Testing Parallel Programs Using Controlled Executions

Michiel Ronsse Mark Christiaens Koen De Bosschere
ELIS Department
Ghent University
Belgium

Abstract *This paper presents a tool that enables programmers to use dynamic testing tools for debugging non-deterministic parallel programs. The solution uses a record/replay mechanism that allows replaying a recorded or an artificial execution. As the replayed execution is guaranteed to be equivalent to the constructed or original execution, highly intrusive testing tools can be used.*

Keywords: cyclic debugging, non-determinism

1 Introduction

Although a number of advanced programming environments, formal methods and design methodologies for developing reliable software are emerging, one notices that the biggest part of the development time is spent while debugging and testing applications.

Normally, a program is debugged or tested using a *program execution*. Both a dynamic analysis tool as a standard debugger focus on one execution only. Indeed, repeating the same program execution over and over will eventually reveal the cause of an error (cyclic debugging). Most programmers still stick to arcane debugging techniques such as adding print instructions or watchpoints or using breakpoints. Using this method, one tries to gather more and more detailed and specific information about the cause of the bug. It is therefore of paramount importance that the same execution is repeated over and over.

Repeating a particular execution of a deterministic program (e.g. a sequential program) is not that difficult. As soon as one can reproduce the program input, the program execution is known (input and program code define the program execution completely). This turns out to be considerably more complicated for non-deterministic programs. The program execution of such a program can not be determined a-priori using the program code and the input only, as these programs make a number of non-deterministic choices during their execution, such as the order in which they enter critical sections, the use of signals, random generators, etc. All modern thread based applications are inherently non-deterministic because the relative execution speed of the different threads is not stipulated by the program code. Cyclic debugging can not be used as such for these non-deterministic programs as one cannot guarantee that the same execution will be observed during repeated executions. Moreover, the use of a debugger will have a negative impact on the non-deterministic nature of the program.

A second problem that occurs while using dynamic analysis tools, is that such a tool can be very intrusive. e.g. a dynamic data race detector easily slows an execution down by as much of 10 times. It is possible that the overhead introduced steers the execution away from a normal path. Ideally, one would like to apply intrusive dynamic analysis techniques to an execution without disturbing the execution.

Of course, the debugging techniques and tools mentioned above test one specific execution only. However, the ultimate target of a debugger is declaring a *program* free of bugs of a particular type, and not a (number) of particular *executions*. Therefore, a programmer

will typically use a number of representative test inputs for testing a number of *different* executions. Coverage analysis tools can help measuring the portions of the code that have been tested. For a sequential program, one tries to force the application to enter all possible states, where each state is reached from another state and the state transition is guided by the program code and the external input the program receives during the particular execution. For parallel programs, the number of possible states that can be reached is much higher, as the state also depends on the order in which the parallel thread execute their instructions. This means that (1) we will need more test inputs as the number of reachable states is much higher, and (2) that refeeding the input alone is no guarantee that the same states will be reached during the execution. Problem (1) can only be solved by using more test inputs while (2) can be solved by extending the 'input' with more information about the execution.

A fourth problem also stems from the huge amount of possible states a parallel execution can be in. It is possible that we want to test an interesting state, but that we fail to generate an execution that visits that particular state.

2 The RecPlay Solution

In this paper, we present our tool, RecPlay, that deals with all problems mentioned above. RecPlay is an execution replay mechanism: it can record information about a nondeterministic parallel execution in a trace file, and it uses the trace file during a replayed execution, making sure that the same execution is obtained. Using a graphical tool, it is possible to inspect and change the trace file of a recorded execution, making it possible to 'replay' artificial executions.

RecPlay effectively solves the problems described above as

1. all replayed executions will be equivalent, making cyclic debugging possible;

2. it is possible to record an execution with minimal intrusion and to apply intrusive dynamic analysis tests during a replayed execution that behaves as a normal execution;

3. the trace file (together with the original input) can be used for testing new versions of a program, e.g. for regression testing;

4. an artificial trace file can be constructed for steering an application through interesting states.

RecPlay is an extension of the tool described in [1]. In the next section, the execution replay method used by RecPlay is described. Section 4 describes the visualization tool that was added. The tool can be used for constructing artificial traces and the next section gives an evaluation. The paper ends with a description of related work and the conclusions.

