Semantics-Aware Synchronization Determinism and Beyond

Qi Zhao

North Carolina State University

The Non-Deterministic Nature of Multithreaded Programs

- Hard to get right
  - A large number of possible schedules
  - Different executions can get different results
  - Difficult to write, test, debug, and verify

- Suffer from concurrency issues

Existing techniques

Bug Detection and Fixing

Automatic Bug Detection

Automatic Bug Fixing

Reduce or Eliminate Non-Determinism

Record and Replay

Deterministic Multithreading (DMT)

Deterministic Multithreading (DMT)

- Ensures that executions of the same program with the same input will have the same schedule
- Two levels of determinism guarantees
  - Weak determinism:
    - Only enforces synchronization determinism
  - Strong determinism:
    - Further enforces memory-access determinism

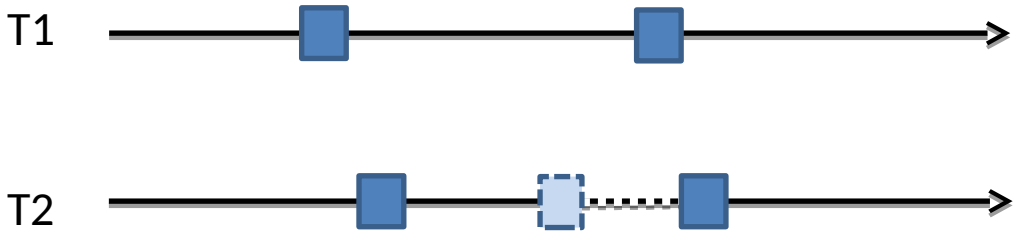
Synchronization Determinism

- Currently not the research focus of DMT systems

- Why synchronization determinism?
  - Reduces the number of remaining races
  - Feasible to check all remaining schedules
  - Serves as a basis for strong determinism

