Presented in the poster session of the 3rd ACM EuroSys Conference, Glasgow (Scotland), April 2008

DDB: Deadlock Debugger

Cristian Zamfir
EPFL
cristian.zamfir@epfl.ch

George Candea
EPFL
george.candea@epfl.ch

1. MOTIVATION

Writing concurrent software is a challenging task prone to hard to debug errors such as deadlocks. Deadlocks are challenging to debug because they may occur rarely, based on a particular thread interleaving and are often hard or even impossible to reproduce in a debugger. Moreover, once they happen, they can bring a large software system to an unexpected halt, requiring a whole system restart.

We introduce DDB, a debugging environment capable of deterministically and efficiently replaying deadlocks in any large software system that uses *pthreads*. It does this without the need to modify the production software and incurs a minimum overhead at runtime. DDB should work with systems like Apache, MySQL, JDK and JBOSS..

DDB's core idea is to capture a lightweight trace at runtime and to shift all the complexity to the debugging process which is executed offline and can take a longer time to complete. The second idea is to avoid capturing program inputs by mutating the execution in order to reproduce the deadlock. Therefore, DDB captures only the synchronization trace of the application that can be subsequently used to replay an equivalent execution inside a debugging environment.

We believe that such a replay environment is highly desirable to the developer since most deadlocks are impossible to reproduce unless the developer has remote access to the deployed application. Even so, deadlocks might only occur in unreproduceble thread interleavings. The replay should be able to offer the debugger meaningful information that improves the time to produce a bug fix.

Furthermore, our approach is highly portable. The debugger might not be required to recreate the setup that caused a specific deadlock and, for instance, traces obtained on a FreeBSD system could be replayed on a completely different Linux system.

2. DESIGN AND IMPLEMENTATION

DDB is made up of two components: the capture component is active at runtime inside production software and captures the thread synchronization operations. The replay part runs in a debugging environment and causes the captured synchronization operations to occur in the same order they did during the capture phase.

DDB is targeted at reproducing deadlocks that occur in production software. Therefore recording the trace of synchronization operations should be done with minimum overhead. Moreover, it should be done without having to modify existing software. In order to achieve a negligible overhead, we chose to perform the recording from inside the pthreads library. Moreover, we intend to use static analysis tools to eliminate some of the tracing points inside threads that are guaranteed not to be involved in a deadlock. Recording more than the synchronization operations, such as all program inputs imposes an unacceptable overhead for large systems. Therefore, we offload all the complexity to the replay component since this procedure is done offline.

DDB's replay component finds an execution path that matches a recorded trace by looking at the control flow graph of the application. After such a path is found, DDB runs the program and dynamically mutates [1] the binary to make sure that the execution path reproduces the order of the synchronization operations in the trace. This will produce an equivalent execution that will reproduce the deadlock. However, it is likely to produce a program that has an unpredictable behavior (for instance it might access unallocated memory and cause a *SIGSEGV*). We can infer a set of program inputs that will reproduce the same trace and run the application with these inputs to replay the deadlock. Alternatively, similar to the *failure oblivious computing* approach [2] we allow the program to run and we intercept possible errors induced by our mutations in order to continue the execution and reproduce the deadlock.

3. STATUS AND FUTURE WORK

We have implemented DDB's capture component in FreeBSD using the *libthr* pthreads implementation. Initial tests show that the capture component induces a reasonable overhead of *20%* for the *sysbench* threads benchmark.

We are currently implementing the replay component by matching the control flow graph produced by the compiler against the traces provided by the capture component. This phase produces an ordered list of function calls and basic blocks that make up an equivalent execution to the one in the trace. Using pin [1], we dynamically rewrite the binary at runtime to produce the desired execution path and modify the thread scheduling in order to replay the deadlock.

DDB is currently work in progress. We aim to write a working prototype, evaluate the overhead and portability of our approach and to compare it against an approach that records all the input. We intend to evaluate on a set of microbenchmarks as well as systems such as MySQL, Apache and JDK and study the overhead imposed at runtime by the record phase as well as the ability to reproduce deadlocks.

4. REFERENCES

[1] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, 2005.

[2] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.