A Study of Dynamic Software Update Quiescence for Multithreaded Programs

Christopher M. Hayden, Karla Saur, Michael Hicks, Jeffrey S. Foster
Update Timing

- Well-defined *update points* make it easier to reason about update correctness

```c
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    dsu_update(); /* update point */
    /* loop body: typically handles a single program event */
  }
}
```

- Good candidates are *quiescent points* in loops which have little in-flight state
DSU and Threading

- *Timeliness* in multithreaded updates:
  - Full quiescence – all threads hit update point

- Concern - Updating at only specific points has the risk of delaying an update for too long, even *indefinitely*
Goals & Approach:

- **Questions:**
  - Quick full quiescence in multithreaded programs?
  - What blocking calls impede quick quiescence?

- **Created library: QBench**
  - Interrupt blocking to facilitate quiescence with minimal delay
  - *Measures time* from update request to full quiescence
  - Idioms we develop in QBench we can roll into DSU systems
Update at Quiescent Points

- Update point 'qbench_update':
  - No update requested: call is a no-op
  - Update requested: calling thread blocks

```c
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update(); /* update point */
    /* loop body: typically handles a single program event */
  }
}
```

- Request an update by sending a SIGUSR2 signal
  - QBench installs a signal handler indicating update requested.
Threats to Quiescence

- Blocking calls in our experiments:

I/O:
Socket blocking on data

```
update pt
accept(....);
```
Thread x

Condition Variables:
Threads sharing a mutex

```
update pt
cond_wait(...);
```
Thread x
Blocking on I/O

Under normal circumstances an accept call will block until a connection is accepted.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {

        res = accept(sockfd, addr, addrlen);

        /* ... handle connection */
    }
}
```
Blocking on I/O

Under normal circumstances an accept call will block until a connection is accepted.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        res = accept(sockfd, addr, addrlen);

        /* ... handle connection */
    }
```

Under normal circumstances an accept call will block until a connection is accepted.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        res = accept(sockfd, addr, addrlen);
        if (res == -1 && errno == EINTR)
            continue;
        /* ... handle connection */
    }
}
```

A signal will interrupt `accept`, return -1, and set `errno` to `EINTR`. 
Under normal circumstances an accept call will block until a connection is accepted.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        res = accept(sockfd, addr, addrlen);
        if (res == -1 && errno == EINTR)
            continue;
        /* ... handle connection */
    }
}
```

A signal will interrupt `accept`, return -1, and set `errno` to `EINTR`.

Returns to top of even loop to immediately hit update point.
UNIX Signals

- Signals are usually handled by main thread
  - Main thread signals all threads not blocked by condition variables
Programmers guard against spurious wake-ups by placing `pthread_cond_wait` in a loop.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        pthread_mutex_lock(&mutex);
        while (!input_is_ready()) {
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        /* ... handle connection */
    }
}
```
Programmers guard against spurious wake-ups by placing `pthread_cond_wait` in a loop

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        pthread_mutex_lock(&mutex);
        while (!input_is_ready() && !qbench_update_requested()) {
            qbench pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        if (qbench_update_requested())
            continue; /* reaches qbench_update */
        /* ... handle connection */
    }
}
```
Blocking on Condition Variables

Programmers guard against spurious wake-ups by placing `pthread_cond_wait` in a loop

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        pthread_mutex_lock(&mutex);
        while (!input_is_ready() && !qbench_update_requested()) {
            qbench_pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        if (qbench_update_requested())
            continue; /* reaches qbench_update */
        /* ... handle connection */
    }
}
```
Programmers guard against spurious wake-ups by placing `pthread_cond_wait` in a loop.

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        pthread_mutex_lock(&mutex);
        while (!input_is_ready() && !qbench_update_requested()) {
            qbench_pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        if (qbench_update_requested())
            continue; /* reaches qbench_update */
        /* ... handle connection */
    }
}
```
Programmers guard against spurious wake-ups by placing `pthread_cond_wait` in a loop

```c
void *thread_entry(void *arg) {
    /* thread init code */
    while (1) {
        qbench_update();
        pthread_mutex_lock(&mutex);
        while (!input_is_ready() && !qbench_update_requested()) {
            qbenchpthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        if (qbench_update_requested())
            continue; /* reaches qbench_update */
        /* ... handle connection */
    }
}
```

- Allows thread to be signaled for update even when waiting on a condition variable
- Reports true if an update request is signaled
- Returns to top of even loop to immediately hit update point
Waking a Blocked Thread

- Condition Variables: Another thread must be available to signal
Waking a Blocked Thread

- Condition Variables: Another thread must be available to signal
**Experiments**

- We chose programs covering a wide range of domains.
- On average, 22 lines of code changed (including update points).
- Manual changes: changes beyond adding calls to QBench.

