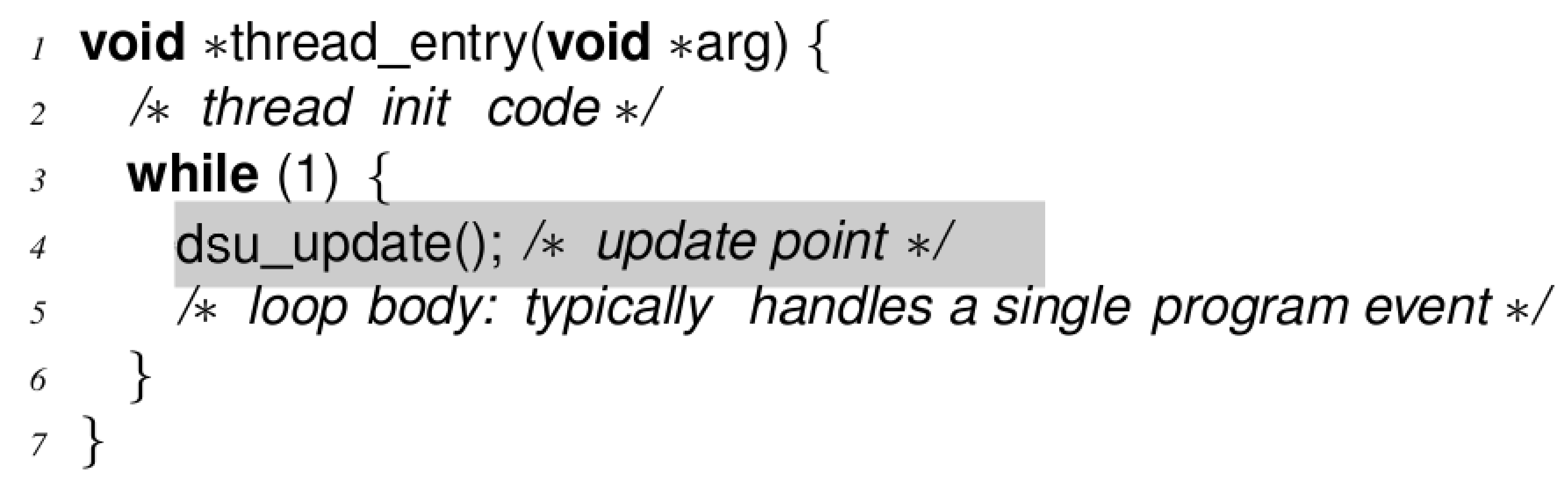
A Study of Dynamic Software Update Quiescence for Multithreaded Programs

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

Update Timing

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

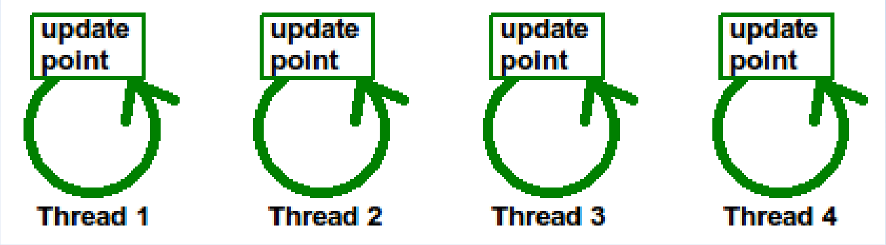


```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    dsu_update(); /* update point */
    /* loop body: typically handles a single program event */
  }
}
```

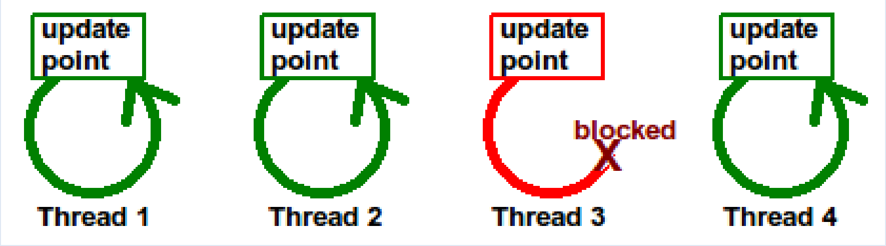
- Good candidates are *quiescent points* in loops which have little in-flight state

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



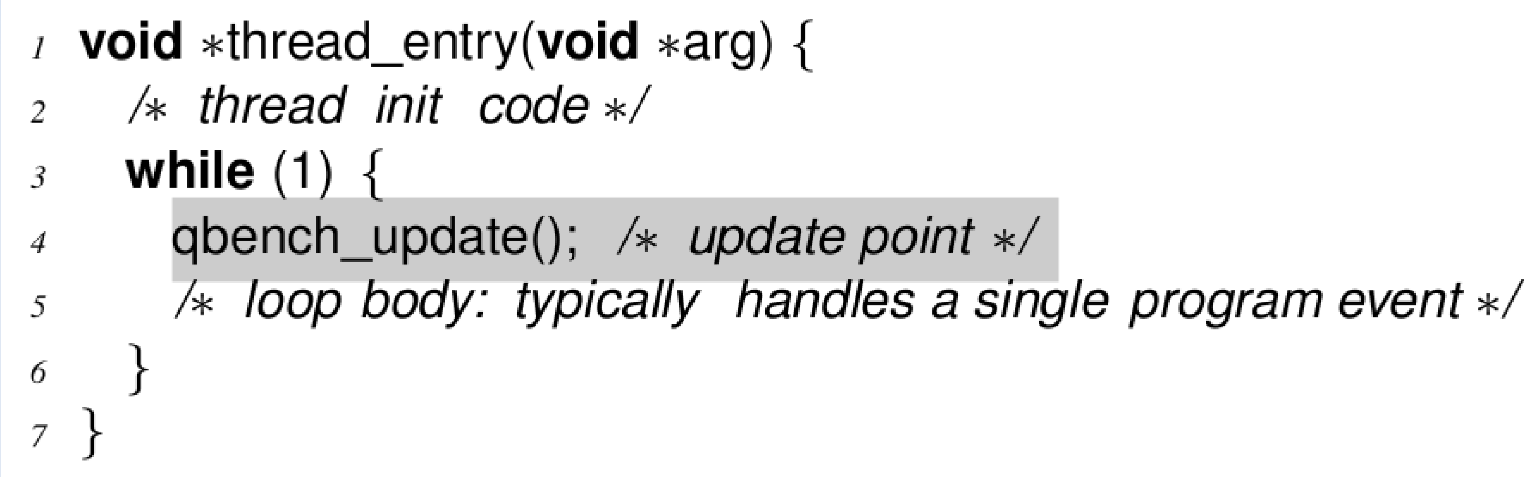
Goals & Approach:

- 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



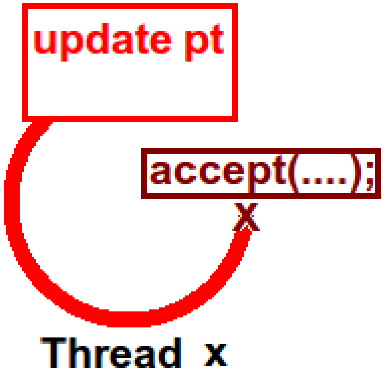
```
void *thread_entry(void *arg) {
   /* thread init code */
   while (1) {
      qbench_update(); /* update point */
      /* loop body: typically handles a single program event */
   }
}
```

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

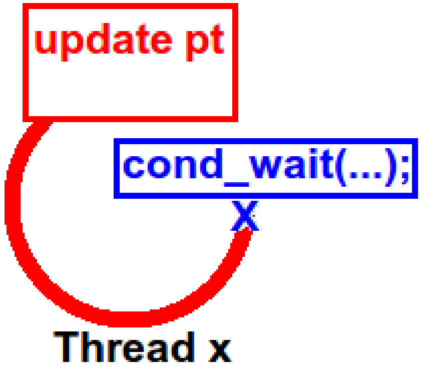
Threats to Quiescence

- Blocking calls in our experiments:

