

Abstract

Deadlocks are challenging to debug because they may occur rarely, based on a particular thread interleaving and are often hard to reproduce in a debugger. We introduce DDB, a debugging environment targeted at replaying deadlocks in large software systems that use the *pthread*s library. It does this without the need to modify the production software and incurs a minimum overhead at runtime. DDB's core ideas are to capture a lightweight trace at runtime and to shift all the complexity to the offline debugging process. Moreover, DDB avoids capturing program inputs and attempts to mutate the execution of the program in order to reproduce the deadlock.

1 Introduction

Deadlocks

- Complex and hard to reproduce bugs.
- Can occur very rarely in particular thread interleavings.
- Recovery from a deadlock usually implies a whole system restart.

Thread 1	Thread 2
Lock(X)	
	Lock(Y)
	Lock(X)
Lock(Y)	

Deadlock Scenario

The deadlock only occurs in this particular thread interleaving.

Previous Work

- Static analysis yields too many false positives.
- Existing replay tools are based on recording and replaying the program inputs.
- High overhead for replaying long running programs.

Goals

- Target: server software (databases, web servers, application servers), GUIs.
- Do not modify production software.
- Small overhead at runtime.
- Deterministic replay of observed deadlocks.
- Don't need to reproduce the setup where the deadlock bug occurred.
- Cross-Platform Debugging: Optionally could try to debug a deadlock in a different OS if there is a port for both OSs.
- Should not be able to infer sensitive information from the trace.

Challenges

- Tracing is intrusive.
- Deadlocks may be triggered a long time after the program was started, therefore the trace should be small.
- Many code paths are dependent on program inputs.
- The replay should not only reproduce the deadlock but also provide meaningful information to the debugger.

2 Architecture

There are two core ideas behind DDB.

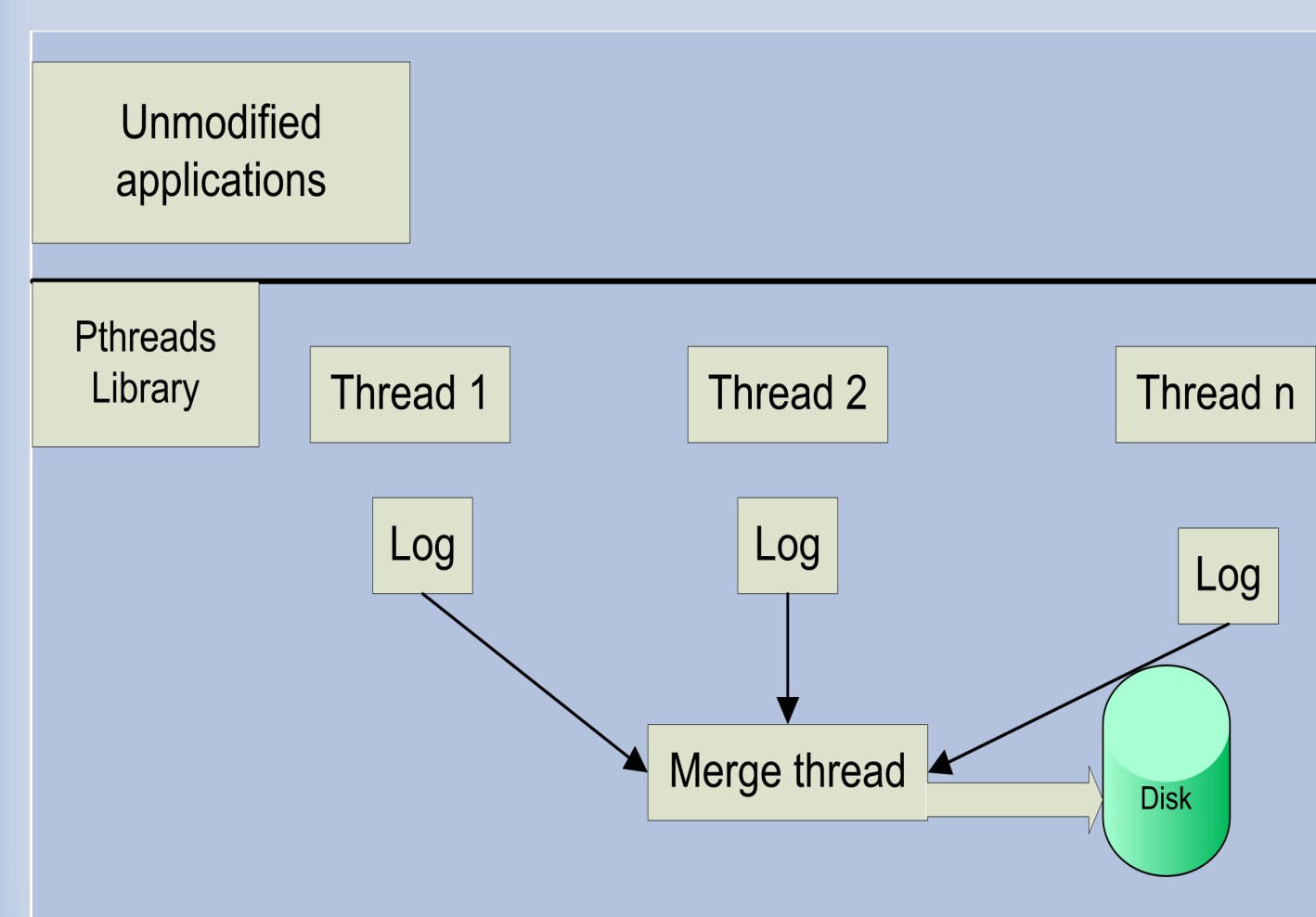
- Capture a lightweight trace during runtime by only recording a small amount of data and shift all the complexity to the debugging process which occurs offline.
- Do not record program inputs. We don't need to do a deterministic replay of the whole program, just to replay the observed deadlock bug.