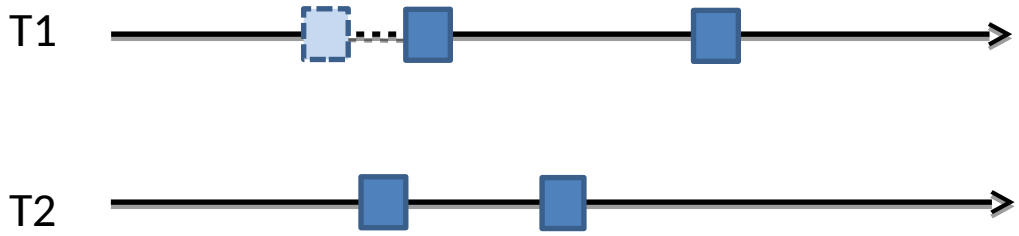
Synchronization Determinism

- Turn-based mechanism



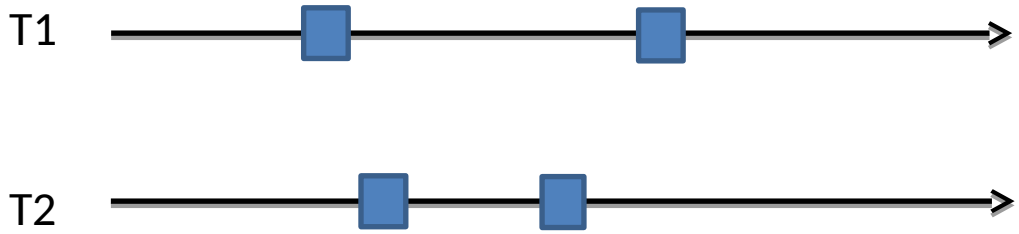
Synchronization Determinism

- Turn-based mechanism
- Different scheduling policies



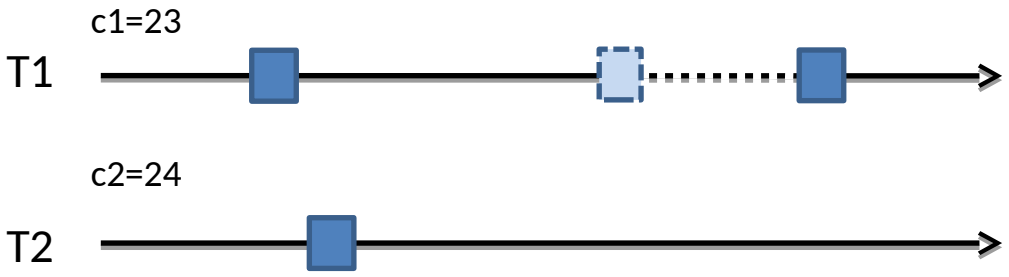
Synchronization Determinism

- Turn-based mechanism
- Different scheduling policies

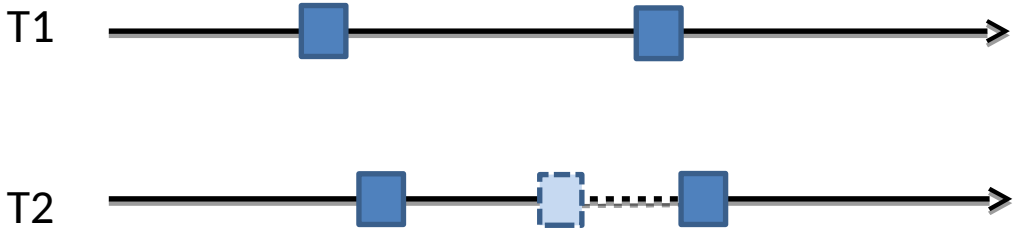


Two Types of Scheduling Policies

Logic clock



Round robin

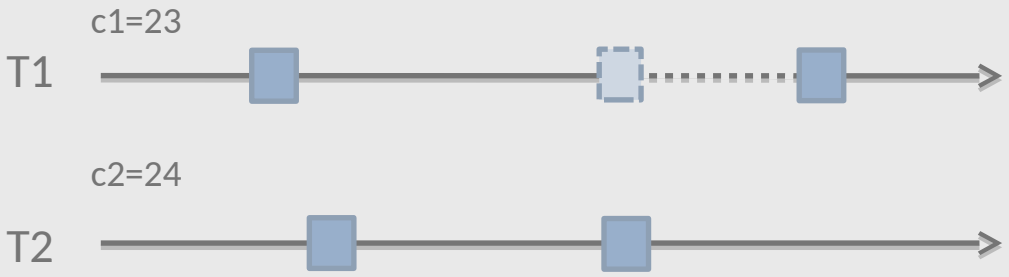


Metrics of Scheduling Policies

- Performance
  - What is the overhead caused by DMT systems
- Stability
  - How frequent will a scheduling policy reuse schedule
- Fairness
- Deadlock free