3 Execution Replay Using RecPlay

In order to obtain a number of equivalent executions, information about the nondeterministic choices should be stored in a trace file. This means that, for parallel programs[1] the outcome of all race conditions should be traced. Such a race conditions occurs if two threads change the same shared variable in an unsynchronized way. As the overhead introduced by tracing all race conditions is far too high (it forces us to intercept all memory accesses), RecPlay uses an approach based on the fact that there are two types of race conditions: synchronization races and data races. *Synchronization races* (introduced by synchronization operations) intentionally introduce non-determinism in a program execution to allow for competition between threads to enter a critical section, to

[1] In the current state, RecPlay only deals with nondeterminism due to shared memory accesses, we suppose e.g. that input is refed during a replayed execution.

lock a semaphore or to implement load balancing. *Data races* on the other hand are not intended by the programmer, and are most of the time the result of improper synchronization. By adding synchronization, data races can (and should) always be removed.

RecPlay starts from the (erroneous) premise that a program (execution) does not contain data races. If one wants to debug such a program, it is sufficient to log the order of the synchronization operations, and to impose the same ordering during a replayed execution. RecPlay uses the ROLT method [2], an ordering-based record/replay method, for logging the order of the synchronizations operations. ROLT logs, using Lamport clocks [3], the partial order of synchronization operations. A timestamp is attached to each synchronization operation, taking the so-called clock condition into consideration: if operation a causally occurs before b in a given execution the timestamp $LC(a)$ of a should be smaller than the timestamp $LC(b)$ of b. Basically, ROLT logs information that can be used to recalculate, during replay, the timestamps that occurred during the recorded execution. It turns out that the vast majority of clock increments is equal to 1, and therefore ROLT only logs the old and the new value of the clock if the increment is bigger than 1.

The ROLT method has the advantage that it produces small trace files and that it is less intrusive than other existing methods [4]. This is of paramount importance as an overhead that is too big will alter the execution, giving rise to Heisenbugs (bugs that disappear or alter their behavior when one attempts to isolate or probe it).

The information in the trace files is used during replay for attaching the Lamport timestamps to the synchronization operations. To get a faithful replay, it is sufficient to stall each synchronization operation until all synchronization operations with a smaller timestamp have been executed.

Of course, the premise that a program (execution) does not contain data races is not correct, esp. for the faulty programs we want to debug. Unfortunately, declaring a program free of data races is an unsolvable problem, at least for all but the simplest programs. RecPlay therefore will check for data races during the first replayed execution. The data race detection uses vector clocks [5] for detecting concurrent operations and introduces a huge overhead. Fortunately, the overhead is no problem as the test is done during a replayed execution and the test has to be done only once for each recorded execution. It the execution is found to be free of data races, normal replayed executions can be used, and if the execution contains a data race, cyclic debugging (replay up to the first data race will still be correct!) can be used for removing the cause of the race. More information about the used data race detector can be found in [1].

As RecPlay traces the order of the synchronization operations, it is possible to use the trace file for a different version of the parallel program, as long as the synchronization is not changed. Regression testing is therefore possible by feeding the input and the trace file to a new version of the program, e.g. a version that uses an optimized version of a mathematical library.

4 Changing an Existing Trace File

The binary trace file contains all information necessary for obtaining a correct replayed execution. Therefore, by changing the trace file one can construct an artificial execution that can be used for replaying. Unfortunately, the trace file contains too little information to be interpreted by human beings. Therefore, RecPlay can construct (during a replayed execution in an intrusive way) an extended trace file containing, for each synchronization operations, the Lamport timestamp, the address of the synchronization variable, the source code of the synchronization operation and the call stack. This trace file can be visualized with a Tcl/Tk program (see Figure 1). The left mouse button can be used to inspect an oper-

ation, while the middle button can be used for dragging the operation along the vertical line representing the thread that executed the operation. The tool protects the user against illegal trace files, e.g. by refusing an operation to be dragged before the preceding or after the following operation *on the same thread* (this is of course precluded by the program code). Moreover, the tool will warn (using colored symbols) for impossible trace files, e.g. by warning if a mutex is taken before it was released by another thread (see Figure 2). After the user has changed the visual representation of the execution, a new trace file can be written and used.

