Cyclic Debugging Using Execution Replay

Michiel Ronsse, Mark Christiaens, and Koen De Bosschere

ELIS Department, Ghent University, Belgium
{ronsse,mchristi,kdb}@elis.rug.ac.be

Abstract. This paper presents a tool that enables programmers to use cyclic debugging techniques for debugging non-deterministic parallel programs. The solution consists of a combination of record/replay with automatic on-the-fly data race detection. This combination enables us to limit the record phase to the more efficient recording of the synchronization operations, and checking for data races during a replayed execution. As the record phase is highly efficient, there is no need to switch it off, hereby eliminating the possibility of Heisenbugs because tracing can be left on all the time.

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. Moreover, 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. One usually starts with a hypothesis about the bug that one wants to prove or deny.

Normally, a program is debugged using a program execution. Indeed, repeating the same program execution over and over will eventually reveal the cause of the error (cyclic debugging). 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. As a debugger can manipulate the execution of the different threads of

© Springer-Verlag Berlin Heidelberg 2001

the application, it is possible that a significant discrepancy in execution speed arises, giving cause to appearing or disappearing. The existence of this kind of errors, combined with the primitive debugging tools used nowadays, makes debugging parallel programs a laborious task.

In this paper, we present our tool, RecPlay, that deals with the non-determinism introduced by one cause of non-determinism that is specific for parallel programs: unsynchronized accesses to shared memory (the so-called *race conditions*[1]). RecPlay uses a combination of techniques in order to allow the usage of standard debuggers for sequential programs for debugging parallel programs. RecPlay is a so-called execution replay mechanism: information about a program execution can be traced (*record phase*) and this information is used to guide a faithful re-execution (*replay phase*). A faithful replay can only be guaranteed if and only if the log contains sufficient information about all non-deterministic choices that were made during the original execution (minimally the outcome of all the race conditions). This suffices to create an identical re-execution, the race conditions included. Unfortunately, this approach causes a huge overhead, severely slows down the execution, and produces huge trace files. An alternative approach we advocate in this paper is to record an execution as if it did not contain data races, and to check for the occurrence of data races during a replayed execution. As has been shown [CM91], replay will be guaranteed to be correct up to the race frontier, i.e., the point in the execution of each thread were a race event is about to take place.

2 The RecPlay Method

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 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.[2] RecPlay uses the ROLT method [LAV94], an ordering-based record/replay method, for logging the order of the synchronizations operations. ROLT logs, using Lamport clocks [Lam78], the partial order of synchronization operations. A timestamp is attached to each synchronization

[1] Technically, a race condition occurs whenever two threads access the same shared variable in an unsynchronized way, and at least one thread modifies the variable.

[2] Remember that RecPlay only deals with non-determinism due to shared memory accesses, we suppose e.g. that input is refed during a replayed execution.

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.

The ROLT method has the advantage that it produces small trace files and that it is less intrusive than other existing methods [Net93]. 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 attemps to isolate or probe it, [Gai86]. Moreover, the method allows for the use of a simple compression scheme [RLB95] which can further reduce the trace files. 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. Unfortunately, declaring a program free of data races is an unsolvable problem, at least for all but the simplest programs [LKN93]. Even testing one particular execution for data races is not easy: we have to detect whether the order in which two memory accesses occur during a particular execution is fixed by the program code or not. Unfortunately, this is only possible if the synchronization operations used reflect the synchronization order dictated by the program code. E.g. this is possible if the program only uses semaphores and the program contains no more than one $P()$ and one $V()$ operation for each semaphore. If this is not the case, it is impossible to decide whether the order observed was forced by the program code or not. However, for guaranteeing a correct replay, we do not need this information as we want to detect if *this* replayed execution contains a data race or not, as a data race would render the replay unreliable. And as *we are imposing a particular execution order* on the synchronization operations using the trace file, we know that the synchronization operations are forced in this order. However, this order is forced by RecPlay, and not by the program itself.[3]

The online data race detection used by RecPlay consists of three phases:

1. *collecting memory reference information* for each sequential block between two successive synchronization operations on the same thread (called segments). This yields two sets of memory references per segment: $S(i)$ are the locations that were written and $L(i)$ are the locations that were read in segment i. RecPlay uses multilevel (see Figure 1) bitmaps for registering the memory accesses. Note that multiple accesses to the same variable in a segment will be counted as one, but this is no problem for detecting data races. The sets $L(i)$ and $S(i)$ are collected on a list.

[3] In fact, it is not necessary to re-execute the synchronization operations from the program, as RecPlay forces an execution order (a total order) on the synchronization operations that is stricter than the one contained in the program (a partial order).

