Dynamic Livelock Analysis of Multi-threaded Programs

Runtime Verification (RV2012)
Sept 25-28, 2012

Malay K. Ganai
Systems Analysis & Verification Group
NEC Labs America
Outline

- Introduction
- Background
- Livelock Analysis
  - Lock Cycle Condition (LCC)
  - Livelock confirmation: detect and induce
- Implementation: CBuster tool
  - Binary instrumentation framework: record/replay
- Case Study: Sqlite-based application
- Conclusion
Introduction

- Multithreaded Programming is error prone
  - lack of forward progress translates into loss of availability, poor performance

- Deadlock: one or more threads are blocked forever without consuming CPU cycles

- Livelock: one or more threads consume CPU cycles but no useful work is done (busy-waiting analog of deadlock) [Starvation, infinite execution]
SQLite

- SQLite: embedded light weight database engine
  - Supports ACID transaction
  - Used in applications such as firefox, adobe, iPhone
  - Reader/writer locks using non-blocking file locks
  - Built-in safety detectors:
    - Prevents starvation: preventing new readers, allowing writer (committer) to finish
    - Detect and report potential deadlock, refuses to commit (throw SQLITE_BUSY to application)
Livelock exhibits strangely

“Strange sqlite_busy deadlock behavior:”

“I am seeing the equivalent of a deadlock, with SQLite_BUSY being returned forever from my code which has 2 threads using SQLite. I can repro this at will. Each thread is using its own connection to the sqlite database, so they are not sharing the same connection ...” april 2009

“It is a problem because people have asked for more concurrency. So we provide more concurrency. But concurrency comes at a price: namely the danger of a deadlock. You can ...enable shared-cache mode and then on the reader thread set "PRAGMA read_uncommitted=ON". This will allow the reader to see uncommitted changes from the writer, and hence never block for anything. You will get great concurrency and no possibility of deadlock. The downside there is that you lose isolation - the "I" in "ACID". Choose your poison.” D. Richard Hipp (the main developer)
Our Focus: Livelock

- Livelock due to **infinite execution** where one or more thread in a group are acquiring and releasing resources in busy-wait cycles to avoid deadlocks.

- It involves
  - **Intra-thread** cycle (while/for loops)
  - **Inter-thread** cycle (potential lock cycles)
Example: Liveloop Potential

G_lock(m):
lock(g);
lock(m);
unlock(g);

G_trylock(m):
lock(g);
r = trylock(m);
unlock(g);
return r;

while (true) {
    G_lock(m1)
    r0 := G_trylock(m2);
    If (r0 = fail) {
        unlock(m1);
        continue;
    }
    ...
    unlock(m2)
    unlock(m1)
    break;
}

Thread:1

While (true) {
    G_lock(m2)
    r1 := G_trylock(m1);
    If (r1 = fail) {
        unlock(m2);
        continue;
    }
    ...
    unlock(m1)
    unlock(m2)
    break;
}

Thread:2

Adapted from SQLite user posts
Our Goals for Livelock Analysis

- Handle real applications
  - Nested/non-nested, recursive locking scheme
  - Shared/exclusive i.e., reader/writer locks
  - Blocking/non-blocking (such as trylock/timedlock)
  - Mutex upgrades from shared to exclusive directly
  - Any combination of the above locking schemes

- Need for a new lock cycle condition that can detect deadlock/livelock potentials precisely.

- Extends existing algorithm (e.g., Goodlock) to exploit engineering advancements made in cycle detection
Background: Deadlock detection

Lock Graph:

- **Nodes** are mutex resources
- **Edge** (n,m) is labeled with tuple <t,m,L> where thread t with a lockset L, n∈L, is acquiring mutex m∉L. (n: first acquire, m: second acquire)
Deadlock cycle condition (DCC)

**DCC**: A cycle in a lock graph is a deadlock potential iff

a) All threads are distinct

b) Mutex to be acquired (i.e., second acquire) is held by the next thread (i.e., first acquire)

c) A second acquire does not happen-before a first acquire

d) Locksets at second acquire do not intersect pairwise

- Cycles \((m_1, m_g), (m_2, m_g)\) satisfy DCC (but spurious due to trylock)