It is of course possible that the new artificial trace file does not correspond to a valid execution because the semantics of the program does not allow two operations to be executed in a different order. In this case, the replayed execution will deadlock.

5 Evaluation

The RecPlay system has been implemented for Sun multiprocessors running Solaris using the JiTI instrumentation tool we also developed [6]. The implementation uses the dynamic linking and loading facilities present in all modern Unix operating system and instruments (for intercepting the memory accesses and the synchronization operations) on the fly: the running process is instrumented.

While developing RecPlay, special attention was given to the probe effect during the record phase. Indeed, the overhead introduced is one of the most important properties of a replay mechanism as it gives an idea about the level of equivalence between the recorded and a normal execution. Table 1 gives an idea of the overhead caused during the record phase for programs from the SPLASH-2 benchmark suite[2]. The average overhead during the record phase is limited to 2.1% which is small enough to keep it switched on all the time. The size of the trace file is also very small. The table also shows the number of synchronization operations executed. There is almost no correlation between the number of synchronization operations and the overhead. This is caused by the fact that the extra work that has to performed for each synchronization operation (a simple addition and comparison of scalar values) is much smaller than the synchronization operation itself. The overhead is solely caused by writing the trace file to disk, and the size of the trace file depends on the number of synchronization operations with a timestamp increment larger than 1.

[2]All experiments were done on a machine with 4 processors and all benchmarks were run with 4 threads.

6 Related Work

In the past, other replay mechanisms have been proposed for shared memory computers. Instant Replay [7] is targeted at coarse grained operations and traces all these operations. It does not use any technique to reduce the size of the trace files nor to limit the perturbation introduced. It does not work for programs containing data races. A prototype implementation for the BBN Butterfly is described.

Netzer [4] introduced an optimization technique based on vector clocks. As the order of all memory accesses is traced, both synchronization and data races will be replayed. It uses comparable techniques as ROLT to reduce the size of the trace files. However, no implementation was ever proposed (of course, the overhead would be huge as all memory accesses are traced, introducing Heisenbugs). We believe that it is far more interesting to detect data races than to record/replay them. Therefore, RecPlay replays the synchronization operations only, while detecting the data races.

Tools for changing trace files are less frequent. ATEMPT [8], *A Tool for Event Manipulation*, is a tool for inspecting and changing the trace file of *message passing* programs. The idea is to test for racing messages by checking whether a changed trace file (messages arriving at the same process can be switched) still leads to the same execution.

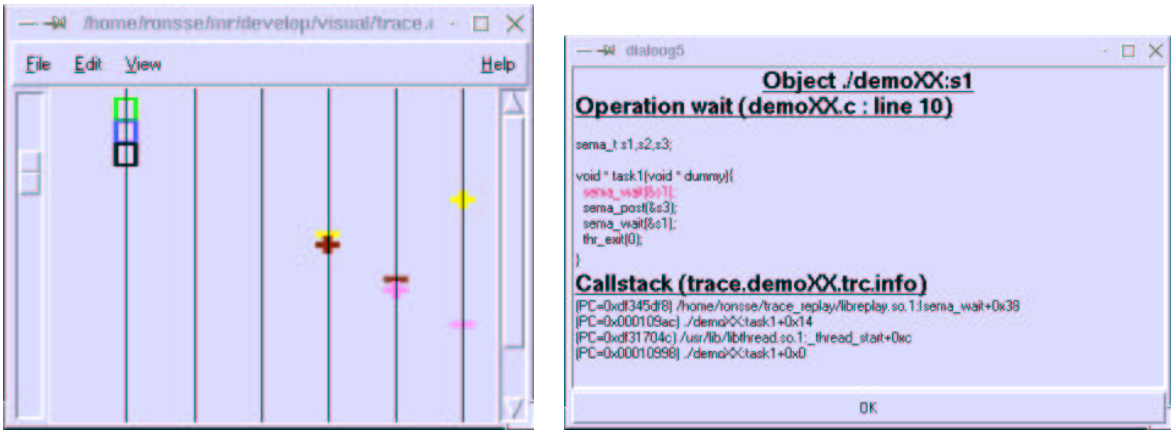

Figure 1: The left figure shows a visualized trace file; each synchronization variable has a unique color and each type of synchronization operation has a unique symbol. The right figure show information about a specific synchronization operation.