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC Total</th>
<th># of Threads</th>
<th>Upd Points</th>
<th>Changed LoC (†)</th>
<th>Required Manual Chgs</th>
</tr>
</thead>
<tbody>
<tr>
<td>httpd-2.2.22</td>
<td>232651</td>
<td>$2 + c^*, \ c = 3$</td>
<td>5</td>
<td>7 (5)</td>
<td>3 (Cond. Var. Loop)</td>
</tr>
<tr>
<td>icecast-2.3.2</td>
<td>17038</td>
<td>6</td>
<td>12</td>
<td>3 (3)</td>
<td>1 (Thread Sleeps)</td>
</tr>
<tr>
<td>iperf-2.0.5</td>
<td>3996</td>
<td>$3 + n^\circ, \ n = 1$</td>
<td>5</td>
<td>8 (3)</td>
<td>1 (Cond. Var. Loop)</td>
</tr>
<tr>
<td>memcached-1.4.13</td>
<td>9404</td>
<td>$2 + c^*, \ c = 4$</td>
<td>4</td>
<td>27 (4)</td>
<td>2 (libevent changes)</td>
</tr>
<tr>
<td>space-tyrant-0.354</td>
<td>8721</td>
<td>$3 + 2n^\circ, \ n = 5$</td>
<td>6</td>
<td>8 (6)</td>
<td>1 (Thread Sleeps)</td>
</tr>
<tr>
<td>suricata-1.2.1</td>
<td>260344</td>
<td>$8 + c^*, \ c = 3$</td>
<td>7</td>
<td>11 (6)</td>
<td>1 (libpcap break)</td>
</tr>
</tbody>
</table>

*Configurable: $c$ workers  \(^\circ\)Varies by $n$ connected clients  \(^\dagger\)Calls to QBench excluding update
Results

- Two Workloads:
  - Server idle (i.e., no connected clients)
  - Performing program-dependent work
- Nearly all programs quiesced in under 1ms
- Some would not quiesce without changes

<table>
<thead>
<tr>
<th>Program</th>
<th>w/Load (ms)</th>
<th>w/o Load (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All Chgs</td>
<td>UpdPt only</td>
</tr>
<tr>
<td>httpd-2.2.22</td>
<td>0.185</td>
<td>0.230</td>
</tr>
<tr>
<td>icecast-2.3.2</td>
<td>105.152</td>
<td>954.32</td>
</tr>
<tr>
<td>iperf-2.0.5</td>
<td>0.193</td>
<td>DNQ</td>
</tr>
<tr>
<td>memcached-1.4.13</td>
<td>0.166</td>
<td>DNQ</td>
</tr>
<tr>
<td>space-tyrant-0.354</td>
<td>0.426</td>
<td>20.583</td>
</tr>
<tr>
<td>suricata-1.2.1</td>
<td>0.503</td>
<td>68.098</td>
</tr>
</tbody>
</table>

DNQ = Does Not Quiesce
Results

- Two Workloads:
  - Server idle (i.e., no connected clients)
  - Performing program-dependent work
- Nearly all programs quiesced in under 1ms
- Some would not quiesce without changes

<table>
<thead>
<tr>
<th>Program</th>
<th>w/Load (ms)</th>
<th>w/o Load (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All Chgs</td>
<td>UpdPt only</td>
</tr>
<tr>
<td>httpd-2.2.22</td>
<td>0.185</td>
<td>0.230</td>
</tr>
<tr>
<td>icecast-2.3.2</td>
<td>105.152</td>
<td>954.32</td>
</tr>
<tr>
<td>iperf-2.0.5</td>
<td>0.193</td>
<td>DNQ</td>
</tr>
<tr>
<td>memcached-1.4.13</td>
<td>0.166</td>
<td>DNQ</td>
</tr>
<tr>
<td>space-tyrant-0.354</td>
<td>0.426</td>
<td>20.583</td>
</tr>
<tr>
<td>suricata-1.2.1</td>
<td>0.503</td>
<td>68.098</td>
</tr>
</tbody>
</table>

DNQ = Does Not Quiesce
Results

- Two Workloads:
  - Server idle (i.e., no connected clients)
  - Performing program-dependent work
- Nearly all programs quiesced in under 1ms
- Some would not quiesce without changes

<table>
<thead>
<tr>
<th>Program</th>
<th>w/Load (ms)</th>
<th>w/o Load (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All Chgs</td>
<td>UpdPt only</td>
</tr>
<tr>
<td>httpd-2.2.22</td>
<td>0.185</td>
<td>0.230</td>
</tr>
<tr>
<td>icecast-2.3.2</td>
<td>105.152</td>
<td>954.32</td>
</tr>
<tr>
<td>iperf-2.0.5</td>
<td>0.193</td>
<td>DNQ</td>
</tr>
<tr>
<td>memcached-1.4.13</td>
<td>0.166</td>
<td>DNQ</td>
</tr>
<tr>
<td>space-tyrant-0.354</td>
<td>0.426</td>
<td>20.583</td>
</tr>
<tr>
<td>suricata-1.2.1</td>
<td>0.503</td>
<td>68.098</td>
</tr>
</tbody>
</table>

DNQ = Does Not Quiesce
Summary & Future Work

- Demonstrated multithreaded quiescence **quickly** and with **little implementation complexity** for many programs with fixed update points.

- Time to quiescence ranged from 0.155 to 107.558 ms; most were below 1 ms.

- We plan to integrate the multi-threaded quiescent functionality back into Kitsune.