@ioana-desktop:/home/ioana/Desktop/New# ./replayer simpleFork_sleep
PID = 5830 time = 334968
PID = 5830 time = 335157
PID = 5831 time = 335351
PID = 5831 time = 335507
PID = 5830 time = 335418
PID = 5831 time = 335724
PID = 5830 time = 335690
PID = 5831 time = 335945
PID = 5834 time = 188192
PID = 5835 time = 200057
PID = 5834 time = 188475
PID = 5835 time = 200334
saving the buffer...
saving the buffer...
^C
root@ioana-desktop:/home/ioana/Desktop/New# ./replayer simpleFork_sleep
PID = 5830 time = 334968
PID = 5830 time = 335157
PID = 5831 time = 335351
PID = 5831 time = 335507
PID = 5830 time = 335418
PID = 5831 time = 335724
PID = 5830 time = 335690
PID = 5831 time = 335945
PID = 5834 time = 188192
PID = 5835 time = 200057
PID = 5834 time = 188475
```

Fig. 8. Second Time Replay Example

Fig. 9. Wrong Scenario for Write

```
    printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
    ...
}
else{
  ...
  gettimeofday (&tv, NULL);
  printf("i=%d time=%ld \n", i,  (long) tv.tv_usec);
  ...
}
```

During replay thread A calls gettime but B starts running earlier. It is allowed to do this because on the write system call there is no identification data. When A resumes execution write will show the correct value but the processes order was disturbed and the results shown are different. This scenario is illustrated in Figure 9. Consequently, any system calls, whether it requires tracing or not, has to write entries in the logs.

An example highlighting the issues discussed is shown in figure 10.

E. Measuring the Performance

Several tests were performed on programs implementing the same scenario with an increasing number of threads and their

```
root@ioana-desktop:/home/ioana/Desktop/New# ./recorder multiFork
saving the buffer...
PID = 6342 time = 854820
PID = 6342 time = 855265
PID = 6344 time = 855679
PID = 6343 time = 862777
PID = 6345 time = 863365
PID = 6346 time = 864424
PID = 6347 time = 867009
PID = 6348 time = 867566
PID = 6350 time = 874914
PID = 6349 time = 875710
PID = 6352 time = 878324
PID = 6351 time = 878754
saving the buffer...
^C
root@ioana-desktop:/home/ioana/Desktop/New# ./replayer multiFork
PID = 6342 time = 854820
PID = 6342 time = 855265
PID = 6344 time = 855679
PID = 6343 time = 862777
PID = 6345 time = 863365
PID = 6346 time = 864424
PID = 6347 time = 867009
PID = 6348 time = 867566
PID = 6350 time = 874914
PID = 6349 time = 875710
PID = 6352 time = 878324
PID = 6351 time = 878754
saving the buffer...
^C
```

Fig. 10. Write example

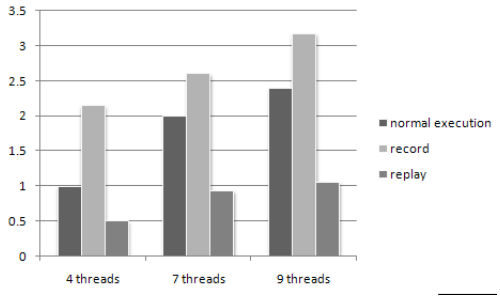

Fig. 11. Execution time

performance was plotted in Figure 11. The execution time is shown for the normal execution, for the record phase and for the replay phase. From the plot several observations can be deducted. The execution time is proportional to the number of threads, since each thread comes with an implementation overhead. Recording the application implies an increase in execution time, while replaying takes considerable less then the normal execution, both behaviors being the ones expected.

V. Conclusion

In this paper, it was proposed a system for assuring fault tolerance to Linux programs based on the record and replay model. While mostly used today for debugging, the technique has the features necessary to be implemented into a fault tolerance tool. It was shown that it offers good support for multithreading applications, while having acceptable performance, and being transparent to the applications and application independent.

As a future development, the current tool can be integrated into another fault tolerant tool based on checkpoint and restart. Using both techniques would provide a better performance and a more secure system. The tool on its own has some limitations. It does not control behavior of thread when signals are executed. This limitation is nevertheless a question of time, since the utrace framework is capable of offering support in this direction. Additional work is necessary in order to cover a larger range of application, but again this is just a time constraint.

The discussed tool therefore accomplishes its goal which was to present, from an academic point of view, a feasible alternative to a fault tolerant system based on the record and replay technique.

Acknowledgment

Special acknowledgement goes to Dipl. Eng. Adrian Colesa, my supervisor, who provided guidance during the development of the entire project.

References

[1] Michael Treaster, “A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems”, Jan 2005

[2] Satish Narayanasamy, Gilles Pokam, and Brad Calder, “Bugnet: Continuously recording program execution for deterministic replay debugging.” in *SIGARCH Comput. Archit. News*, 2005, 33(2):284-295

[3] Yasushi Saito, *“Jockey: A Userspace Library for Record replay Debugging”*, 2005, Technical Report, HP Labs

[4] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou “Flashback: A lightweight extension for rollback and determinsitic replay for software debugging” in *USENIX Annual Tech. Conf.*, Boston, MA, USA, June 2004.

[5] Ekaterina Itskova, *“Echo: A deterministic record/replay framework for debugging multithreaded applications”*. Raport proiect individual, Colegiul Imperial, Londra, 2006.

[6] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, Vara Prasad, “Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps” *Proceedings of the Linux Symposium*, 2007.

[7] Utrace Documentation last accessed in May 2010. [Online]. Available: http://people.redhat.com/roland/utrace/

[8] Lucian Cocan, “R&R Infrastructura pentru inregistrarea si reexecutia determinista a aplicatiilor utilizator”, Diploma Project, 2009

[9] David Johnson, Willy Zwaenepoel “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing” in *Journal of Algorithms*, Volume 11, Issue 3, September 1990, pg. 462-491

[10] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, *Linux device driver 3rd edition*, chapter 6, O’Reilly, 2005.

[11] Pradeep Padala, “Playing with ptrace” in *Linux Journal*, Volume 2002, Issue 103, ISSN:1075-3583, pg. 5, 2002

[12] H Zhong, J Nieh, *Crack: Linux checkpoint/restart as a kernel module*. Department of Computer Science, Columbia University 2001

[13] J Duell, PH Hargrove, ES Roman *Requirements for Linux checkpoint/restart*. Lawrence Berkeley National Laboratory, 2002

[14] Robert Love, *Linux System Programming. Talking direct to the Kernel and C library*, O’Reilly, chapter 5, pg. 135, 2007.