2. *detecting conflicting memory references* in concurrent segments. There will be a data race between segment i and segment j if either $(L(i) \cup S(i)) \cap S(j) \neq \emptyset$ or $(L(j) \cup S(j)) \cap S(i) \neq \emptyset$ is true. If the comparison indicates the existence of a data race, RecPlay saves information about the data race (address and threads involved, and type of operations (load or store)). For each synchronization operation, RecPlay will compare the bitmaps of the segment that just ended against the bitmaps of the parallel segments on the list. Moreover, RecPlay will try to remove obsolete segments from the list. A segment becomes obsolete if it is no longer possible for future segments to be parallel with the given segment.
3. *identifying the conflicting memory accesses* given the traced information. This requires another replayed execution.

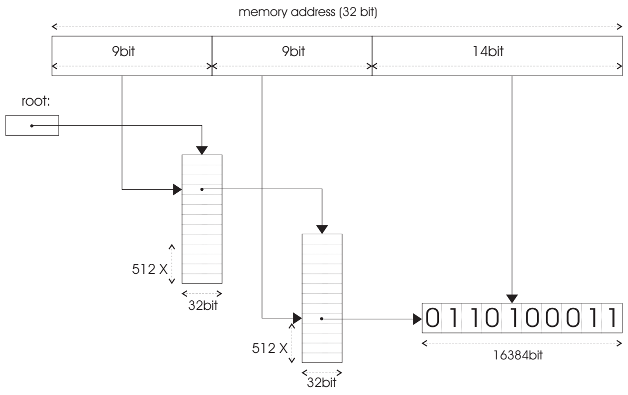

Fig. 1. RecPlay uses a 3-level bitmap where each level is addressed using a different part of the address: the first two parts are used to address lists of pointers, while the last part of the address points to the actual bit. Such a bitmap favors programs with a substantial memory locality.

In our race detection tool, we use a classical logical vector clock [Mat89, Fid91] for detecting concurrent segments as segments x and y can be executed in *parallel* if and only if their vector clocks are not ordered (p_x is the thread on which segment x was executed):

$$x \| y \Leftrightarrow \begin{cases} (VC_x[p_x] \geq VC_y[p_x]) \text{ and } (VC_x[p_y] \leq VC_y[p_y]) \\ \text{or} \\ (VC_x[p_x] \leq VC_y[p_x]) \text{ and } (VC_x[p_y] \geq VC_y[p_y]) \end{cases}$$

This is possible thanks to the strong consistency property of vector clocks. For detecting and removing the obsolete segments, RecPlay uses an even stronger clock: snooped matrix clocks [DBR97].

It is clear that data race detection is not a cheap operation. The fact that all memory accesses must be intercepted does indeed impose a huge overhead.

Fortunately, RecPlay performs the data race detection during a replayed execution, making it impossible for the data race detector to alter the normal execution. Moreover, for each recorded execution, only one data race check is necessary. If no data races are found, it is possible to replay the execution without checking for data races. This will lead to a much faster re-execution that can be used for cyclic debugging.

3 Evaluation

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

program	normal	record		replay		replay+detect	
	runtime	runtime	slow-down	runtime	slow-down	runtime	slow-down
cholesky	8.67	8.88	1.024	18.90	2.18	721.4	83.2
fft	8.76	8.83	1.008	9.61	1.10	72.8	8.3
LU	6.36	6.40	1.006	8.48	1.33	144.5	22.7
radix	6.03	6.20	1.028	13.37	2.22	182.8	30.3
ocean	4.96	5.06	1.020	11.75	2.37	107.7	21.7
raytrace	9.89	10.19	1.030	41.54	4.20	675.9	68.3
water-Nsq.	9.46	9.71	1.026	11.94	1.26	321.5	34.0
water-spat.	8.12	8.33	1.026	9.52	1.17	258.8	31.9

The RecPlay system has been implemented for Sun multiprocessors running Solaris using the JiTI instrumentation tool we also developed [RDB00]. 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. Table 1 gives an idea of the overhead caused during record, replay, and race detection for programs from the SPLASH-2 benchmark suite [4]. 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 average overhead for replay is 91% which can seem high, but is feasible during debugging. The automatic race detection is however very slow: it slows down the program execution about 40 times (the overhead is mainly caused by JiTI intercepting all memory accesses). Fortunately, it can run unsupervised, so it can run overnight and we have to run it only once for each execution.