- Cycle \((m_1, m_2)\), does not satisfy condition DCC:(d)
Proposed Livelock Analysis: Basic Steps

- **Static step:** We propose a lock cycle condition (LCC) to detect inter-thread cycles
  - detect livelock and deadlock potentials
  - handle general locking scheme: nested/non-nested, shared/exclusive, blocking/non-blocking

- **Dynamic step:** We identify intra-thread cycles by orchestrating a partial-order schedule that induces a livelock potential identified in the static step.
Definitions: Lock order set

Mutexes (m):

- \( m = \{ \text{ID}, <\text{Exclusive} | \text{Shared}> \} \)
- Two mutexes \( m, m' \) conflicts iff \( m.\text{ID} = m'.\text{ID} \), and, one of them is held in exclusive state

Lock order set (\( L \)):

- Ordset of mutexes, ordered as per acquisition, with no two mutexes having the same ID (similar to lockset but with order history)
Lock order graph

Nodes are mutex resources, $m \equiv \{\text{ID}, <E|S>\}$

Edge $(n,m)$ is a labeled with tuple $<t,m,L>$, where thread $t$ with a lock order set $L$, $n \in L$, is acquiring mutex $m$ that does not conflict with any $p \in L$. 
Lock cycle condition (LCC)

LCC: A cycle in a lock order graph is defined as a lock cycle iff

a) All threads are distinct
b) Mutex to be acquired (i.e., second acquire) is held by the next thread in conflicting state (i.e., first acquire)
c) A second acquire should not happen-before first acquire
d) Lock order sets at second acquire do not intersect pairwise on a conflicting mutex that was last acquired before first acquires of the corresponding pair threads.

(changes from DCC are highlighted)
LCC: An Example

Cycles \((m_1,m_g), (m_2,m_g)\) satisfy LCC

\[\text{Cycle (}m_1,m_2\text{), is a lock cycle as it satisfies condition LCC:(d)}\]

\[m_1/m_2: \text{first/second acquire of } t_1\]
\[m_2/m_1: \text{first/second acquire of } t_2\]
\[m_g \text{ is acquired by } t_1,t_2 \text{ after first acquire}\]
**Types of lock cycles**

**Trylock:** A lock cycle where all second acquires use non-blocking mutexes (such as trylock/timedlock). This is also a livelock potential.

**Deadlock:** A lock cycle where all second acquires use blocking mutexes (such as lock).

**Mixed:** Neither a trylock nor deadlock cycle (uninteresting).
Comparing DCC vs. LCC

Case A: (mutexes are acquired in only exclusive state)

- DCC(d) and LCC(d) are different, rest (a)-(c) are same.
- DCC(d) do not allow any common conflicting mutex at second acquires
- LCC(d) allow common conflicting mutexes at second acquires that was acquired after first acquire but not before.

For this case, a DCC cycle is also a LCC cycle.
Comparing DCC vs. LCC

Case B: (mutexes are acquired in exclusive/shared state)

- **DCC finds a spurious cycle**
  - Ex: $m_0, m_1$ are acquired in shared states. The cycle $(m_0, m_1)$ is marked a DCC cycle, but not a lock cycle as LCC(b) is not satisfied.

- **DCC can miss a deadlock**
  - Ex: $m_0, m_1$ are first acquired in shared state, and then both are upgraded to exclusive. DCC(d) is not satisfied, but is flagged correctly as deadlock cycle using LCC.

![Diagram showing the states and transitions for DCC and LCC cases.]

Underline denotes shared state.
Correctness

Valid Cycle: A lock cycle is valid iff all the threads in the cycle can acquire (respecting mutual exclusion, and happens-before ordering) the corresponding first mutexes before any thread can acquire the corresponding second mutex.

Each lock cycle is a valid cycle.
Inducing livelock

Guard: (Count=0)
Actions:
State := Head
Count := CycleLen

Guard: (Count≠0)
Actions:
State := Body
Count := CycleLen

Guard: (Count=0)
Actions:
State := Tail
Count := CycleLen

Guard: (Count≠0)

Global Scheduler
Inducing livelock (cntd...)