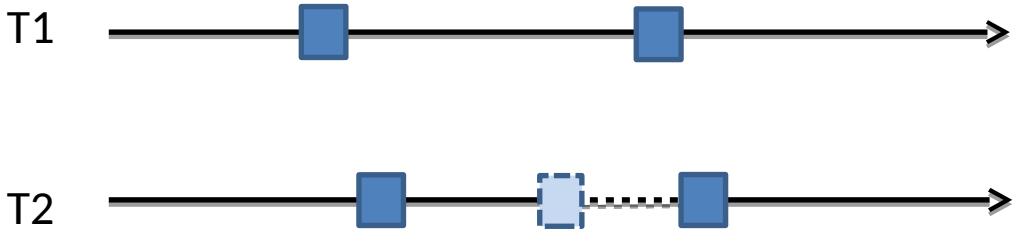
Two Types of Scheduling Policies

Logic clock



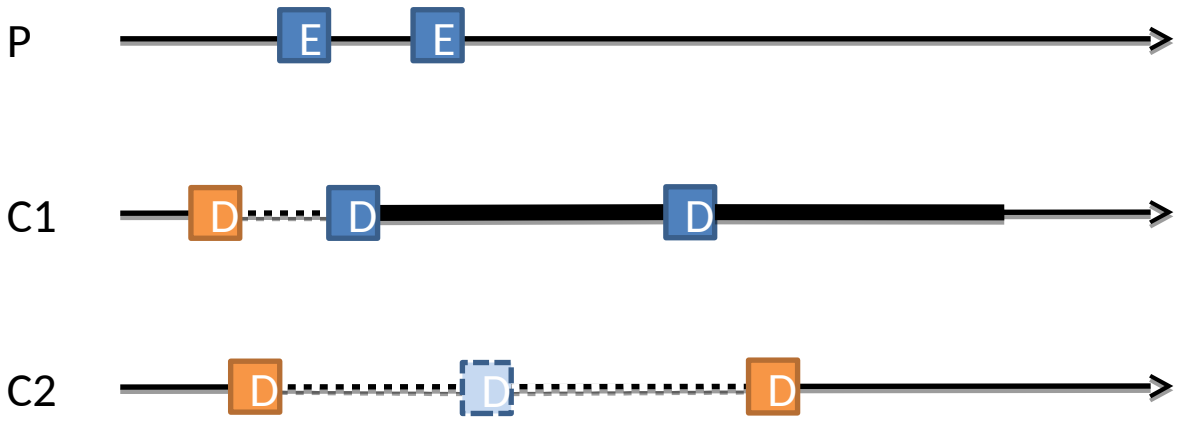
Performance
Stability

Round robin



Stability
Performance

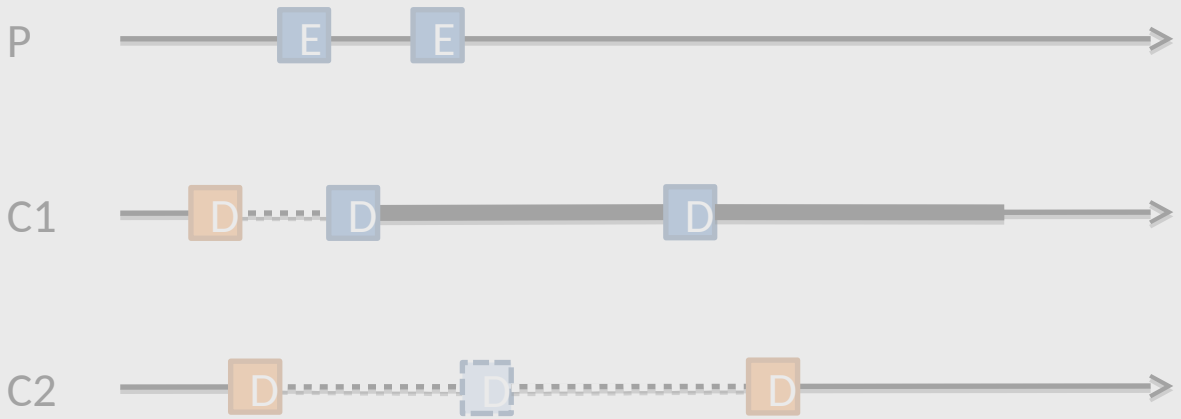
Round-robin Example




```
void producer(void *arg){
  ...
  ...
  for (i=0; i<nblocks; ++i){
    char *block = read_block(i);
    enqueue(q, block);
  }
  ...
}
```

```
void *consumer(void *arg) {
  ...
  while(1) {
    char *block = dequeue(q);
    ...
    ...
    compress(block);
  }
}
```

Parrot's Soft Barrier




```
void producer(void *arg){
  soba_init(number_of_consumers)
  ...
  for (i=0; i<nblocks; ++i){
    char *block = read_block(i);
    enqueue(q, block);
  }
  ...
}
```

```
void *consumer(void *arg) {
  ...
  while(1) {
    char *block = dequeue(q);
    ...
    soba_wait()
    compress(block);
  }
}
```

Parrot's Soft Barrier

Problems with Soft Barriers

- Extra manual effort
- Can sometimes be difficult to place
- Need to be maintained

```
void producer(void *arg){
  soba_init(number_of_consumers)
  ...
  for (i=0; i<nblocks; ++i){
    char *block = read_block(i);
    enqueue(q, block);
  }
  ...
}
```

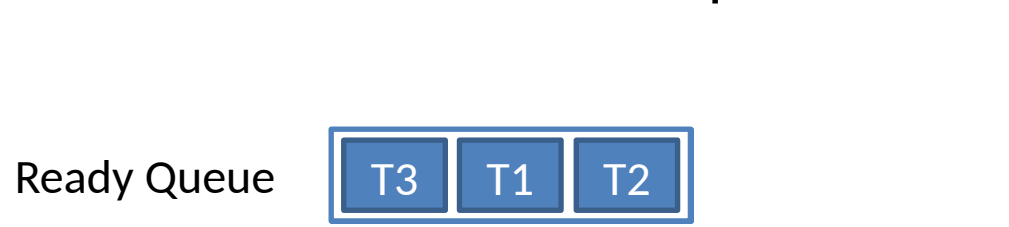
```
void *consumer(void *arg) {
  ...
  while(1) {
    char *block = dequeue(q);
    ...
    soba_wait()
    compress(block);
  }
}
```

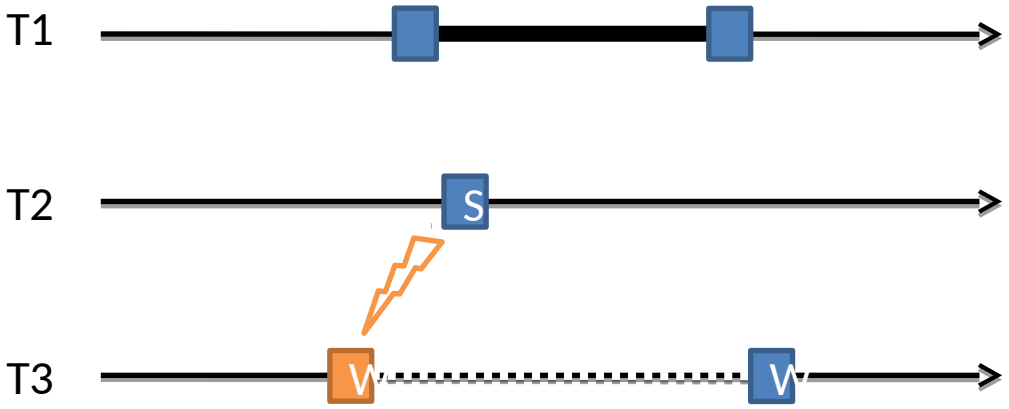
QiThread

- BoostBlocked
- CreateAll
- CSWhole
- WakeAMAP
- BranchedWake
- Our insight: By leveraging synchronization semantics, we can not only remove the need of manual annotations, but also achieve good performance and schedule stability