The memory consumption is far more important during the data race detection. The usage of vector clocks for detecting the races is not new, but the

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

Table 2. Number of segments created and compared during the execution, and the maximum number of segments on the list.

program	created	max. stored	compared
cholesky	13 983	1 915 (13.7%)	968 154
fft	181	37 (20.5%)	2 347
LU	1 285	42 (3.3%)	18 891
radix	303	36 (11.9%)	4 601
ocean	14 150	47 (0.3%)	272 037
raytrace	97 598	62 (0.1%)	337 743
water-Nsq.	637	48 (7.5%)	7 717
water-spat.	639	45 (7.0%)	7 962

mechanism used for limiting the memory consumption is. The usage of multi-level bitmaps and the removal of obsolete segments (and their bitmaps) allows us to limit the memory consumption considerably. Table 2 shows the number of segments that was created during the execution, the maximum number on the list, and the number of parallel segments during a particular execution (this is equal to the number of segments compared). The average maximum number of segments on the list is 8.0%, which is a small number. Without removing obsolete segments, this number would of course be 100%. Figures 2 and 3 show the number of segments on the list and the total size of the bitmaps in function of the time (actually the number of synchronization operations executed so far) for two typical cases: `lu` and `cholesky`[5]. For `lu`, the number of segments is fairly constant, apart from the start and the end of the execution. The size of the bitmaps is however not that constant; this is caused by the locality of the memory accesses as can be seen in the third graph showing the number of bytes used by the bitmaps divided by the number of segments. The numbers for `cholesky` are not constant, but the correlation between the number of segments and the size of the bitmaps is much higher, apart from a number of peaks. The number of segments drops very quickly at some points, caused by *barrier* synchronization creating a large number of obsolete segments.

4 Related Work

In the past, other replay mechanisms have been proposed for shared memory computers. Instant Replay [LM87] 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 [Net93] introduced an optimization technique based on vector clocks. As the order of all memory accesses is traced, both synchronization and data

[5] These are not the runs used for Table 2

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.

Race Frontier [CM91] describes a similar technique as the one proposed in this paper (replaying up to the first data race). Choi and Min prove that it is possible to replay up to the first data race, and they describe how one can replay up to the race frontier. A problem they do not solve is how to efficiently find the race frontier. RecPlay effectively solves the problem of finding the race frontier, but goes beyond this. It also finds the data race event.

Most of the previous work, and also our RecPlay tool, is based on Lamport's so-called *happens-before* relation. This relation is a partial order on all synchronization events in a particular parallel *execution.* If two threads access the same variable using operations that are not ordered by the happens-before relation and one of them modifies the variable, a data race occurs. Therefore, by checking the ordering of all events and monitoring all memory accesses data races can be detected for one *particular program execution.* Another approach is taken by a more recent race detector: Eraser [SBN$^+$97]. It goes slightly beyond work based on the happens-before relation. Eraser checks that a *locking discipline* is used to access shared variables: for each variable it keeps a list of locks that were hold while accessing the variable. Each time a variable is accessed, the list attached to the variable is intersected with the list of locks currently held and the intersection is attached to the variable. If this list becomes empty, the locking discipline is violated, meaning that a data race occurred.

The most important problem with Eraser is however that its practical applicability is limited in that it can only process mutex synchronization operations and in that the tool fails when other synchronization primitives are build on top of these lock operations.

5 Conclusions

In this paper we have presented RecPlay, a practical and effective tool for debugging parallel programs with classical debuggers. Therefore, we implemented a highly efficient two-level record/replay system that traces the synchronization operations, and uses this trace to replay the execution. During replay, a race detection algorithm is run to notify the programmer when a race occurs. After removing the data races, normal sequential debugging tools can be used on the parallel program using replayed executions.

References

[CM91] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proc. of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 145–154, July 1991.

[DBR97] Koen De Bosschere and Michiel Ronsse. Clock snooping and its application in on-the-fly data race detection. In *Proceedings of the 1997 International Symposium on Parallel Algorithms and Networks (I-SPAN'97)*, pages 324–330, Taipei, December 1997. IEEE Computer Society.

[Fid91] C. J. Fidge. Logical time in distributed computing systems. In *IEEE Computer*, volume 24, pages 28–33. August 1991.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LAV94] 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.

[LKN93] Hsueh-I Lu, Philip N. Klein, and Robert H. B. Netzer. Detecting race conditions in parallel programs that use one semaphore. Workshop on Algorithms and Data Structures (WADS), Montreal, August 1993.

[LM87] 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.

[Mat89] 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.

[Net93] 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.

[RDB00] 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.

[RLB95] M. Ronsse, L. Levrouw, and K. Bastiaens. Efficient coding of execution-traces of parallel programs. In J. P. Veen, editor, *Proceedings of the ProRISC / IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, pages 251–258. STW, Utrecht, March 1995.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