I/O:
Socket blocking on data

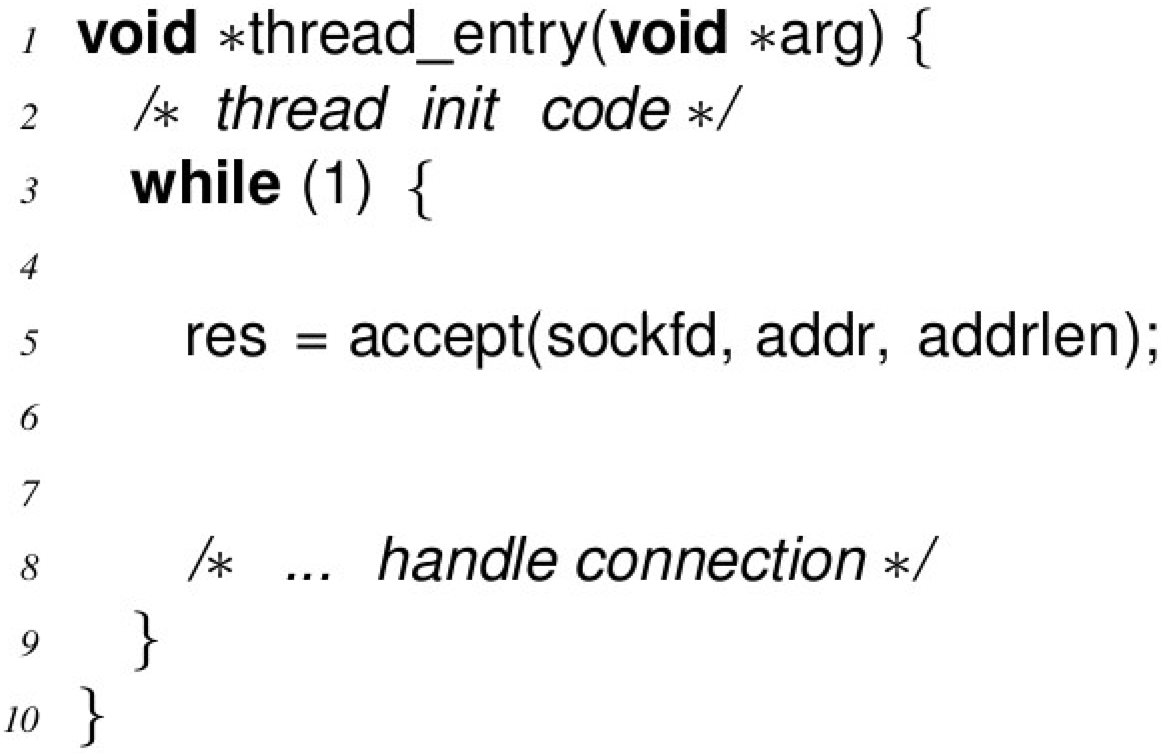


Condition Variables:
Threads sharing a mutex



Blocking on I/O

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



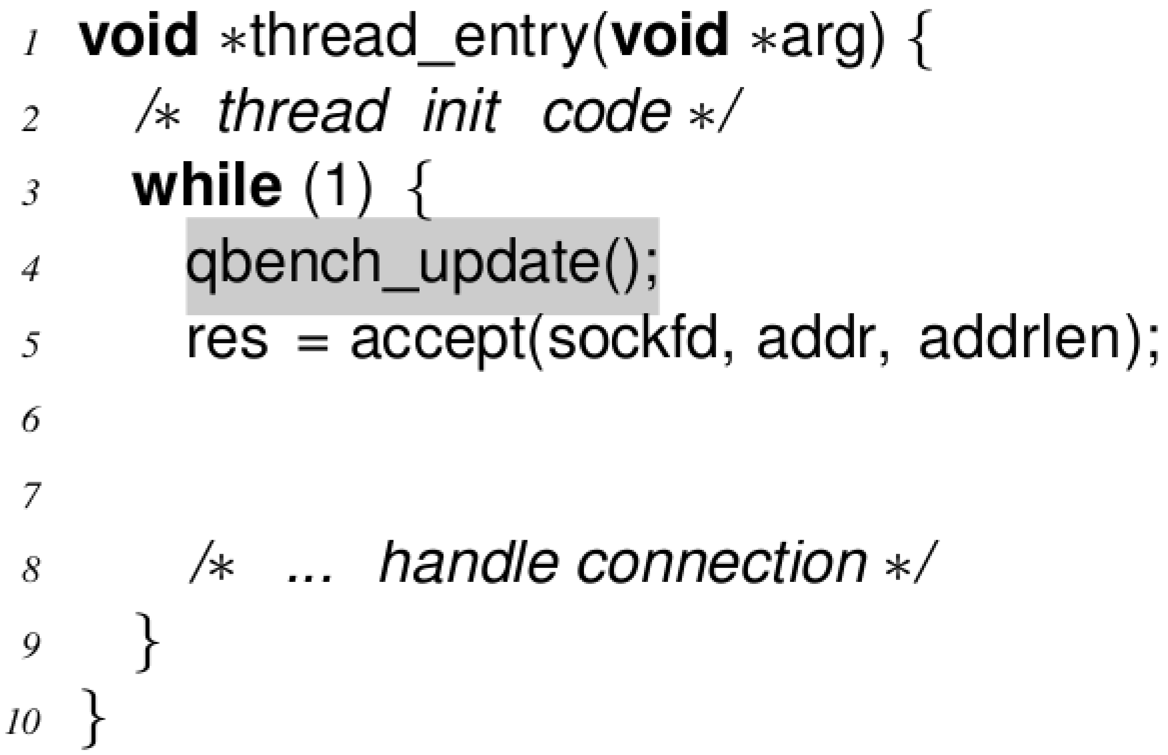
```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {

    res = accept(sockfd, addr, addrlen);


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

Blocking on I/O

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

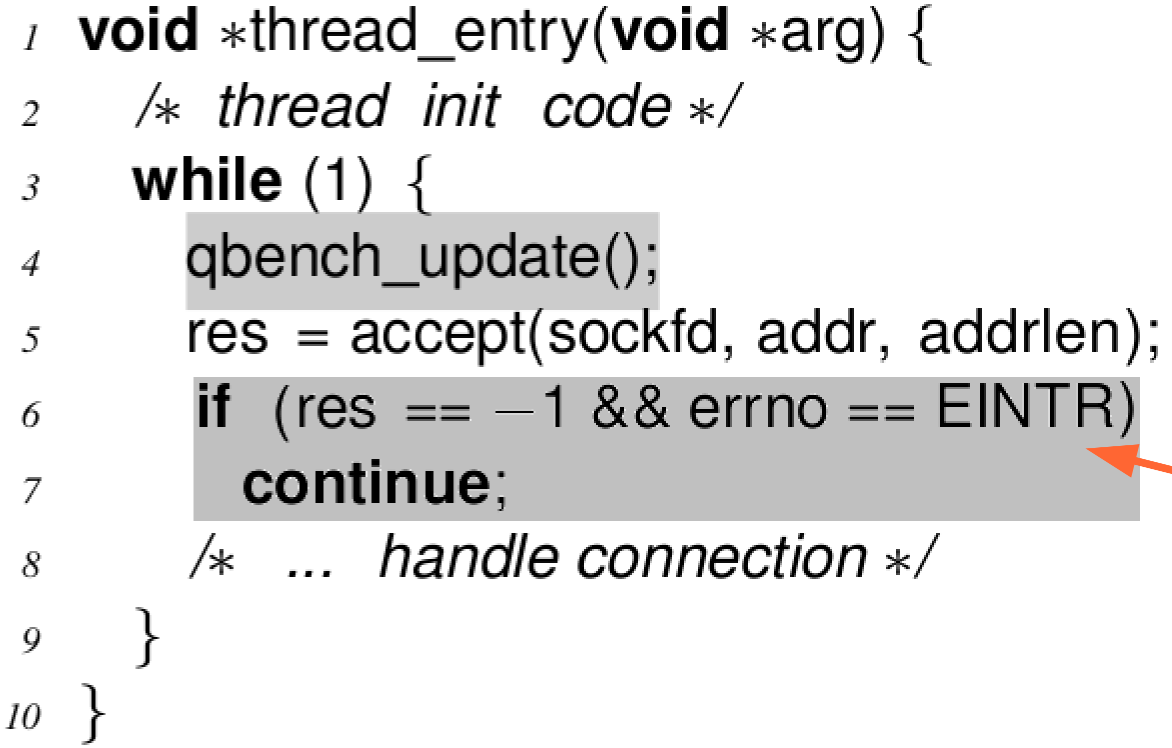