DDB is made up of two components.

- The capture component is active at runtime and records synchronization operations inside the pthreads library.
- The replay component runs in a debugging environment and replays the events in the order they occurred in the trace.

Trace capture

- Trace Lock and Unlock operations -> obtain a partial order by preserving interleaving constraints.
- Tracing is done inside the pthreads library with a low overhead. Unmodified software only needs to be linked against our patched version of pthreads.
- Each thread keeps a queue of its own locking activity. The logs are merged offline.

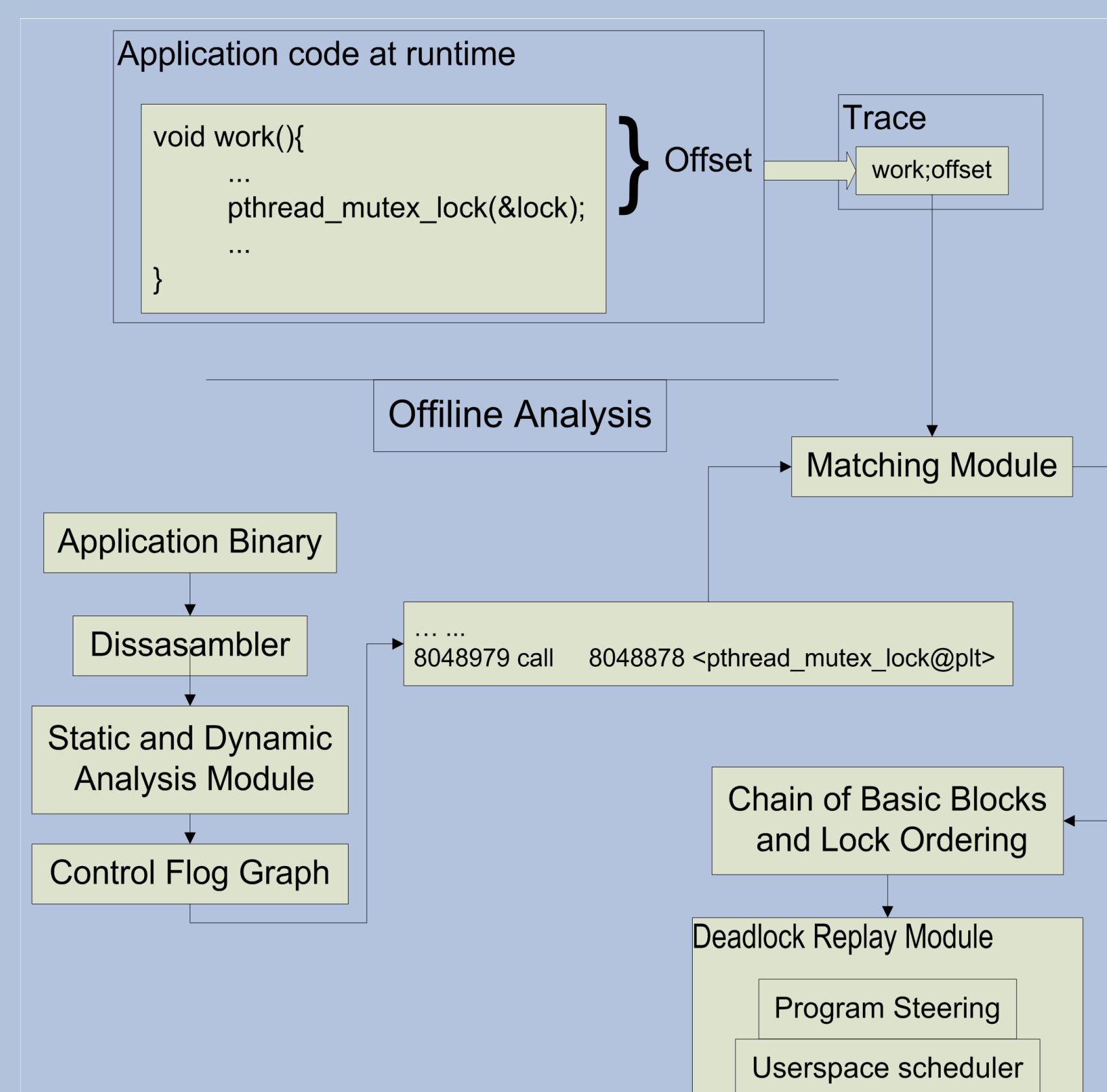


Tracing Architecture

Applications are not aware of the tracing component. The log data structures are one per thread to keep the overhead of tracing low. Initial tests with a micro benchmark show less than 20% overhead and very likely to be improved by future optimizations.

Deadlock Replay

- The replay part occurs at the developer's site.
- The developer has access to the sourcecode and the replay can take longer if necessary.
- Match the control flow graph (CFG) with the trace to identify a set of execution paths that replay the deadlock. We use runtime *program steering* to control program execution using dynamic binary rewriting (inspired by Enhancing Server Availability and Security Through Failure-Oblivious Computing, OSDI'04).
- Alternative: backward propagate constraints on variables to manufacture the right set of inputs necessary to replay the trace.



Replay Component Architecture

