Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs

Speaker: Martin Lanter
Deadlock

- Circular-mutex-wait deadlocks in conventional shared-memory multithreaded programs.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire lock 1</td>
<td>acquire lock 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>acquire lock 2</td>
<td>acquire lock 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>release locks</td>
<td>release locks</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Gadara

- Intelligently postpones lock acquisition attempts
- All circular-mutex-wait deadlocks are eliminated
- No new deadlocks or other liveness or progress bugs
- No global reasoning necessary for the programmer
Petri nets

- A Petri net is a triple $N = (P, T, F)$
  - $P$ is a set of states, called places.
  - $T$ is a set of transitions.
  - $F$ is a set of flow relations
Lock for critical sections

```plaintext
lock() → Critical section → unlock()

lock() → Critical section → unlock()
```

`lock()` and `unlock()` are used to synchronize access to critical sections in concurrent computation.
Gadara Architecture
Philosopher

```c
void * philosopher(void *arg) {
    ...  
    if (random()>0.5) {
        /* grab A first */
        pthread_mutex_lock(&forkA);
        pthread_mutex_lock(&forkB);
    } else {  
        /* grab B first */
        pthread_mutex_lock(&forkB);
        pthread_mutex_lock(&forkA);
    }
    eat();
    pthread_mutex_unlock(&forkA);  
    pthread_mutex_unlock(&forkB);  
    ...  
}
```

http://gadara.eecs.umich.edu/papers/UMGS10_poster.pdf
Philosopher

http://gadara.eecs.umich.edu/papers/UMGS10_poster.pdf
Philosopher

http://gadara.eecs.umich.edu/papers/UMGS10_poster.pdf
Philosopher

http://gadara.eecs.umich.edu/papers/UMGS10_poster.pdf
void * philosopher(void *arg) {
    ...
    if (random() > 0.5) {
        /* grab A first */
        gadara_lock(&forkA, &ctrlplace);
        pthread_mutex_lock(&forkB);
    } else {
        /* grab B first */
        gadara_lock(&forkB, &ctrlplace);
        pthread_mutex_lock(&forkA);
    }
    eat();
    gadara_replenish(&ctrlplace);
    pthread_mutex_unlock(&forkA);
    pthread_mutex_unlock(&forkB);
    ...
}
Siphon

- Set of places that never regains a token if it becomes empty
Control logic synthesis

- **Siphon**
  - Set of places that never regains a token if it becomes empty

- **Supervision Based on Place Invariants technique (SBPI)**
  - Solves a system of inequalities

- Insert control places that encode feedback control logic

- Apply SBPI repeatedly to fix newly created siphons
Gadara control logic

- Provably Deadlock free
- Maximally permissive
  - With respect to the program model
- Decentralized
- Fine-grained
- Highly concurrent

- Conservative
  - Needs annotations
Limitations

- Limits inherent to the problem domain
  - Inevitable deadlocks
  - E.g. repeatedly locking a nonrecursive mutex

- Artifacts of the current prototype
  - False paths problem

- Only Pthread functions
  - Homebrew synchronization primitives must be annotated
Benchmark

- C/Pthread client-server publish-subscribe application
  - subscribe to a channel
  - publish data to a channel
  - request a snapshot of all of their current subscriptions

- 12 worker threads
- 4 x 1024 clients
# Benchmark Results

<table>
<thead>
<tr>
<th></th>
<th>Heavy Load Throughput (Mbit/s)</th>
<th>Light Load Resp. Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL free</td>
<td>94.25</td>
<td>10.83</td>
</tr>
<tr>
<td>Gadarized</td>
<td>76.88</td>
<td>10.52</td>
</tr>
<tr>
<td>STM</td>
<td>47.15</td>
<td>66.70</td>
</tr>
</tbody>
</table>
Gadara Performance

- Runtime performance overhead < 18%
  - Typically negligible

- Compile time overhead tolerable
  - Not worse than build time
Credits

Yin Wang, Terence Kelly, Manjunath Kudlur
(EECS Department, University of Michigan)

Stéphane Lafortune, Scott Malke
(Hewlett-Packard Laboratories)