Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading

Junfeng Yang
http://www.cs.columbia.edu/~junfeng
Joint work w/ my brilliant students
Heming Cui, Jingyue Wu, Yang Tang, Gang Hu
Columbia University

One-slide overview

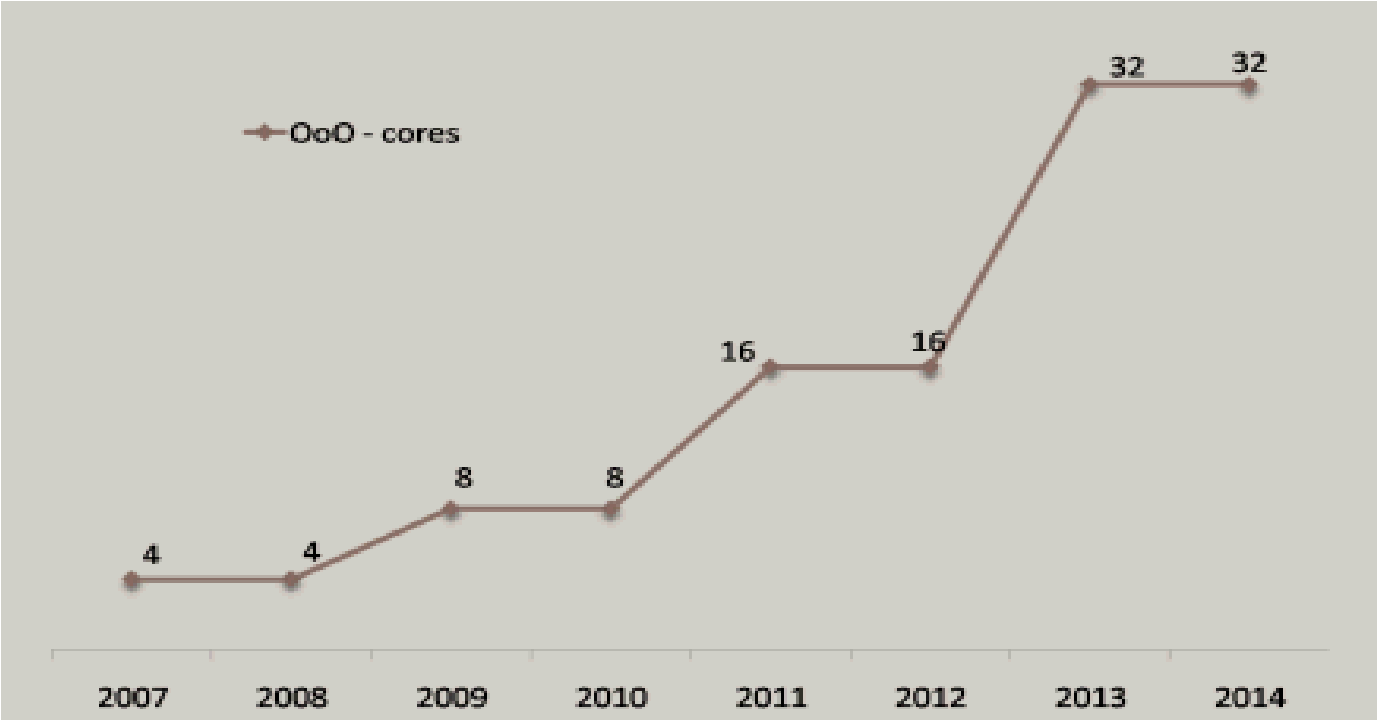
- Despite major advances in tools, multithreading remains hard to get right

- Why? ~~Nondeterminism~~ too many thread interleavings, or *schedules*

- *Stable Multithreading (StableMT)*: a radical approach to reducing the set of schedules for reliability with low overhead [Tern OSDI 10] [Peregrine SOSP 11] [Specialization PLDI 12] [Parrot SOSP 13] [HotPar 13] [CACM 14]

Background and motivation

Multithreaded programs: pervasive and critical



http://www.drdobbs.com/parallel/design-for-manycore-systems/219200099

Multithreaded programs: pervasive and critical

But, extremely hard to get right

Getty

But, extremely hard to get right

- Plagued with concurrency bugs [Lu ASPLOS 09]
  - Data races, atomicity violations, order violations, deadlocks, etc

But, extremely hard to get right

- Plagued with concurrency bugs [Lu ASPLOS 09]
  - Data races, atomicity violations, order violations, deadlocks, etc
- Concurrency bugs: bad
  - Have taken lives in the Therac 25 incidents and caused the 2003 Northeast blackout
  - May be exploited by attackers to violate confidentiality, integrity, and availability of critical systems [Hotpar 12]

