On the Implementation of a Replay Mechanism

Michiel Ronsse* and Luk Levrouw

Department of Electronics and Information Systems
Universiteit Gent, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Abstract. Parallel programs can be nondeterministic: consecutive runs with the same input can result in different executions. Therefore we cannot use cyclic debugging techniques. In order to be able to use those techniques we need a tool that traces information about an execution so it can be replayed for debugging. Because the recording interferes with the program, possibly perturbating the execution, we must limit the amount of information and keep the algorithm simple. This paper presents an implementation of the ROLT replay mechanism for a multi-threaded operating system (Solaris).

1 Introduction

The debugging of most parallel programs is a time-consuming task due to the complex nature of parallel programs. Moreover, most parallel programs are non-deterministic, limiting the use of cyclic debugging techniques. These techniques are based on the fact that re-executions of a program will result in the same program flow if we supply the same input.[2] During those re-executions we can analyze the program execution by setting breakpoints and watching variables until we find the error.

2 Replay Mechanisms

If we can force re-executions to be 'equivalent' to the execution that contains an error, we can still use cyclic debugging techniques. This can be accomplished using a replay mechanism: we trace a program execution (record phase), and use those traces to force subsequent executions (replay phase) to be 'equivalent' to the traced one. As these forced re-executions will be deterministic, we can use intrusive debugging techniques (breakpoints, collecting data, ...). To be practical, it is important that the trace mechanism produces small trace files and has a small overhead.

Recently, a new replay method called ROLT (Reconstruction Of Lamport Timestamps) was introduced [LAV94]. The mechanism produces smaller trace

* Michiel Ronsse is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT).

[2] For the remainder of this paper, we will assume that the user input, file input, system calls, ...return the same result during subsequent executions.

files and is much less intrusive than comparable mechanisms [Net93, LM87]. The implementation described in this paper traces the order of the synchronisation operations. Forcing these operations to occur in the same order during replay will yield the same execution as long as the program is data race free. A data race is an unsynchronized access to a shared variable by two processors when at least one modifies the variable. This is the caused by a lack of synchronization (or the wrong synchronization). As there is no synchronization, no information will be stored during the record phase. So, during replay, we will be unable to force the accesses to occur in the same order as in the record phase. This doesn't mean that the mechanism is totally unusable in the presence of a data race. A replay of an execution that contains a data race will yield an equivalent re-execution up to the point were the data race occured. Therefore, an (intrusive) data race detection method can be used during replay to find the race.

The ROLT mechanism uses Lamport clocks [Lam78] to attach logical timestamps to synchronization operations. As logical timestamps may not reflect the actual real-time order of the operations and because it is required that operations on the same synchronization variable have consistent timestamps (the timestamps of all operations on the same object should reflect their execution order) a simple update scheme is used at every synchronisation operation [LAV94].

During replay, the Lamport timestamps of the original execution are used to add sufficient synchronization for a correct replay: operations with a lower Lamport timestamp are executed first. To be able to do this, the same Lamport timestamps as in the original execution must be attached to the corresponding operations. Some of these timestamps can be deterministically recomputed during replay, the others were traced.

3 Implementation

Solaris offers the parallel programmer different synchronization types. These facilities are build using a layered approach. The lowest layer consists of a synchronization facility directly supported by the Sparc processors: `ldstub` (load-store unsigned byte). This is a simple *read-modify-write* instruction. The next layer (offered by `libc`) consists of two functions: `_lock_try` and `_lock_clear`. The first one tries to lock a byte and returns the result (succeeded or not), the last one clears a lock. The next layer consists of the four different synchronization types offered by the thread library (`libthread`). These are: mutual exclusion (`mutex`) locks, condition variables (`condvar`), counting semaphores (`sema`) and multiple readers, single writer locks (`rwlock`).

The replay mechanism was implemented using the dynamic linking facility of Solaris: we wrote two new `libthread` libraries, one for the record phase, and one for the replay phase. By using the dynamic linking facilities, the user doesn't have to do anything to use the replay mechanism. He doesn't have to recompile or relink his program, he can use whatever interactive debugger he likes (`gdb`, `dbx`, `debugger`, ...). Moreover, a user can add his own synchronization primitives

(e.g. barriers) and use the instrumented Solaris functions in the record library to implement them.

As mentioned before, we will generate trace information about a particular execution during the tracing. The information is stored in a buffer in memory, and written to disk when the buffer is full. We have to consider four different cases :

The program exits normally, and the result is correct. This means that this particular program execution is correct. We can discard the traces, or we can use them to force a deterministic replay and collect more information about the program execution.

The program exits normally, but the result is not correct. In that case, we can use the traces to force a replay. Using an interactive debugger we can find the error using watchpoints, breakpoints, ...

The program ends, but at a wrong exit point (it crashes). As the tracing is performed by the library, the tracing mechanism will crash together with the application. This means that the information that is still in the memory buffer is lost. To tackle this problem, we use a Unix-daemon that controls and owns the memory used by the library. It checks on a regular base if one of the programs it provides with memory has crashed, and saves the memory to disk if necessary.

The program never ends (deadlock, infinitive loop). In this case, we have to force a program crash (i.e. sending a kill signal using `^C`).

Record Phase During the record phase, we have to trace the execution order of the synchronization operations. As Solaris provides different levels of synchronization primitives, we can trace at different levels:

- we can trace at the lowest level (lock level). This will force all levels above this level to be replayed in the correct order;
- we can trace at the mutex level. All levels above this level will be replayed in the correct order, the level beneath it (lock level) won't. As it is possible that mutex functions have to call `_lock_try` several times before the lock is grabbed, this will diminish the number of operations to be traced;
- we can trace at the highest level: the synchronization operations (mutex, rwlock, condvar, sema) performed by the application are traced, the synchronization operations called by the rwlock, condvar and sema operations aren't.

Replay Phase Every thread recomputes its Lamport timestamps during replay. When a thread wants to perform a synchronization operation, the thread waits until all other threads have executed the operations with smaller Lamport clocks. This adds the extra synchronization needed to yield an equivalent execution.

4 Experimental Evaluation

Up to now, only limited experiments were performed with a parallel implementation of the Maximum Likelihood Expectation Maximization algorithm. The program was executed on a Sun with 4 processors. The following table shows the number of synchronization operations performed, the size of the trace files and the execution time.

level	#operations	logsize (b)	real time (s)
sema	2406	11824	44.98
mutex	7140	12424	45.11
lock	7791	17056	45.51

It is clear that, as expected, we have to log less, and that the overhead is smaller if we trace at a higher abstraction level. The total execution time for a normal execution was 44.23 seconds, for a traced execution 44.98 seconds and for a re-execution 48.42 seconds (mean values for 100 program runs; sema level). As we will be using an interactive debugger during the program re-executions, the increase in execution time causes no harm.

5 Conclusions

This paper showed that it is necessary to use a replay mechanism if one wishes to use cyclic debugging techniques for the debugging of non-deterministic parallel programs. An implementation of the ROLT mechanism for Solaris was proposed. It generates small logfiles and has a low overhead.

References

[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.

[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.

[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.

[RLB95] M.A. Ronsse, L.J. 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.