Guard:
(State = Tail \land
rt_{t}.m = m \land
rt_{t}.etype = etype)

Actions:
Count := Count - 1

Guard:
(State = Tail \lor rh_{t}.cxt \neq cxt \lor rh_{t}.etype \neq etype)

Actions:
if (rh_{t}.cxt = cxt \land rh_{t}.etype = etype)
waitFor(State=Head);

Guard:
(State = Head \land
rh_{t}.cxt = cxt \land
rh_{t}.etype = etype)

Actions:
Count := Count - 1, rt_{t}.m := m

Guard:
(State = Body \land
rb_{t}.cxt = cxt \land
rb_{t}.etype = etype)

Actions:
Count := Count - 1

Guard:
(State = Body \lor
rt_{t}.m \neq m \lor
rt_{t}.etype \neq etype)

Actions:
if (rt_{t}.m = m \land
rt_{t}.etype = etype)
waitFor(State=Tail);

Guard:
(State = Head \lor
rb_{t}.cxt \neq cxt \lor
rb_{t}.etype \neq etype)

Actions:
if (rb_{t}.cxt = cxt \land
rb_{t}.etype = etype)
waitFor(State=Body);
Example: Inducing Livelock Potential

G_lock(m):
lock(g);
lock(m);
unlock(g);

G_trylock(m):
lock(g);
\( r \) = trylock(m);
unlock(g);
return \( r \);

while (true) {
    G_lock(m1)
    r0 := G_trylock(m2);
    if (r0 = fail) {
        unlock(m1);
        continue;
    }
    \ldots
    unlock(m2)
    unlock(m1)
    break;
}

Thread:1

while (true) {
    G_lock(m2)
    r1 := G_trylock(m1);
    if (r1 = fail) {
        unlock(m2);
        continue;
    }
    \ldots
    unlock(m1)
    unlock(m2)
    break;
}

Thread:2

Adapted from SQLite user posts
Cbuster: Lock cycle buster tool

- X86 Binary-based instrumentation tool
  - Use LD_PRELOAD facility (in linux)
  - Runtime hooks on pthread primitives
  - Scheduler: global and thread local
- Three phases
  - **Record**: thread, event type, context, vector clock
  - **Analyze** (static): use LCC to identify deadlock/livelock (inter-thread cycles)
  - **Replay** (dynamic): Confirm livelock in an orchestrated execution to induce global transition sequence Head → Body → Tail → Head at least once.

(Record+Replay: 1.7KLOC, Analyze: 1KLOC C++)
Case Study: SQLite-based Application

```c
void thread1(void *arg) {
    sqlite3_open("index.db", &db);
    sqlite3_exec(db, "CREATE TABLE scoreTbl (name TEXT, score INTEGER)", ...);
    sqlite3_exec(db, "INSERT INTO scoreTbl VALUES (Tom,95)", ...);
    sqlite3_exec(db, "INSERT INTO scoreTbl VALUES (Sam,98)", ...);
    ... 
    pthread_create(&T2, NULL, thread2, NULL);
    sqlite3_exec_wrap(db, "UPDATE scoreTbl SET score=98 WHERE name=Tom");
    ...
}

void thread2(void *arg) {
    sqlite3_open("index.db", &db);
    sqlite3_exec_wrap(db, "UPDATE scoreTbl SET score=95 WHERE name=Sam");
    ...
}

int sqlite3_exec_wrap(sqlite3 *db, char *sql_cmd) {
    while (1) {
        if (sqlite3_exec(db, sql_cmd) != SQLITE_BUSY) return err;
    }
}
```