Concurrency bug example

```
    Thread 0               Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)
                    mutex_lock(M)
                    free(obj)
                    mutex_unlock(M)
```

Apache Bug #21287 (simplified)

Concurrency bug example

```
    Thread 0              Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)
                        mutex_lock(M)
                        free(obj)
                        mutex_unlock(M)
```

```
    Thread 0              Thread 1
                        mutex_lock(M)
                        free(obj)
                        mutex_unlock(M)
mutex_lock(M)
*obj = …
mutex_unlock(M)
```

Apache Bug #21287 (simplified)

Concurrency bug example

```
    Thread 0              Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)

                    mutex_lock(M)
                    free(obj)
                    mutex_unlock(M)
```

```
    Thread 0              Thread 1
                       mutex_lock(M)
                       free(obj)
                       mutex_unlock(M)

mutex_lock(M)
*obj = …
mutex_unlock(M)
```

Apache Bug #21287 (simplified)

- *Input*: everything a program reads from environment
  - E.g., main() arguments, data read from file or socket

Concurrency bug example

```
    Thread 0                 Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)
                          mutex_lock(M)
                          free(obj)
                          mutex_unlock(M)
```

```
    Thread 0                 Thread 1
                          mutex_lock(M)
                          free(obj)
                          mutex_unlock(M)
mutex_lock(M)
*obj = …
mutex_unlock(M)
```

Apache Bug #21287 (simplified)

- *Input*: everything a program reads from environment
  - E.g., main() arguments, data read from file or socket
- *Schedule*: sequence of communication operations
  - E.g., total order of synchronizations such as lock()/unlock()

Concurrency bug example

```
     Thread 0                Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)
                        mutex_lock(M)
                        free(obj)
                        mutex_unlock(M)
```

```
     Thread 0                Thread 1
                         mutex_lock(M)
                         free(obj)
                         mutex_unlock(M)
mutex_lock(M)
*obj = …
mutex_unlock(M)
```

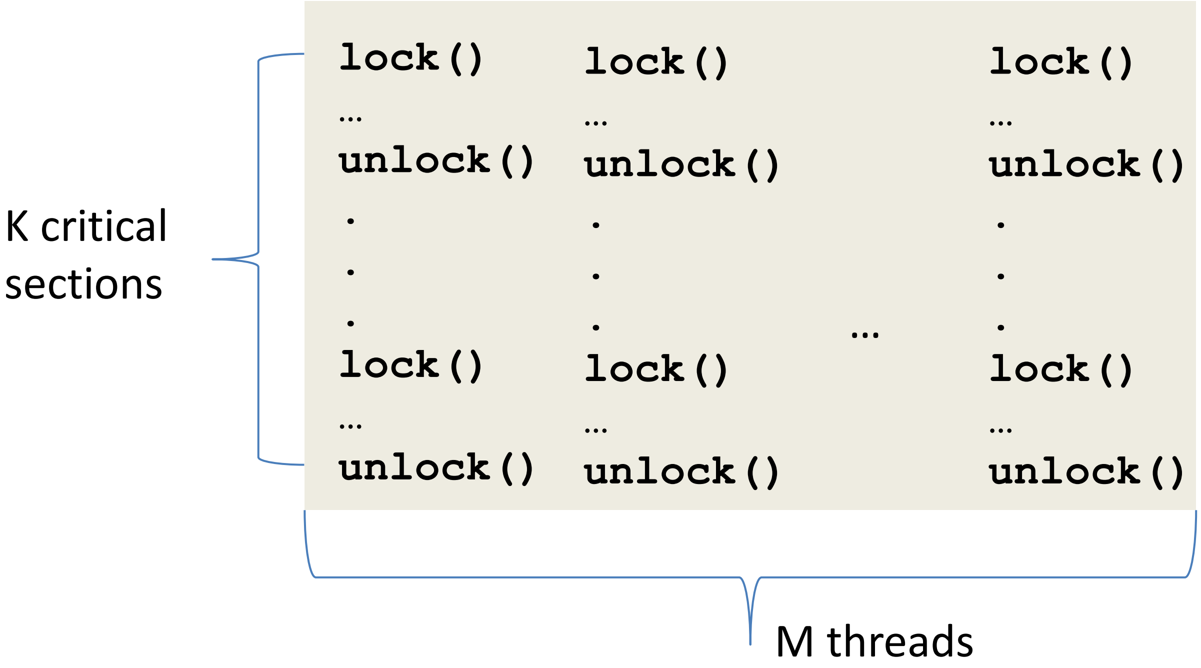
Apache Bug #21287 (simplified)