```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update();
    res = accept(sockfd, addr, addrlen);


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

Blocking on I/O

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

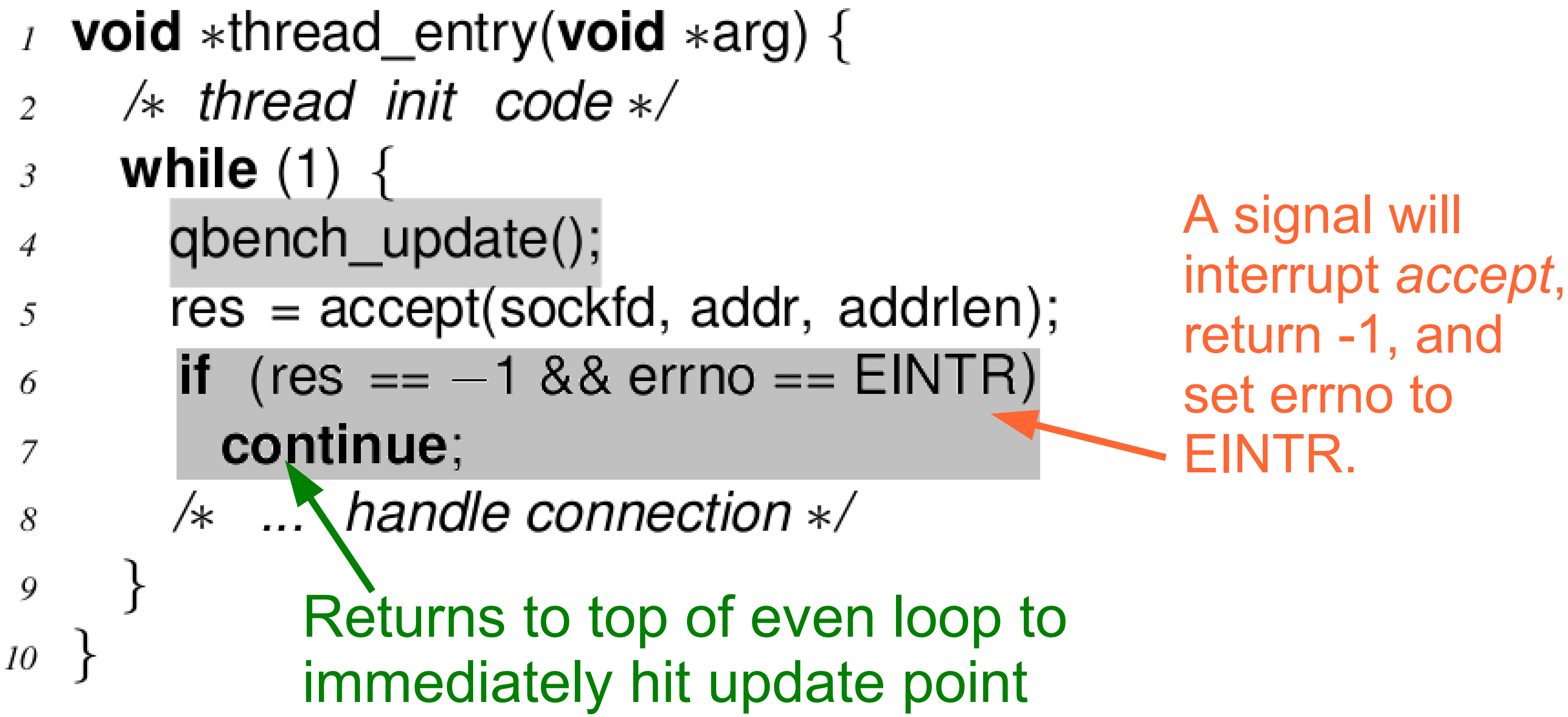


```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update();
    res = accept(sockfd, addr, addrlen);
    if (res == −1 && errno == EINTR)
      continue;
    /* ... handle connection */
  }
}
```

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

Blocking on I/O

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



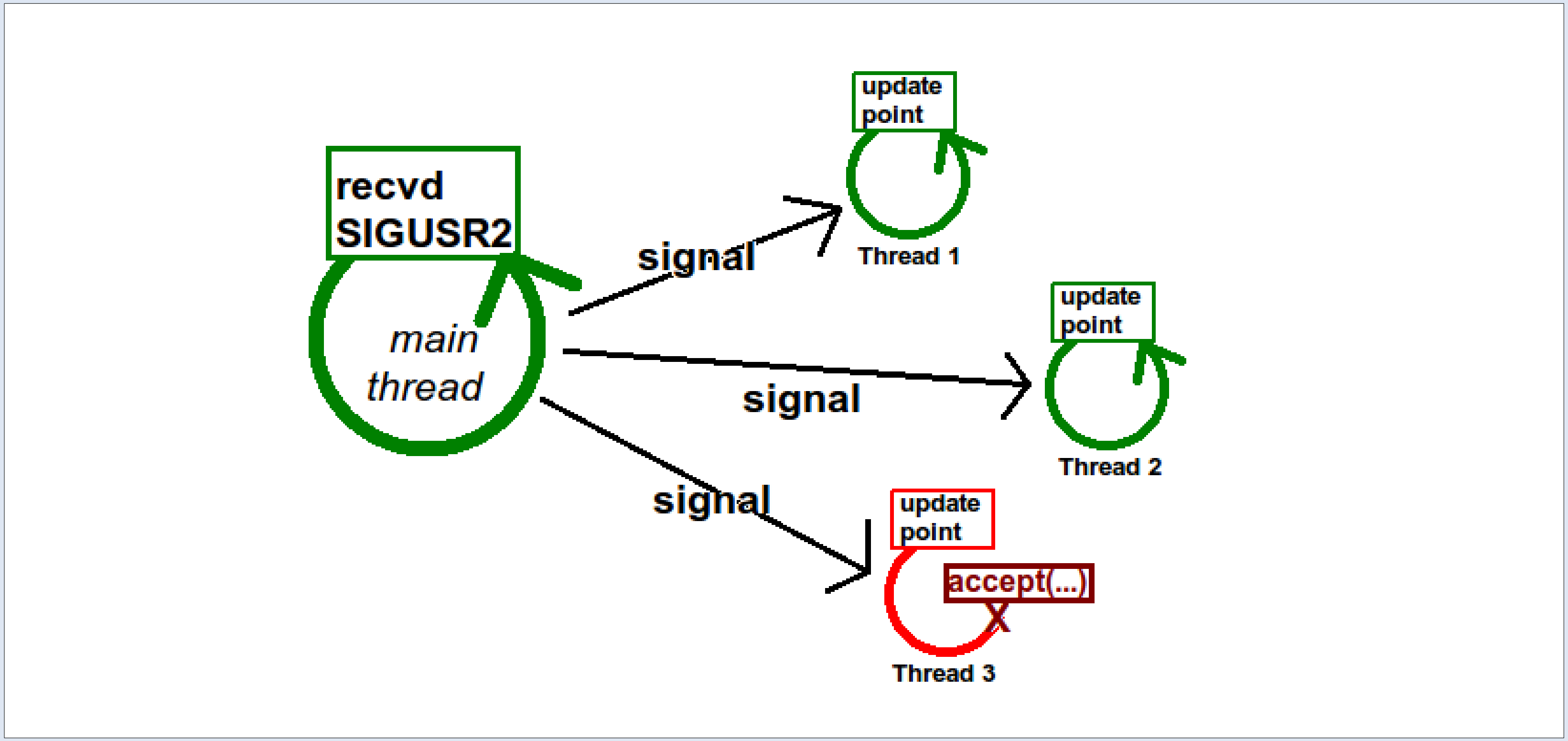
```
void *thread_entry(void *arg) {
  /*  thread  init   code */
  while (1)  {
    qbench_update();
    res  = accept(sockfd, addr,  addrlen);
    if   (res  ==  −1 && errno == EINTR)
      continue;
    /*   ...  handle connection */
  }
}
```