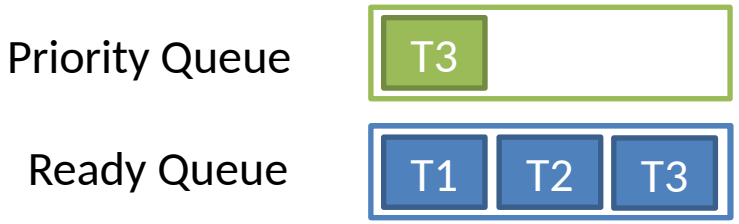
BoostBlocked

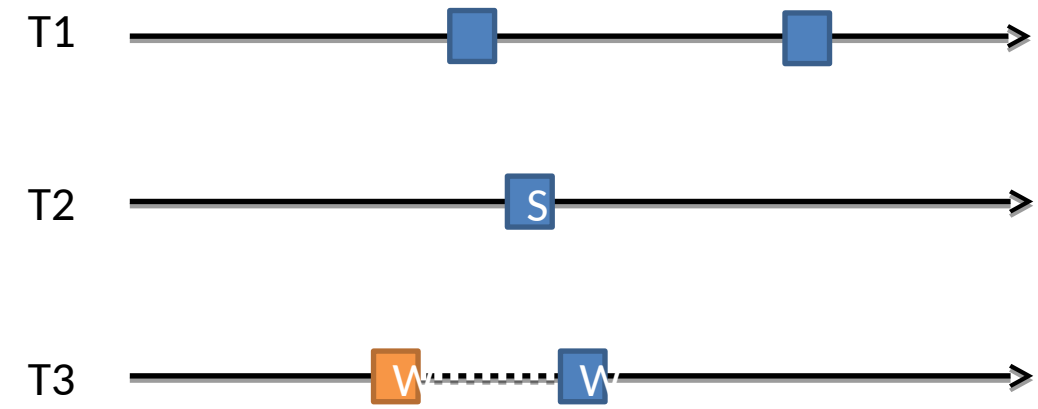
Prioritize threads woken up from the blocked state





Round-robin





BoostBlocked Applied

CreateAll

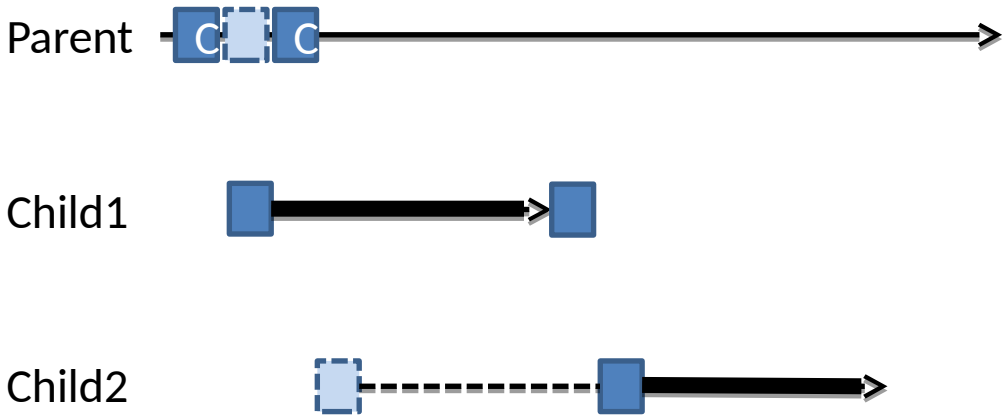
Schedule multiple pthread_create together