- *Input*: everything a program reads from environment
  - E.g., main() arguments, data read from file or socket
- *Schedule*: sequence of communication operations
  - E.g., total order of synchronizations such as lock()/unlock()
- *Buggy schedule*: schedule triggering concurrency bug

Advances in tools

- The pursuit of results: systems research focus shifted from speed to reliability around 2000
- More effective static analysis, model checking, symbolic execution, verification
  - E.g., vulgar version of model checking that enumerates through real executions for bugs [Verisoft POPL 97] [CMC OSDI 02] [FiSC OSDI 04] [eXplode OSDI 06] [MaceMC NSDI 07] [Chess ODSI 08] [MoDIST NSDI 09] [Inspect SPIN 09] [dBug SPIN 11]
- Unfortunately, concurrency/multithreading remains the bane of these tools

Why hard?



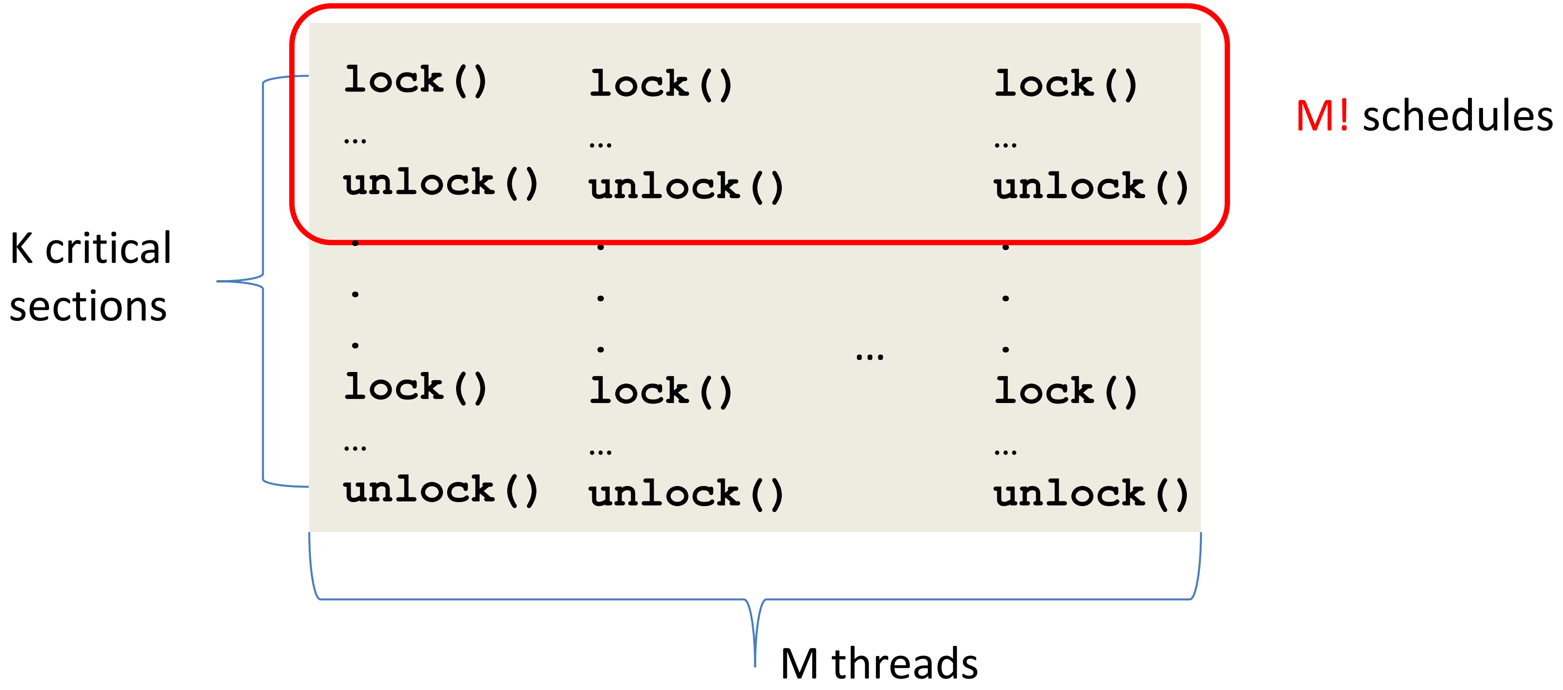
```
lock()      lock()                lock()
…           …                     …
unlock()    unlock()              unlock()
.           .                     .
.           .                     .
.           .             …       .
lock()      lock()                lock()
…           …                     …
unlock()    unlock()              unlock()
```