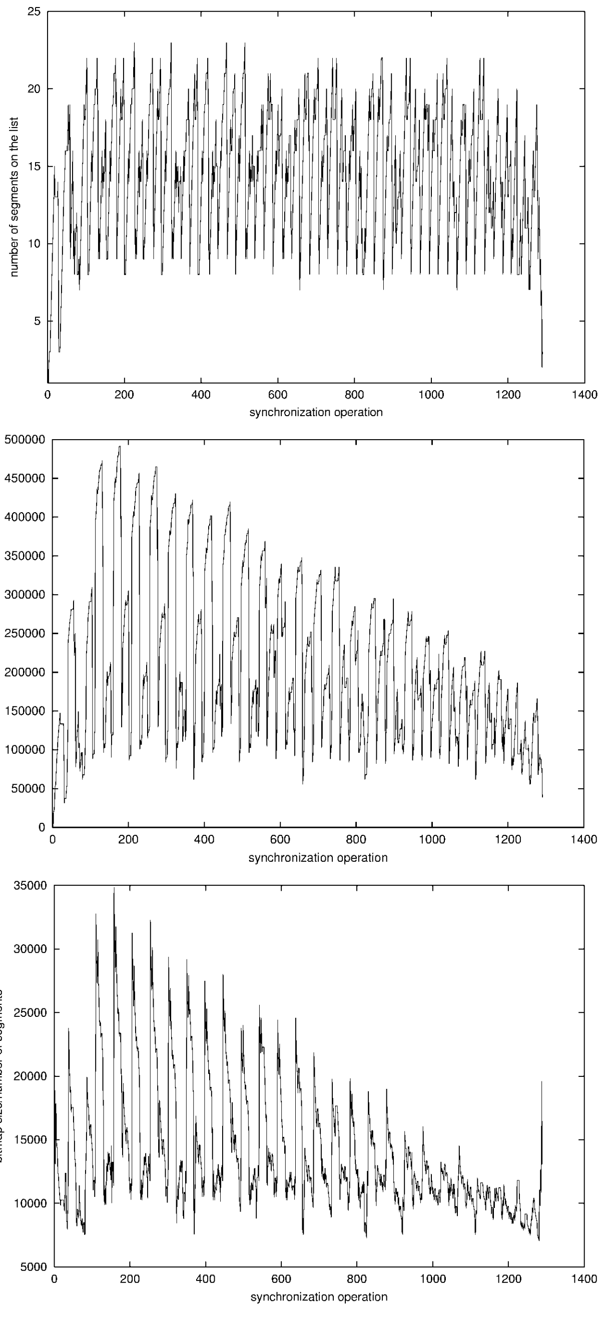

Fig. 2. Number of segments, size of the bitmaps and number of bytes per segment for `lu`.

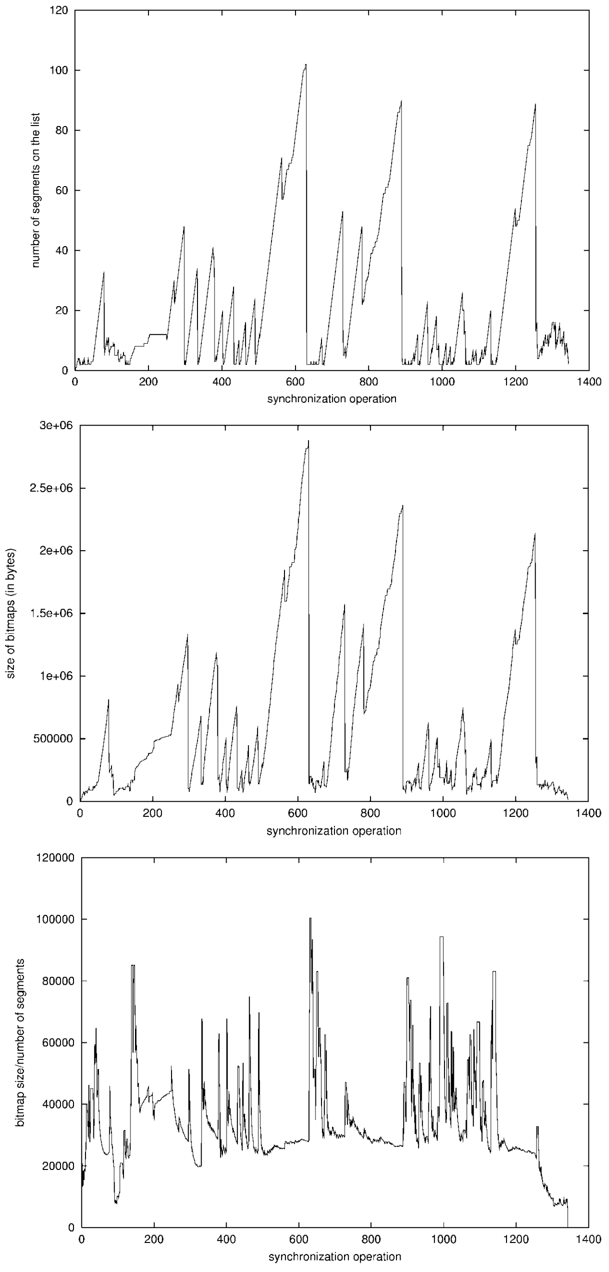

Fig. 3. Number of segments, size of the bitmaps and number of bytes per segment for `cholesky`.