Parrent thread

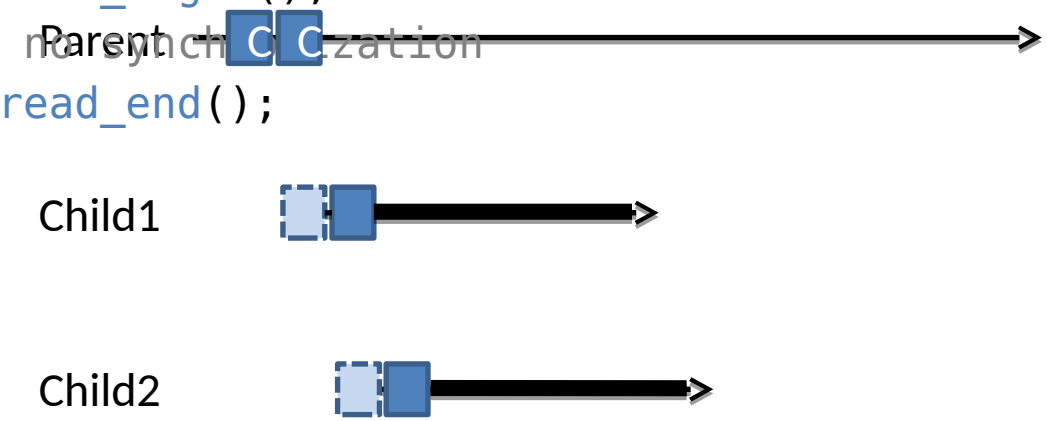
```
void parent_main(…){
  for (i=0;i<nThreads;i++
    pthread_create(…,…,
        child_main,…);
}
```

Child thread

```
void child_main(…){
  thread_begin();
  // no synchronization
  thread_end();
}
```



Round-robin

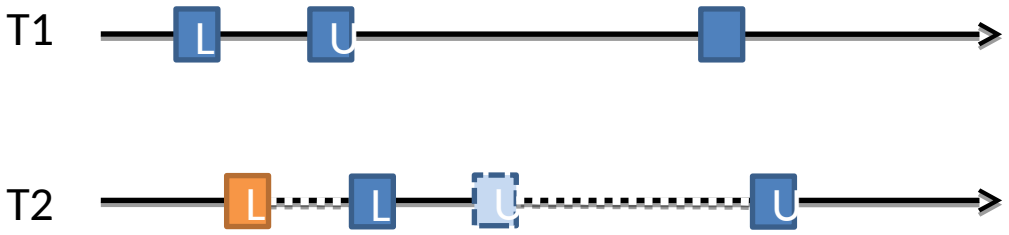


CreateAll Applied

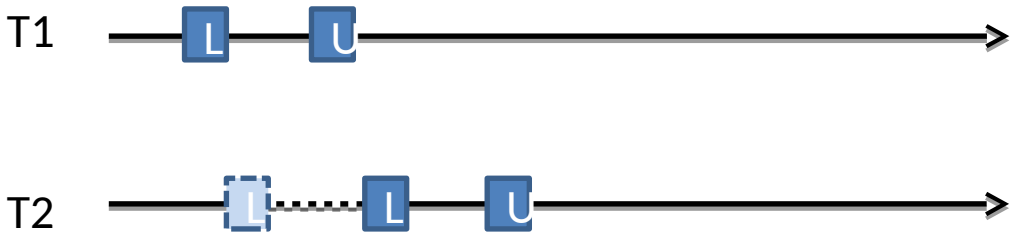
CSWhole

Schedule a critical section as a whole

```
…
pthread_mutex_lock(&m);
counter++;
pthread_mutex_unlock(&m);
…
```