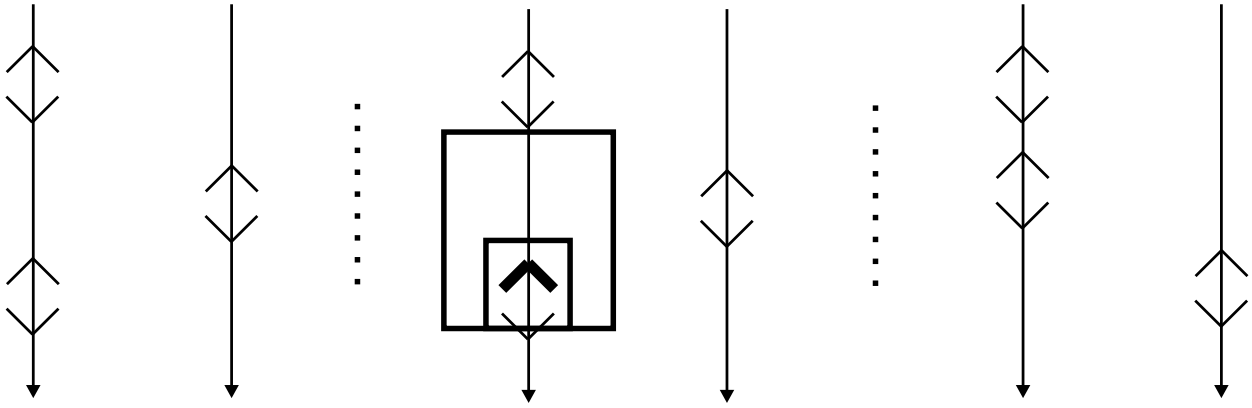

Figure 2: The leftmost part of the figure shows two threads using the same mutex (lock and unlock operations are denoted by the up and down pointing symbols). Suppose we want to change the order in which the two threads accesses the mutex, as shown in the rightmost part of the figure. This can be accomplished by dragging the symbols using the tool. As soon as a symbol (suppose the locking of the mutex by the first thread (shown by the thicker symbol) in the middle figure) is dragged, the tool defines two areas, an area that cannot be left by the symbol (as shown by the largest rectangular region) and a safe area (the smaller rectangular region). The large area can not be left by the mutex operation as this would mean it was executed before or after the surrounding operations by the *same* thread. Leaving the safe are means that (temporarily) a deadlock is created (the mutex is grabbed while it is still locked by the second thread). Note that it is impossible to change the left execution in the right execution without creating (temporarily) an execution trace that would lead to deadlock.

program	normal	record		size of trace	number of
	runtime	runtime	slowdown	file (b)	sync. op.
cholesky	8.67	8.88	1.024	1 132	13857
fft	8.76	8.83	1.008	65	177
LU	6.36	6.40	1.006	134	127
radix	6.03	6.20	1.028	108	273
ocean	4.96	5.06	1.020	6 458	22981
raytrace	9.89	10.19	1.030	41 416	150960
water-Nsq.	9.46	9.71	1.026	336	631
water-spat.	8.12	8.33	1.026	332	625

Table 1: Basic performance of RecPlay (all times in seconds).

7 Conclusions

In this paper we have presented RecPlay, a practical and effective support tool for dynamic analysis of parallel programs. The tool allows intrusive debugging techniques and tools to be used during a replayed execution that behaves as a normal execution. Moreover, a visualization tool can be used for constructing artificial trace files.

References

[1] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[2] Luk J. Levrouw, Koenraad M. Audenaert, and Jan M. Van Campenhout. A new trace and replay system for shared memory programs based on Lamport Clocks. In *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, pages 471–478. IEEE Computer Society Press, January 1994.

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

[4] Robert H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, May 1993.

[5] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Roberts, editors, *Proceedings of the Intl. Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V., North-Holland, 1989.

[6] M. Ronsse and K. De Bosschere. Jiti: A robust just in time instrumentation technique. In *Proceedings of WBT-2000 (Workshop on Binary Translation)*, Philadelphia, 10 2000.

[7] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[8] D. Kranzlmller, S. Grabner, and J. Volkert. Debugging massively parallel programs with atempt. In *Proceedings of HPCN-Europe'96*, pages 806–811, Brussels, Belgium, 4 1996.