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

Returns to top of even loop to immediately hit update point

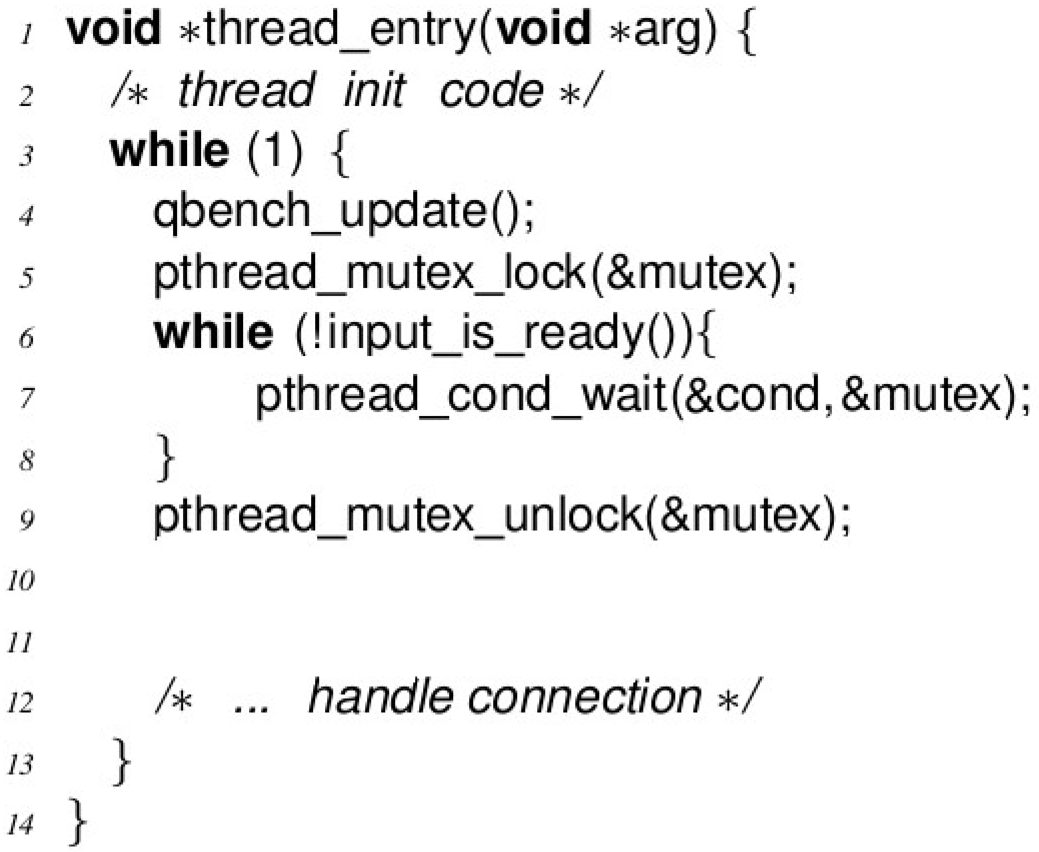
UNIX Signals

- Signals are usually handled by main thread
  - Main thread signals all threads not blocked by condition variables



Blocking on Condition Variables

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

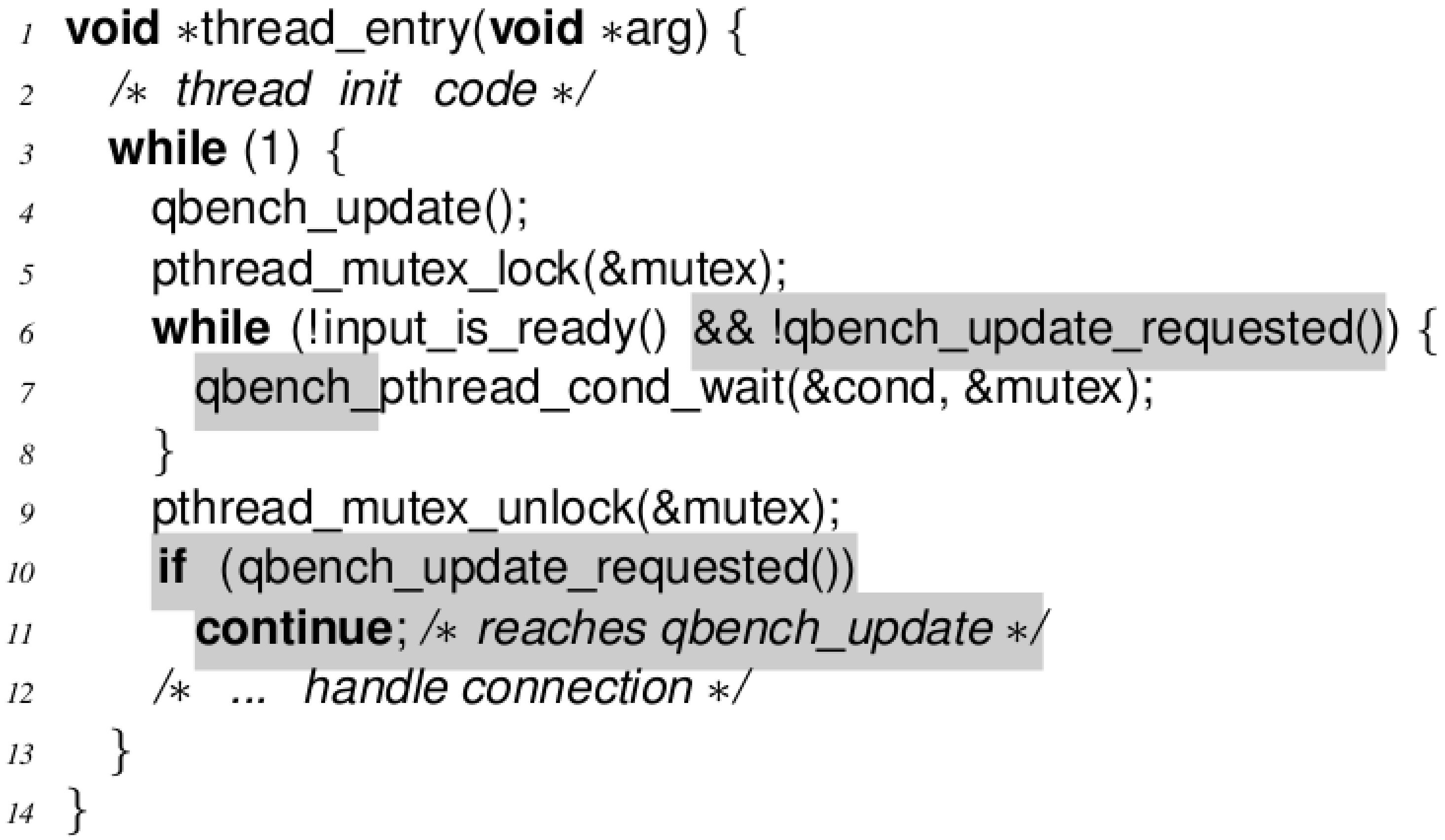


```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update();
    pthread_mutex_lock(&mutex);
    while (!input_is_ready()){
        pthread_cond_wait(&cond,&mutex);
    }
    pthread_mutex_unlock(&mutex);


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

Blocking on Condition Variables

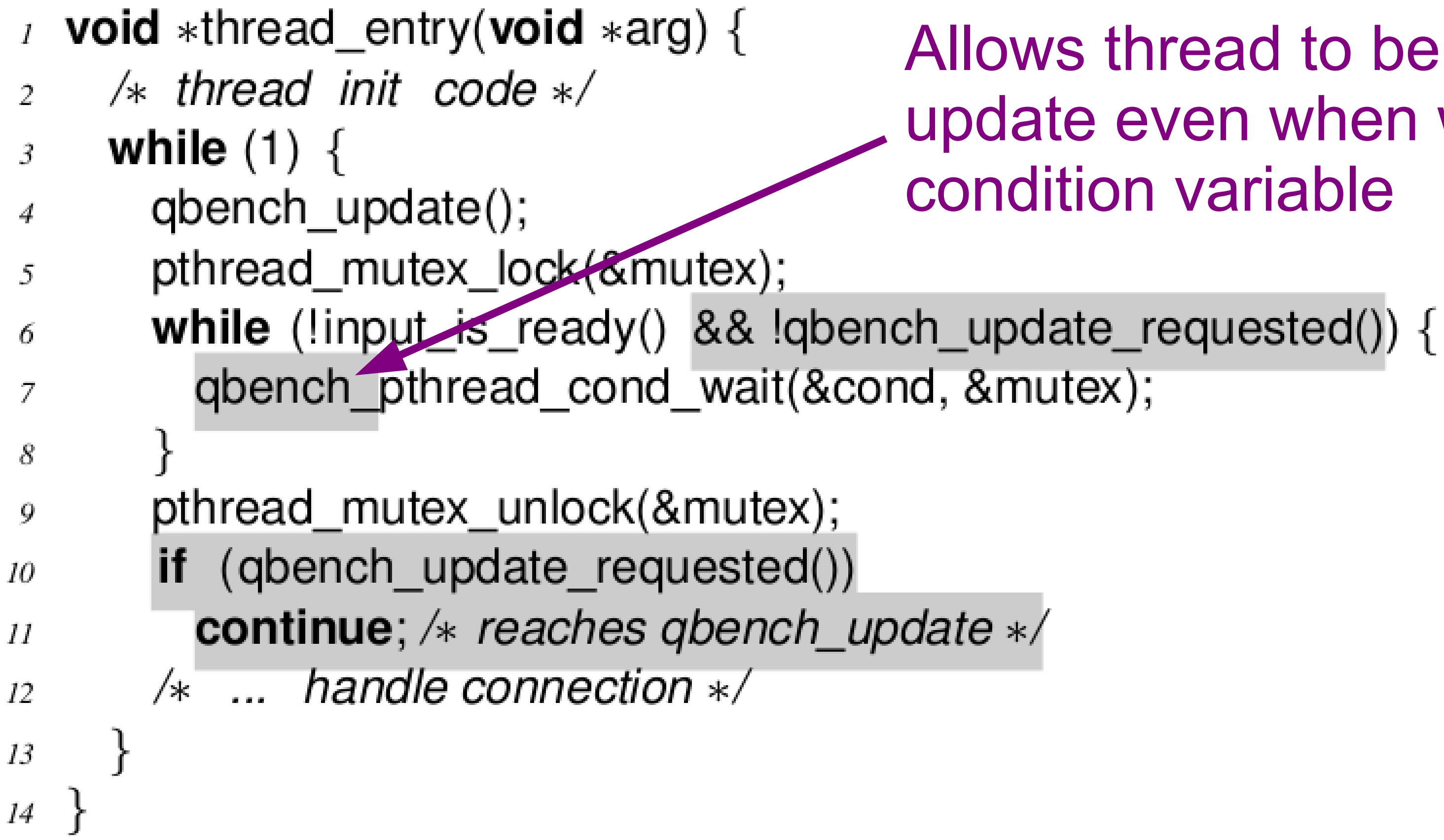
Programmers guard against spurious wake-ups by placing pthread_cond_wait in a loop



