TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Deadlock Immunity: Enabling Systems to Defend Against Deadlocks

H. Jula, D. Tralamazza, C. Zamfir, G. Candea

presented by Bjoern Doebel

Dresden, 2009-09-08

Deadlocks

Deadlock bugs

- Study [16] (105 bugs, 31 deadlocks)
  - “Some 22% of the deadlock bugs are caused by one thread acquiring resource held by itself.”
  - “Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most two resources.”
  - “Many (61%) of the examined deadlock bugs are fixed by preventing one thread from acquiring one resource. Such fix can introduce non-deadlock concurrency bugs.”

Deadlock avoidance done wrong

```
Thread 1                          Thread 2 ··· Thread n        Monitor thread
void buf_flush_try_page() {                                    void error_monitor_thread() {
    ...
    rw_lock(&lock);               rw_lock(&lock);                if(lock_wait_time[i] >
                                                                         fatal_timeout)
}                                                                  assert(0, "We crash the server;
                                                                      It seems to be hung.");
                                                               }
            MySQL buf0flu.c                                            MySQL srv0srv.c
```

Figure 3. A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than `fatal_timeout` amount of time. In this bug, programmers under-estimate the workload (`n` could be very large), and therefore the lock waiting time would frequently exceed `fatal_timeout` and crash the server. (We simplified the code for illustration)

Doing it (more) right - Dimmunix

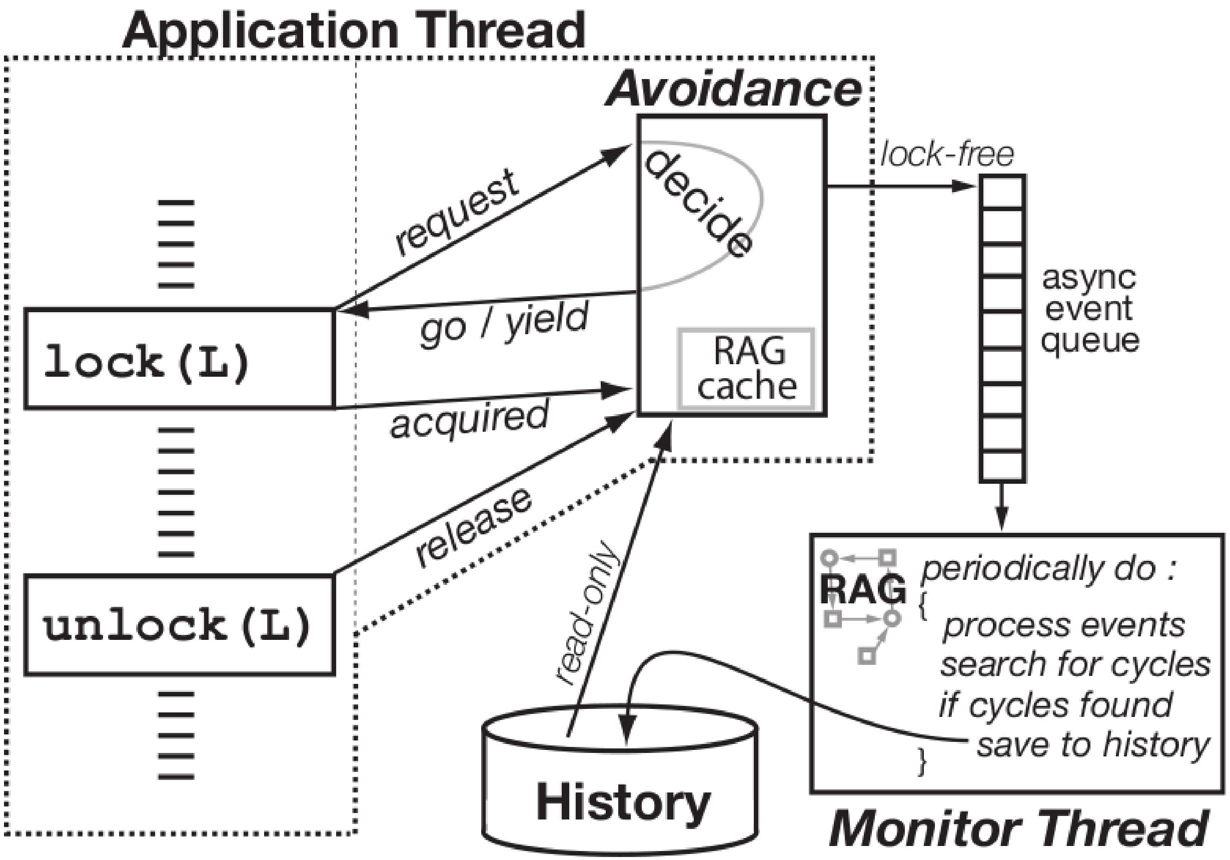

Figure 1: Dimmunix architecture.