Round-robin

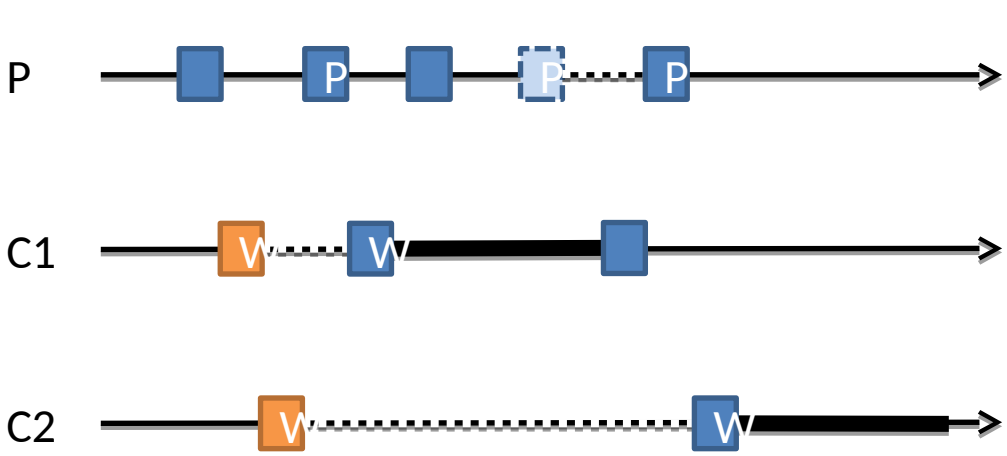


CSWhole applied

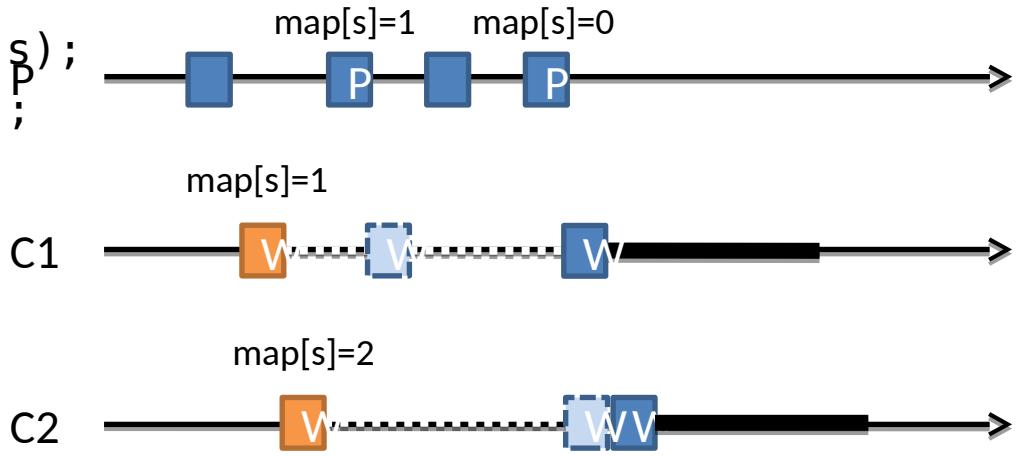
WakeAMAP

Wake up as many threads as possible

```
void producer(void *arg){
  while(cond1) {
    sync();
    sem_post(s);
    ...
  }
}
void consumer(void *arg) {
  ...
  sem_wait(s);
  do_work();
  ...
}
```



Round-robin



WakeAMAP applied

BranchedWake

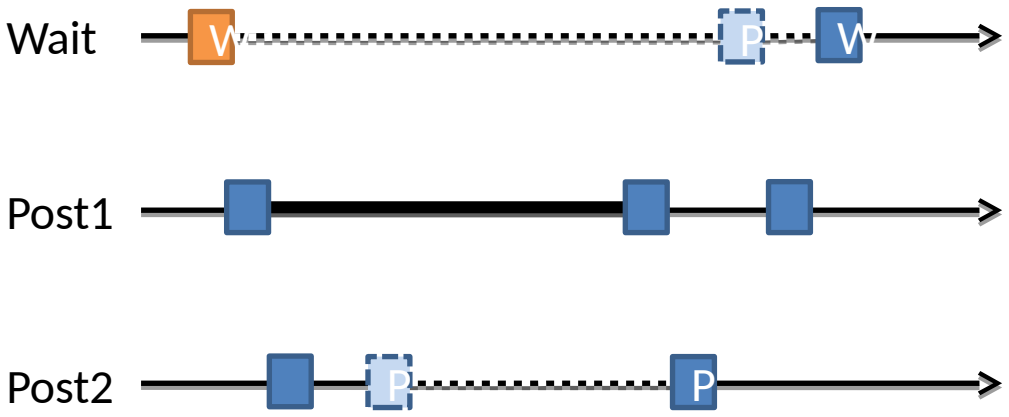
Handle threads that have wake-up synchronization in a branch

Multiple Post threads:

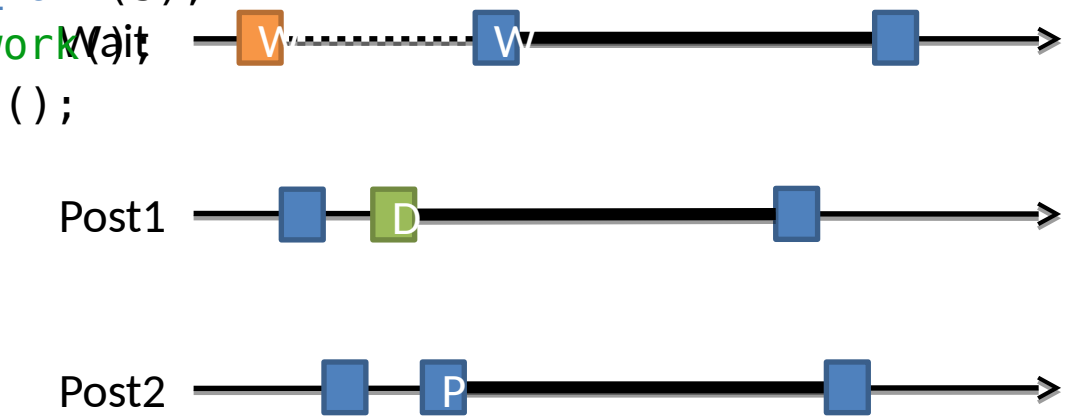
```
sync();
if (isLastPostThread)
// The last thread posts
  sem_post(s);
do_work();
sync();
```