The replay component first builds the CFG of the application. Then, the matching module inspects the trace and obtains an execution path made out of chains of basic blocks and the order in which they need to execute for an accurate replay of the trace.

The deadlock replay module enforces the specified thread schedule and steers the program execution on the desired path using dynamic binary rewriting.

Implementation

- Tracing is done inside the *libthr* FreeBSD library.
- The trace is execution independent and might be replayed in a different OS.
- Initial results of replaying in Linux, traces obtained on FreeBSD.
- The replay environment is based on PIN
- Thread scheduling is based on futexes.
- Implemented a prototype of program steering based on instrumenting basic blocks, branch instructions and thread start routines.

3 Status and future work

```
r= rand()%2;
if(r) {
  pthread_mutex_lock (&lock0);
  pthread_mutex_lock (&lock1);
  ...
} else {
  pthread_mutex_lock (&lock1);
  pthread_mutex_lock (&lock0);
  ...
}
```

We can currently replay some deadlocks such as the ones caused by this code using program steering.

Open questions

- program steering may not always produce a possible execution.
- backward propagate the constraints on variables and inputs to manufacture a set of inputs that produce the desired execution.
- semantic compression of the trace: discard locking events and inputs that do not interfere with the deadlock.
- static analysis and dynamic analysis on the source code can be used to eliminate many of the tracing points and further decrease the overhead at runtime.
- Infrequent fine-grained checkpointing for reducing the replay speed of long running programs.