```
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

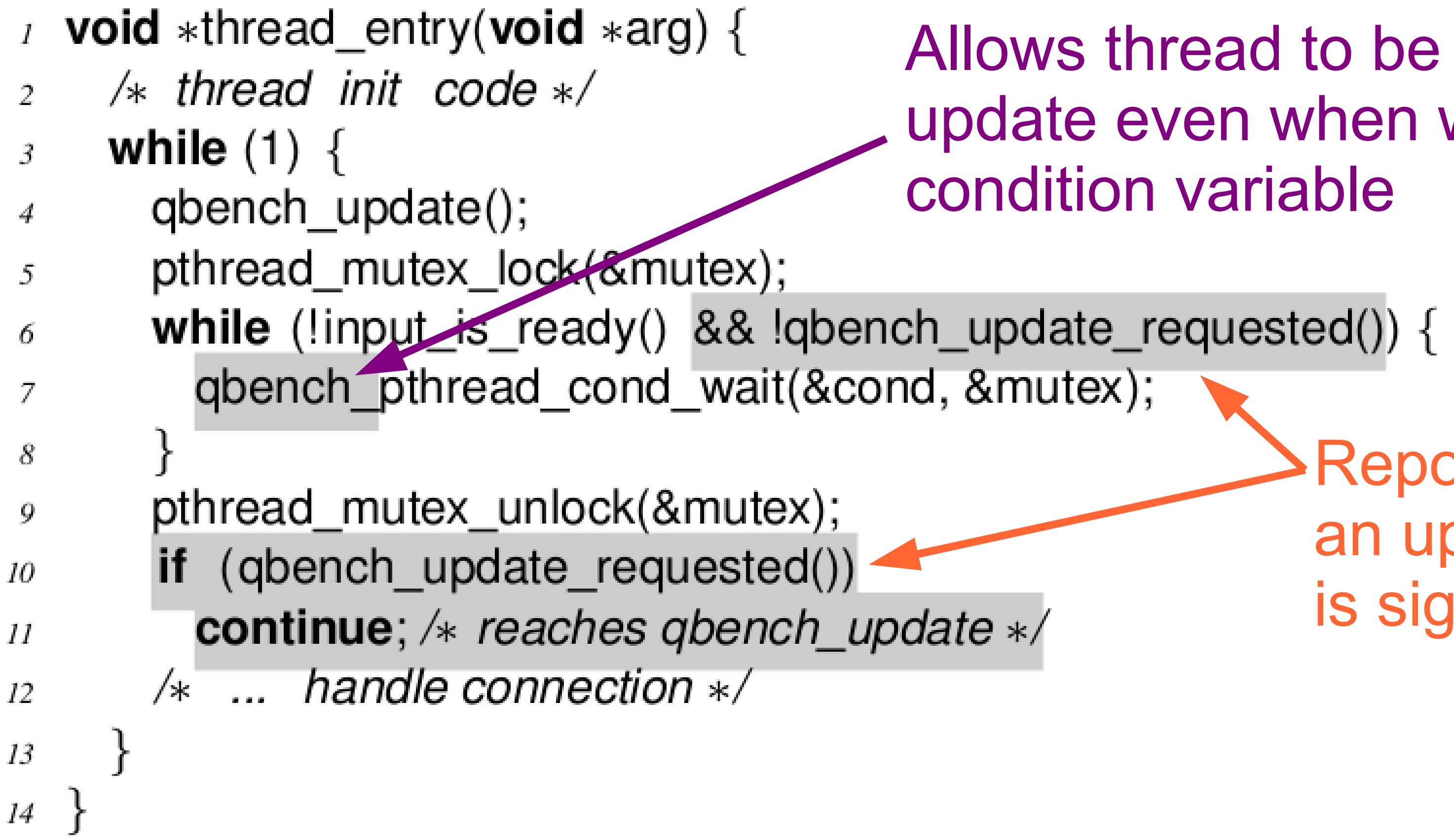


```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update();
    pthread_mutex_lock(&mutex);
    while (!input_is_ready() && !qbench_update_requested()) {
      qbench_pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
    if (qbench_update_requested())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

Allows thread to be signaled for update even when waiting on a condition variable

Blocking on Condition Variables

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



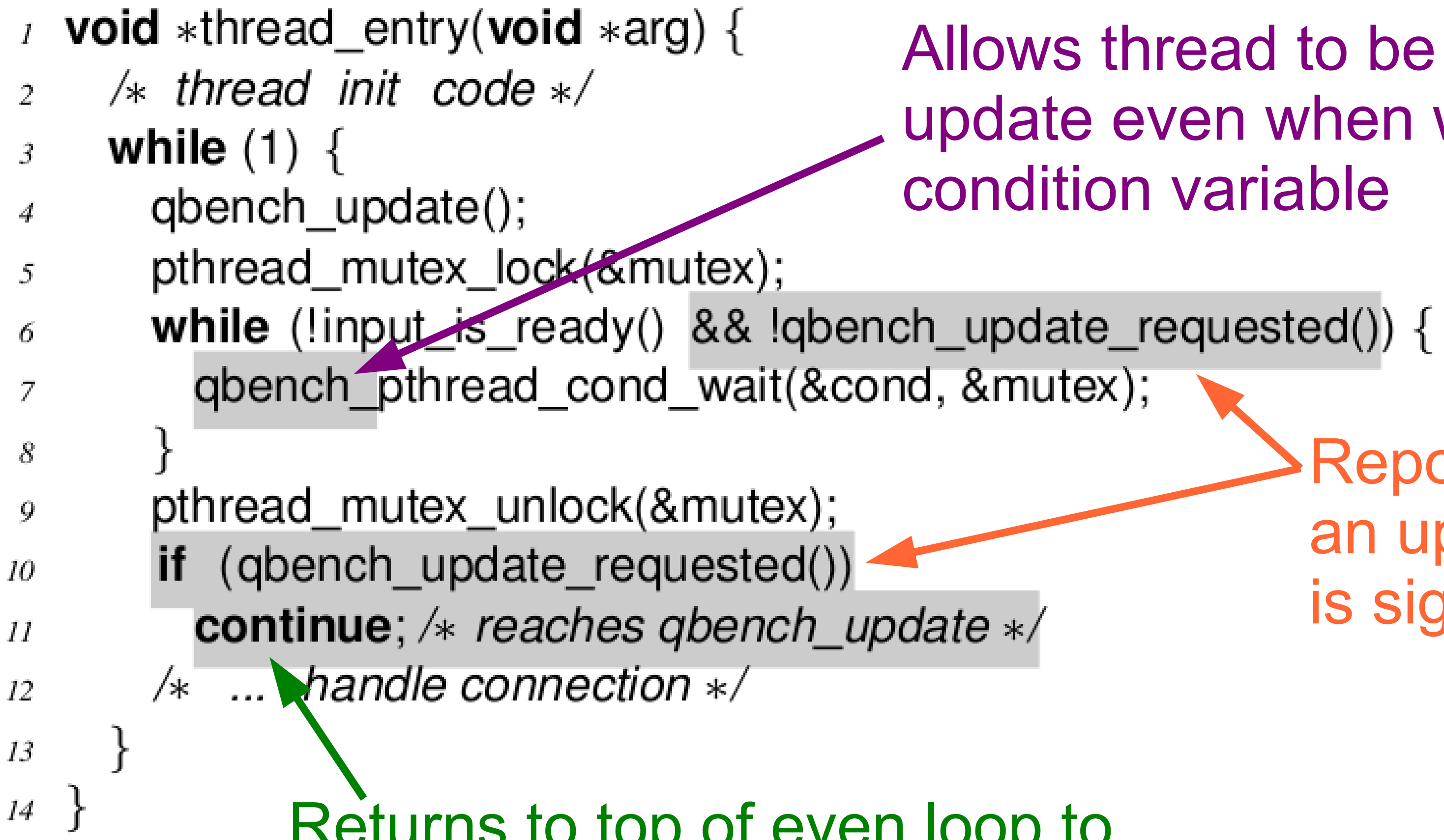
```
void *thread_entry(void *arg) {
  /* thread init code */
  while (1) {
    qbench_update();
    pthread_mutex_lock(&mutex);
    while (!input_is_ready() && !qbench_update_requested()) {
      qbench_pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
    if (qbench_update_requested())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