Adapted from SQLite user posts
Internals of SQLite

- SQLite: (Rollback Journal Mode)
  - 5 lock states of DB
    - Unlock, shared, reserved, pending, and exclusive
  - 3 flocks (file locks)
    - flock_P (pending), flock_R(reserved), and flock_D (exclusive)
  - DB Reader
    - trylockS(flock_P), trylockS(flock_D); unlock(flock_P)
  - DB Writer
    - trylockS(flock_P), trylockS(flock_D); unlock(flock_P); [Shared state]
    - trylock(flock_R), [Reserved State: prevent other writers]
    - trylock(flock_P), [Pending State: prevents new readers]
    - trylock(flock_D)), [Exclusive State: commits db]
**Trace of SQLite Example**

/* Shared lock state */
lock(mg)
trylockS(flock_P)
trylockS(flock_D)
unlock(flock_P)
unlock(mg)

/* Reserved lock state */
lock(mg)
trylock(flock_R)
unlock(mg)

/* Exclusive lock state */
lock(mg)
trylock(flock_P)
trylock(flock_D)
unlock(mg)

Thread: 1

/* Shared lock state */
lock(mg)
trylockS(flock_P)
trylockS(flock_D)
unlock(flock_P)
unlock(mg)

/* Reserved lock state */
lock(mg)
trylock(flock_R)
unlock(mg)

/* Exclusive lock state */
lock(mg)
trylock(flock_P)
trylock(flock_D)
unlock(mg)

Thread: 2
C1 Lock cycle (trylock)

/* Shared lock state*/
lock(m_g)
trylockS(flock_P);
trylockS(flock_D);
unlock(flock_P);
unlock(m_g);

/* Reserved lock state */
lock(m_g);
trylock(flock_R);
unlock(m_g);

/* Exclusive lock state */
lock(m_g);
trylock(flock_P);
trylock(flock_D);
unlock(m_g);

<t1, m_R, {m_D, m_g}>

<flock_D> trylock cycle <flock_R>

<t2, m_D, {m_D, m_R, m_g, m_P}>

m_D ≡ flock_D
m_P ≡ flock_P
m_R ≡ flock_R
C2 Lock cycle (trylock)

/* Shared lock state */
lock(mg)
trylockS(flock_P);
trylockS(flock_D);
unlock(flock_P);
unlock(mg);

/* Reserved lock state */
lock(mg);
trylock(flock_R);
unlock(mg);

/* Exclusive lock state */
lock(mg);
trylock(flock_P);
trylock(flock_D);
unlock(mg);

<t1, mD, {mD,mR, mg ,mP}>

flock_D

trylock cycle

<t2, mD, {mD,mR, mg, mP}>

mD≡ flock_D
mP≡ flock_P
mR≡ flock_R
C3 Lock cycle (mixed)

/* Shared lock state*/
lock(mg)
trylockS(flock_P);
trylockS(flock_D);
unlock(flock_P);
unlock(mg);

/* Reserved lock state */
lock(mg);
trylock(flock_R);
unlock(mg);

/* Exclusive lock state */
lock(mg);
trylock(flock_P);
unlock(mg);

mD ≡ flock_D
mP ≡ flock_P
mR ≡ flock_R
/* Shared lock state */
lock(m_g)
trylockS(flock_P);
trylockS(flock_D);
unlock(flock_P);
unlock(m_g);

/* Reserved lock state */
lock(m_g);
trylock(flock_R);
unlock(m_g);

/* Exclusive lock state */
lock(m_g);
trylock(flock_P);
trylock(flock_D);
unlock(m_g);

C4 Lock cycle (mixed)

<t_1, m_R, \{m_D,m_g\}>

mixed cycle

<t_2, m_g, \{m_D,m_R\}>

m_D \equiv \text{flock}_D
m_P \equiv \text{flock}_P
m_R \equiv \text{flock}_R
Summary

- Presented lock cycle condition, a general scheme for detecting deadlock/livelock potential
  - Subsumes previous DCC, improves quality without any appreciable overhead
- Handles general locking schemes
  - Blocking/non-blocking, shared/exclusive, file locks, nested/non-nested
- Incorporated in a binary-based instrumentation tool (Cbuster) with record and replay capability
  - Demonstrated over Sqlite-based application
- In future, we would like to expand our case studies, and integrate with a symbolic analysis.