Waiting thread:

```
sem_wait(s);
do_work();
sync();
```



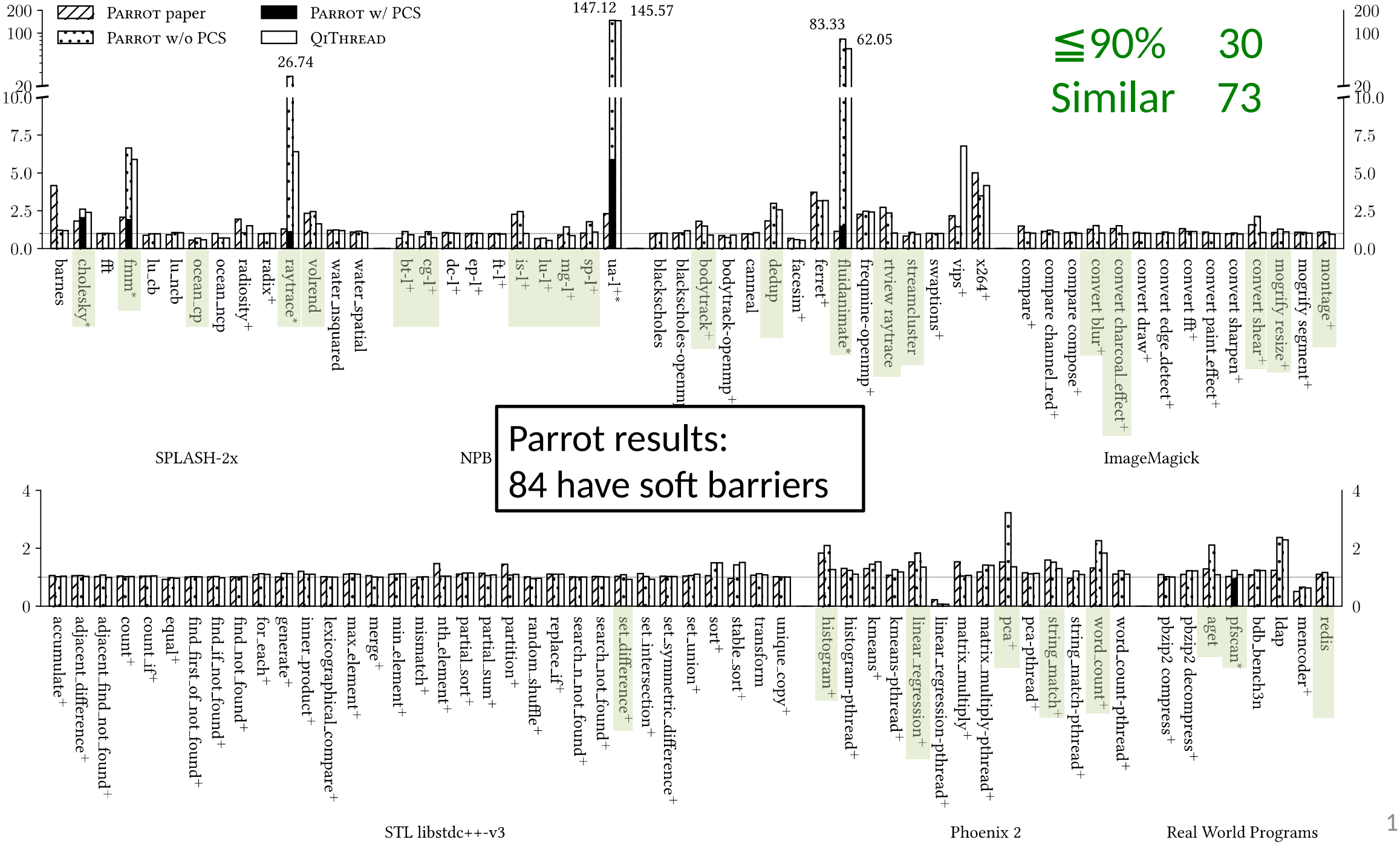
Round-robin

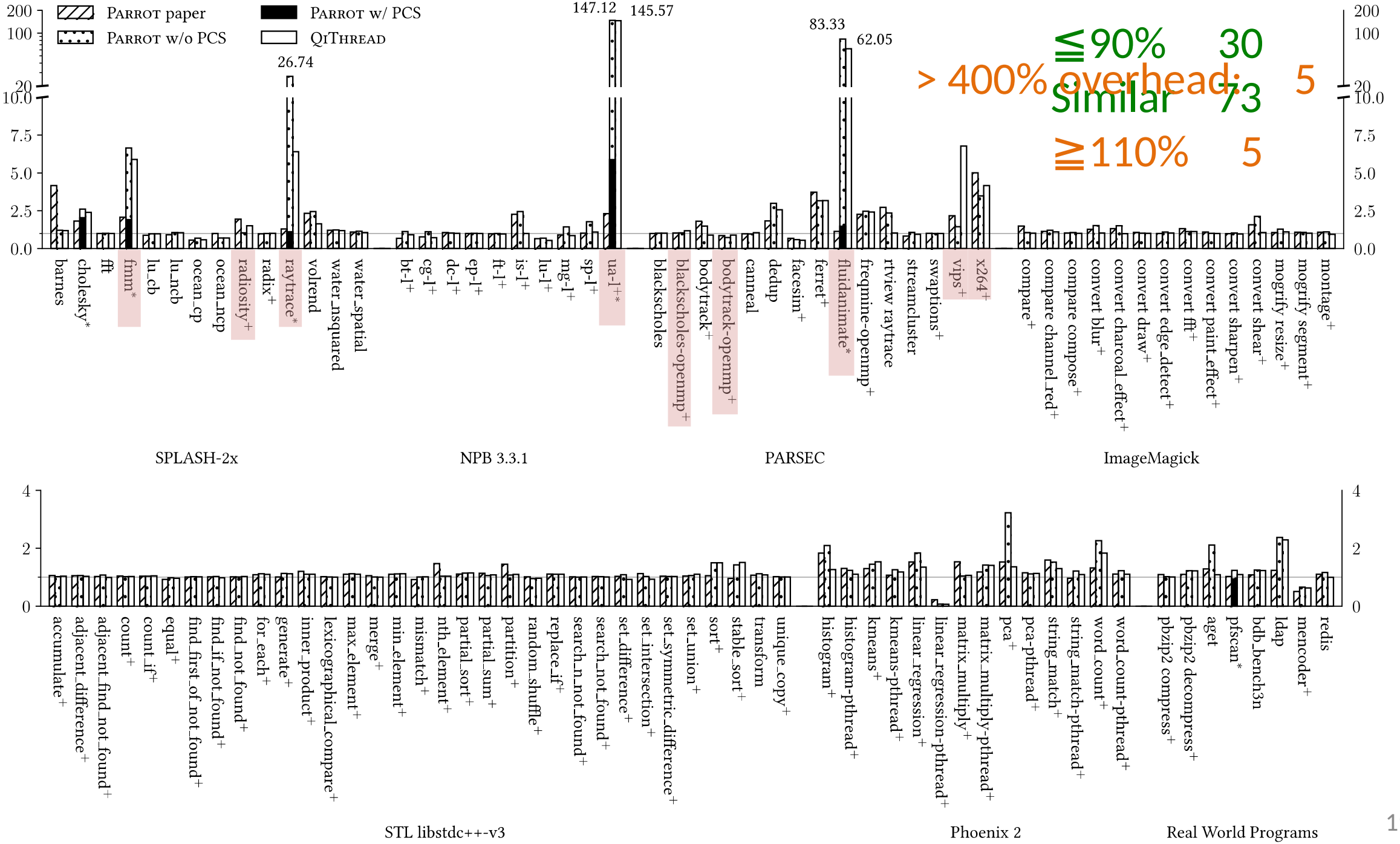


BranchedWake & BoostBlocked

Evaluation

- Performance: all 108 programs
  - Used the same thread count as in Parrot
  - Parrot w/ soft barrier hint and QiThread w/ all policies enabled


Parrot paper
Parrot w/o PCS
Parrot w/ PCS
QtThread
147.12
145.57
83.33
62.05
26.74
≦90% 30
Similar 73
Parrot results:
84 have soft barriers
SPLASH-2x
NPB
ImageMagick
STL libstdc++-v3
Phoenix 2
Real World Programs


Parrot paper
Parrot w/o PCS
Parrot w/ PCS
QtThread
147.12
145.57
83.33
62.05
26.74
≦90% 30
> 400% overhead: 5
Similar 73
≧110% 5
SPLASH-2x
NPB 3.3.1
PARSEC
ImageMagick
STL libstdc++-v3
Phoenix 2
Real World Programs

Future Work

- High-level thread role analysis
  - Analyze synchronization relationship and data dependencies
  - Model thread role: producer, consumer, pipeline, etc.
  - Prioritize threads which produce data that other threads depends on
  - Instrument role changing annotations

Summary

Thank you! ☺

Questions ?

- QiThread
  - Five semantics-aware policies on top of round-robin
  - No manual performance hints needed
  - Widely effective
- Efficient and stable synchronization determinism can be achieved by leveraging synchronization semantics
- Future work: scheduling with the understanding of high-level thread roles