Allows thread to be signaled for update even when waiting on a condition variable

Reports true if an update request is signaled

Blocking on Condition Variables

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



```
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 */
  }
}
```

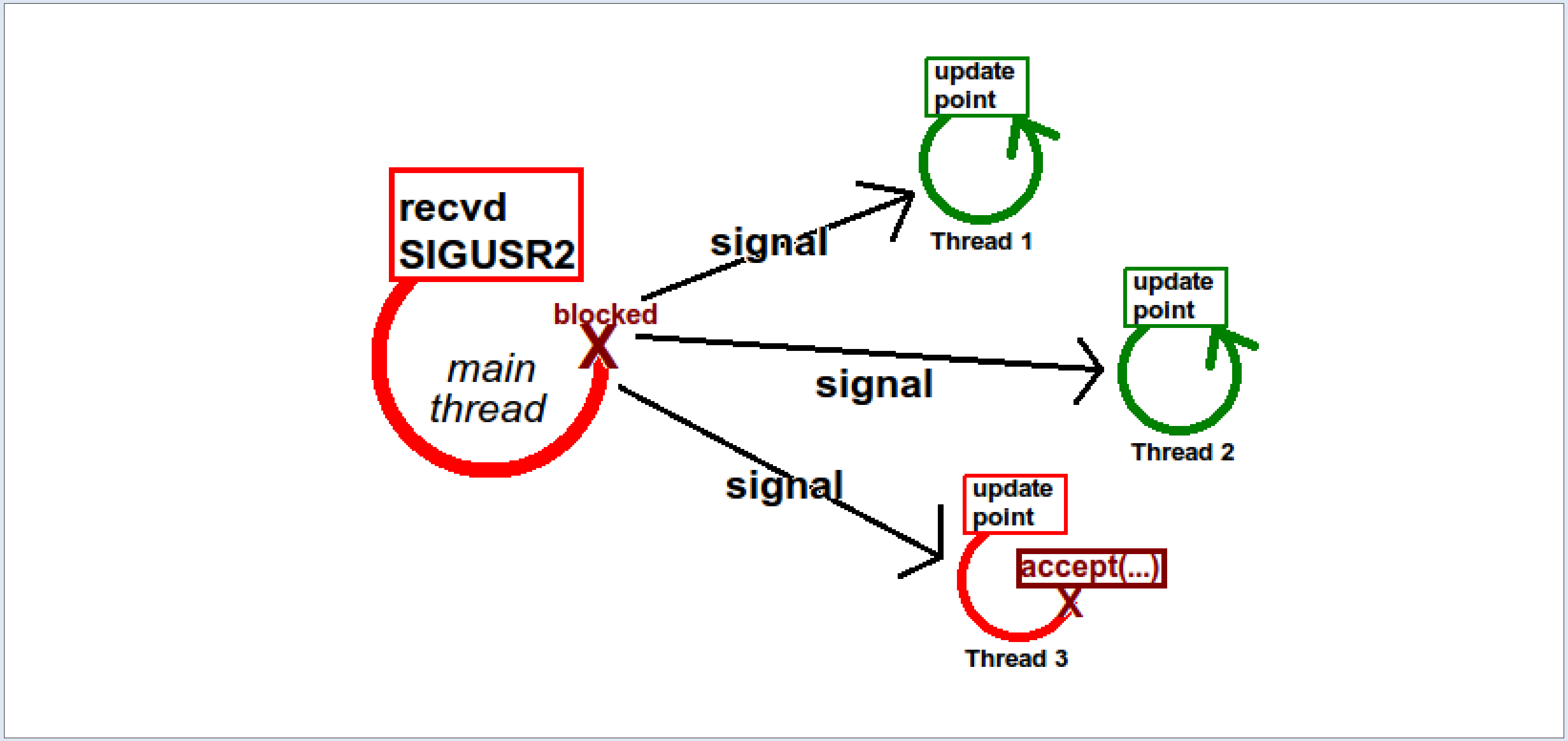
Allows thread to be signaled for update even when waiting on a condition variable