Resource Allocation Graph

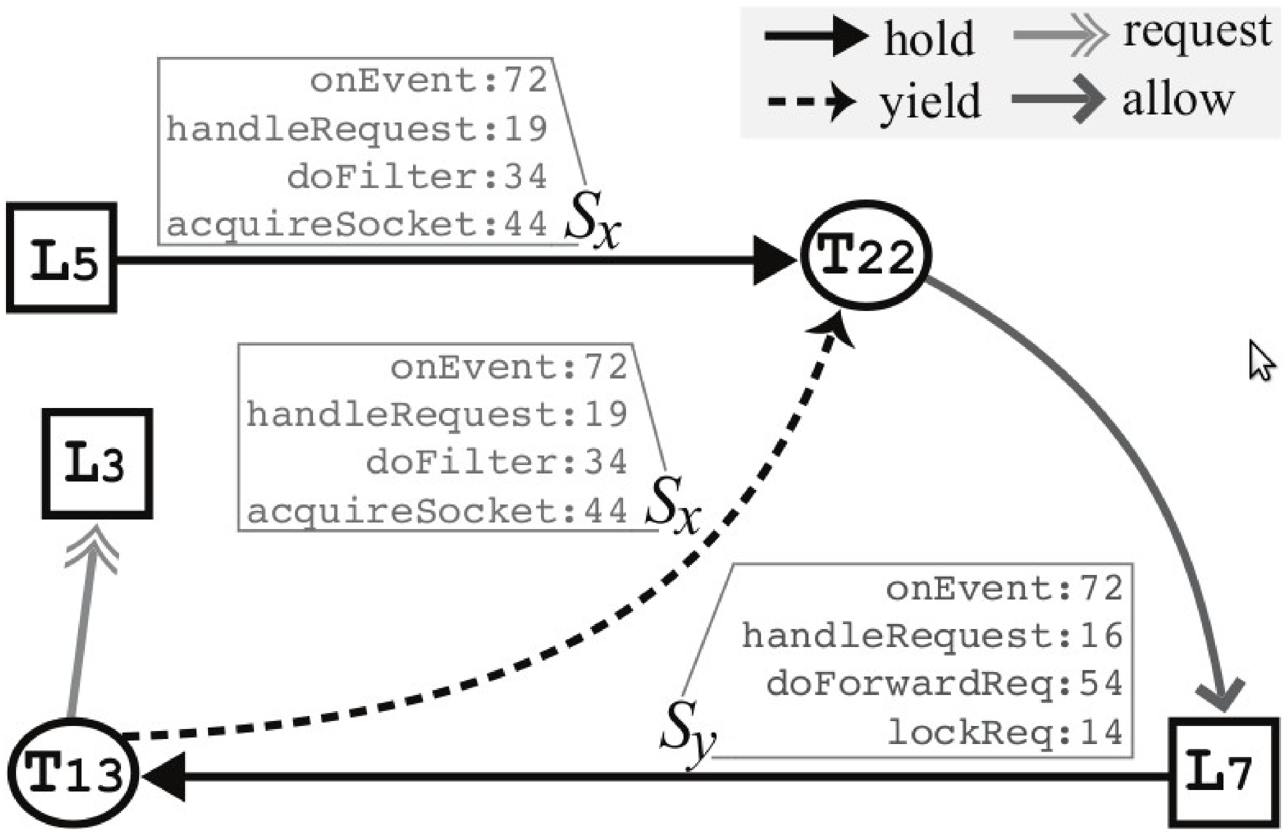

Figure 2: Fragment of a resource allocation graph.

Avoiding Deadlocks

- When DL is found:
  - Store “deadlock signature” of participating threads & wait for some recovery to happen.
- Later runs:
  - For each lock acquisition: check whether this would lead to a previously seen deadlock state
  - If so, make calling thread yield until at least one other participant has released its locks.
  - May lead to starvation – yield cycles.

Performance & Applicability

- Able to find & cure real-world deadlock bugs.
- Between 2 and 7 % runtime overhead.
- Lock throughput benchmark:
  - 4.5% overhead for pthreads, 17.5% for Java
- Overhead mostly from data updates and avoidance code.
  - Automatic calibration of signature stack depth
    - false positives vs. performance

TECHNISCHE UNIVERSITÄT DRESDEN

Remarks

- Signatures are control-flow based, w/o regarding data – false positives:

```
update(a,b)                  update(c,d)
    ..            <-->           ..
update(b,a)                  update(d,c)
```

```
update(x,y) {
   lock(x); lock(y);
   ..
   unlock(x); unlock(y);
}
```

More remarks

- Why can't we find those bugs before deploying?
  - Static source code analysis → RacerX
    - But: need access to source code
  - Static binary analysis
    - hard
  - Dynamic analysis → Valgrind Thread Checker

- RAG: request vs. allow edges?

Back to [16]

- "Some 22% of the deadlock bugs are caused by one thread acquiring resource held by itself."
  - Ignored due to availability of other mechanisms (non-recursive pthreads)
- "Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most two resources."
  - Means that real-world RAGs are not that complex.
- "Many (61%) of the examined deadlock bugs are fixed by preventing one thread from acquiring one resource. Such fix can introduce non-deadlock concurrency bugs."
  - Need to handle yield cycles.