Reports true if an update request is signaled

Returns to top of even loop to immediately hit update point

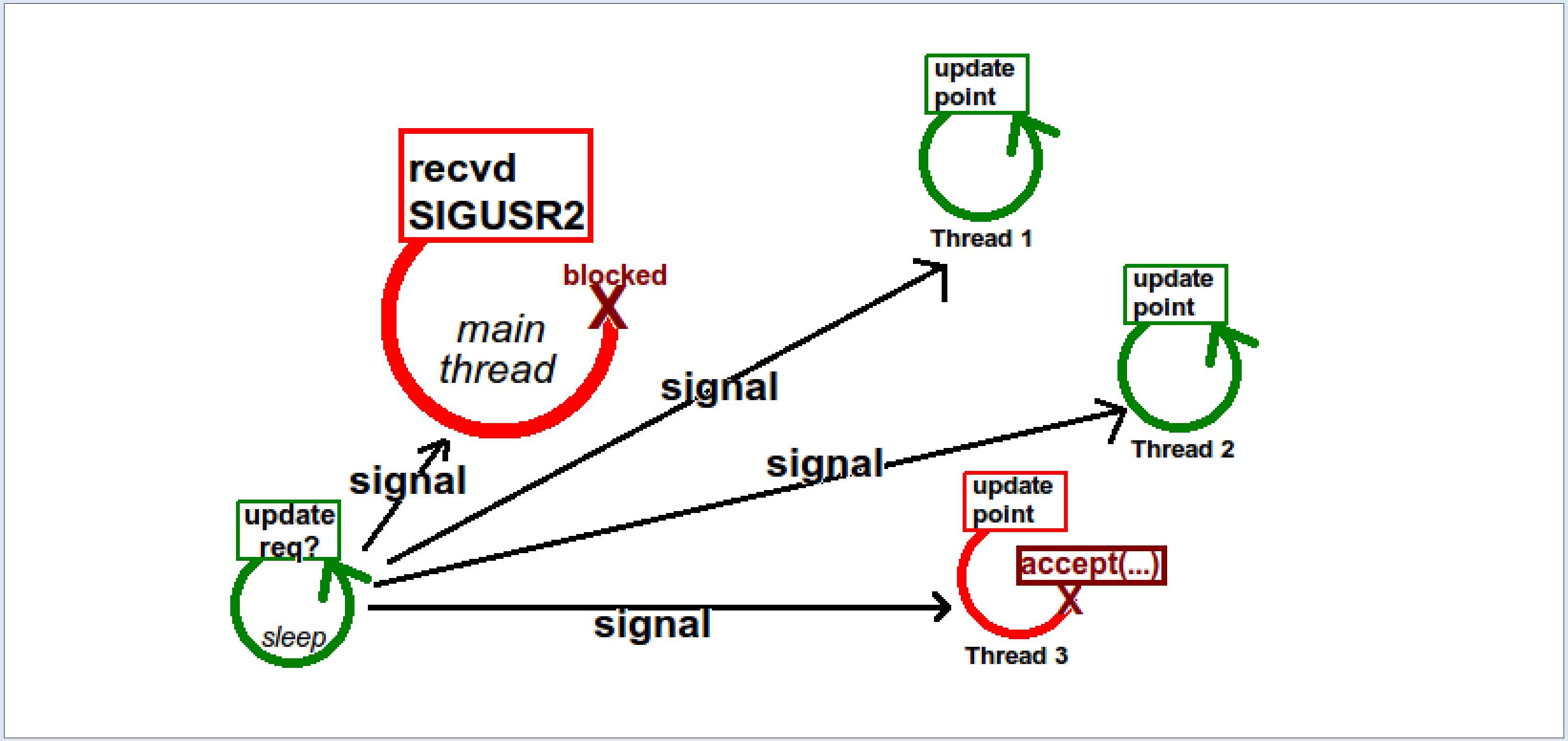
Waking a Blocked Thread

- Condition Variables: Another thread must be available to signal



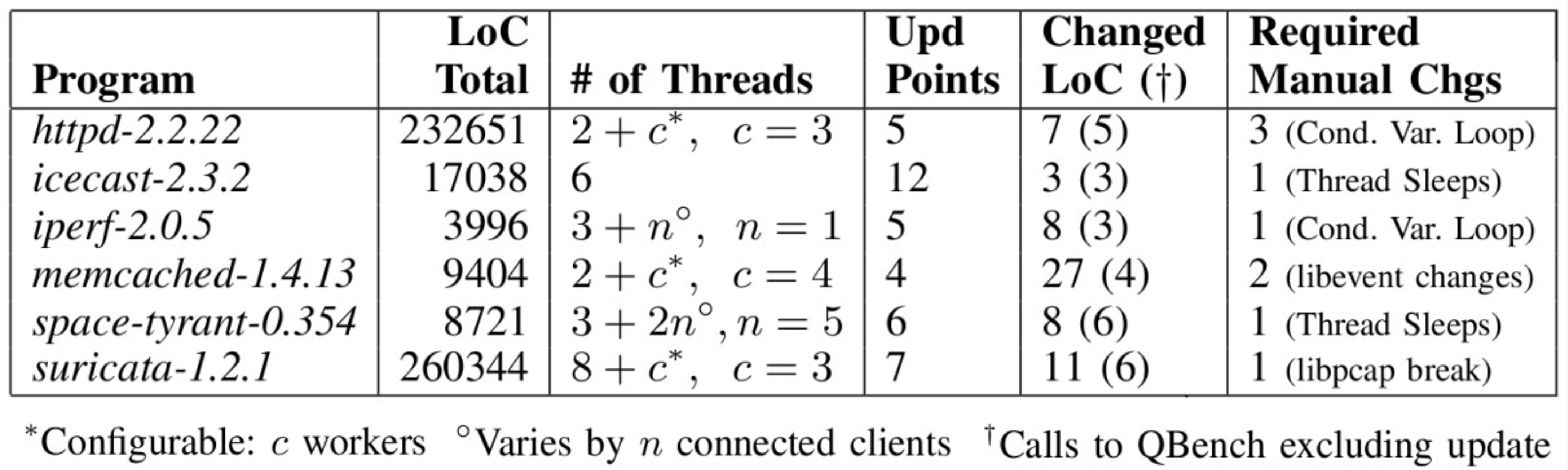
Waking a Blocked Thread

- 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.

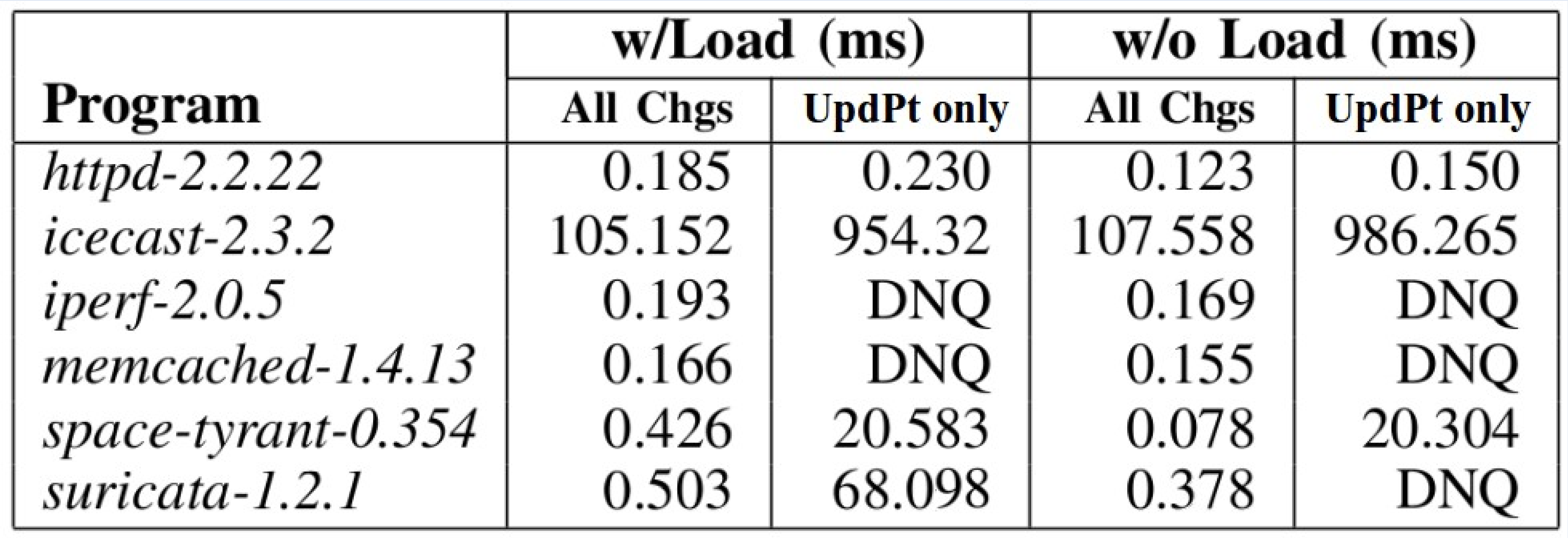


Program	LoC Total	# of Threads	Upd Points	Changed LoC (†)	Required Manual Chgs
httpd-2.2.22	232651	$2 + c^*, \quad c = 3$	5	7 (5)	3 (Cond. Var. Loop)
icecast-2.3.2	17038	6	12	3 (3)	1 (Thread Sleeps)
iperf-2.0.5	3996	$3 + n^\circ, \quad n = 1$	5	8 (3)	1 (Cond. Var. Loop)
memcached-1.4.13	9404	$2 + c^*,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

*Configurable: c workers °Varies by n connected clients †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

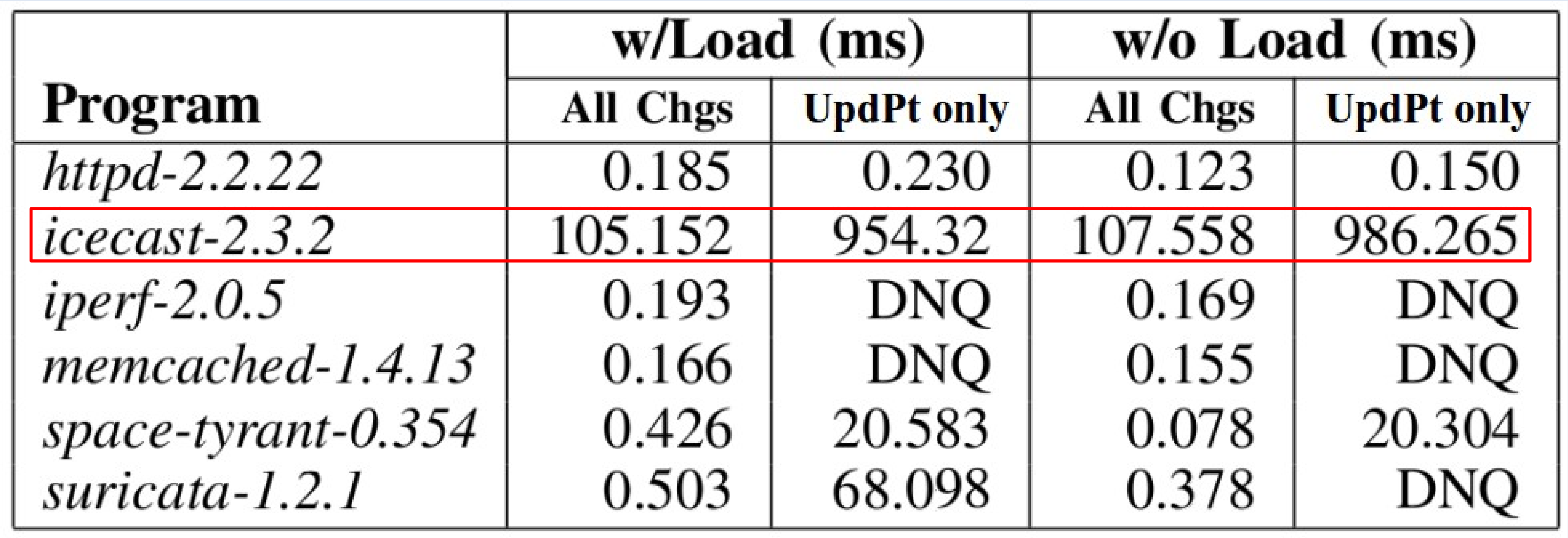


Program	w/Load (ms)		w/o Load (ms)	
	All Chgs	UpdPt only	All Chgs	UpdPt only
httpd-2.2.22	0.185	0.230	0.123	0.150
icecast-2.3.2	105.152	954.32	107.558	986.265
iperf-2.0.5	0.193	DNQ	0.169	DNQ
memcached-1.4.13	0.166	DNQ	0.155	DNQ
space-tyrant-0.354	0.426	20.583	0.078	20.304
suricata-1.2.1	0.503	68.098	0.378	DNQ

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

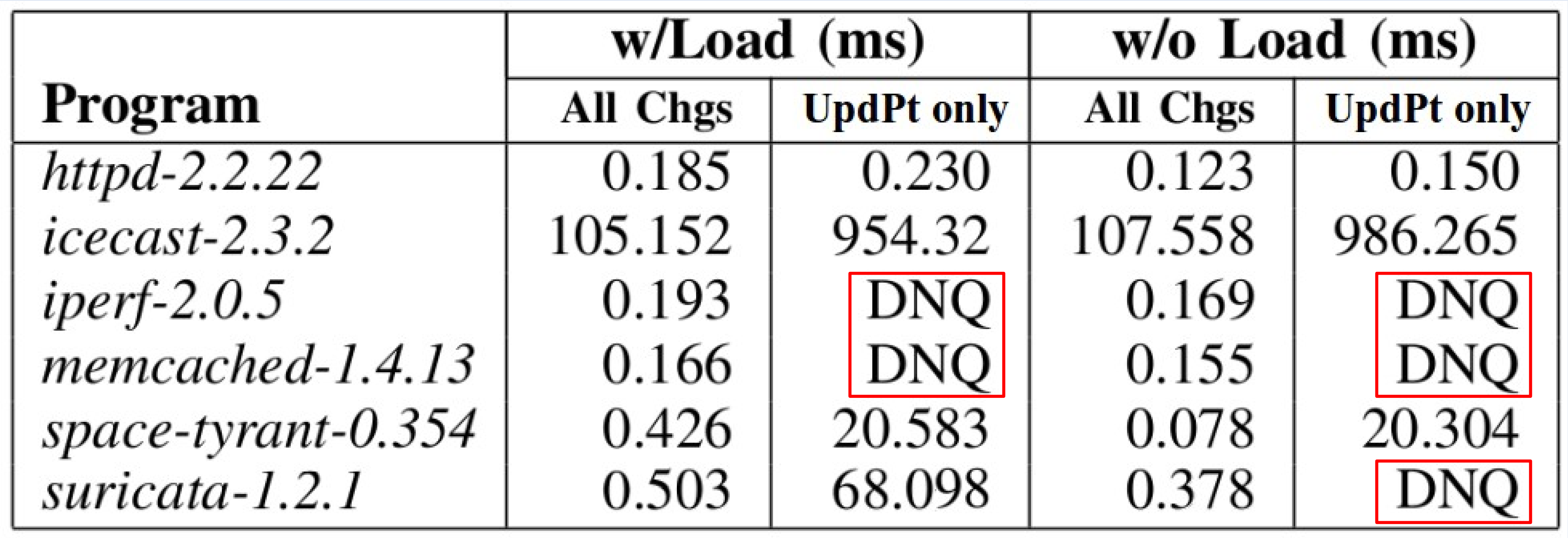


Program	w/Load (ms)		w/o Load (ms)	
	All Chgs	UpdPt only	All Chgs	UpdPt only
httpd-2.2.22	0.185	0.230	0.123	0.150
icecast-2.3.2	105.152	954.32	107.558	986.265
iperf-2.0.5	0.193	DNQ	0.169	DNQ
memcached-1.4.13	0.166	DNQ	0.155	DNQ
space-tyrant-0.354	0.426	20.583	0.078	20.304
suricata-1.2.1	0.503	68.098	0.378	DNQ

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



Program	w/Load (ms)		w/o Load (ms)	
	All Chgs	**UpdPt only**	**All Chgs**	**UpdPt only**
httpd-2.2.22	0.185	0.230	0.123	0.150
icecast-2.3.2	105.152	954.32	107.558	986.265
iperf-2.0.5	0.193	DNQ	0.169	DNQ
memcached-1.4.13	0.166	DNQ	0.155	DNQ
space-tyrant-0.354	0.426	20.583	0.078	20.304
suricata-1.2.1	0.503	68.098	0.378	DNQ

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