K critical sections

M threads

- Number of schedules: exponential in K, M
- Even more schedules aggregated over all inputs

Why hard?



```
lock()      lock()              lock()
…           …                   …
unlock()    unlock()            unlock()
.           .                   .
.           .                   .
.           .          …        .
lock()      lock()              lock()
…           …                   …
unlock()    unlock()            unlock()
```

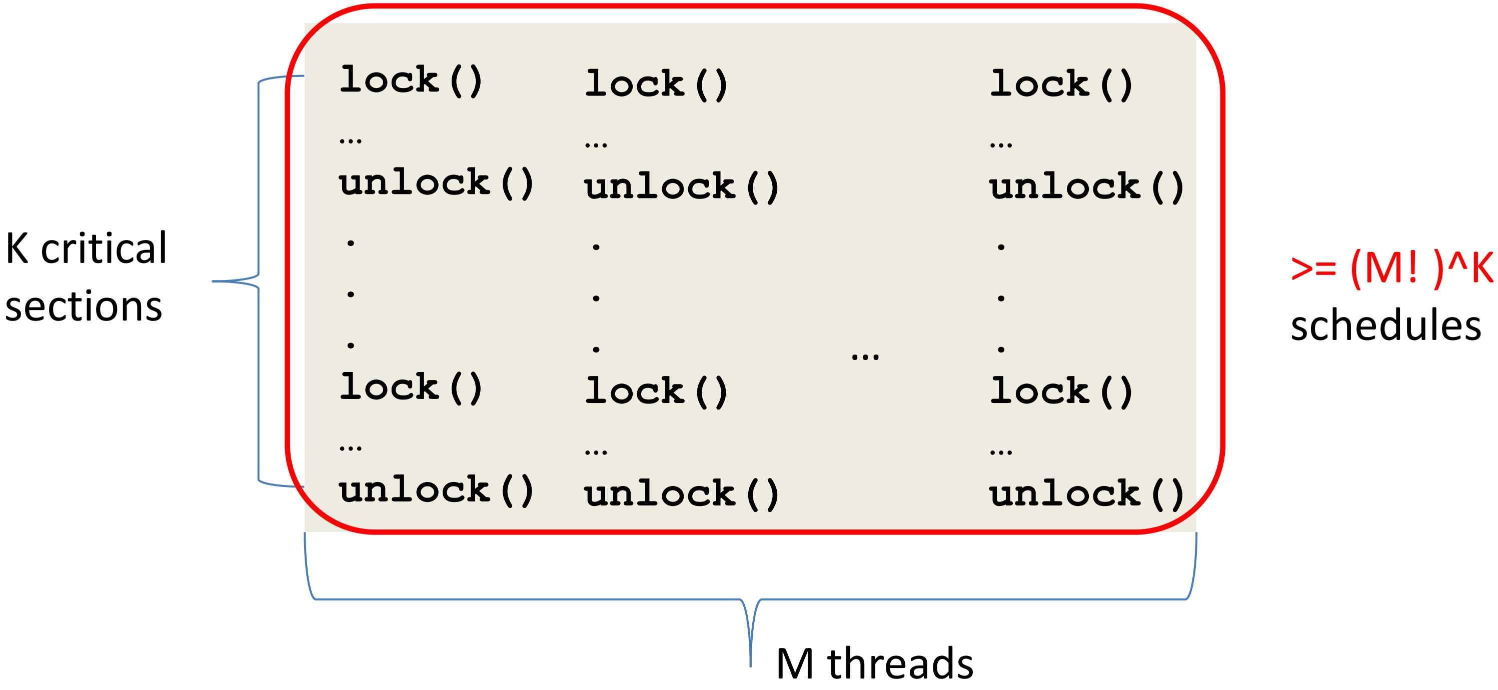
M! schedules

K critical sections

M threads

- Number of schedules: exponential in K, M
- Even more schedules aggregated over all inputs

Why hard?



```
lock()      lock()              lock()
…           …                   …
unlock()    unlock()            unlock()
.           .                   .
.           .                   .
.           .          …        .
lock()      lock()              lock()
…           …                   …
unlock()    unlock()            unlock()
```

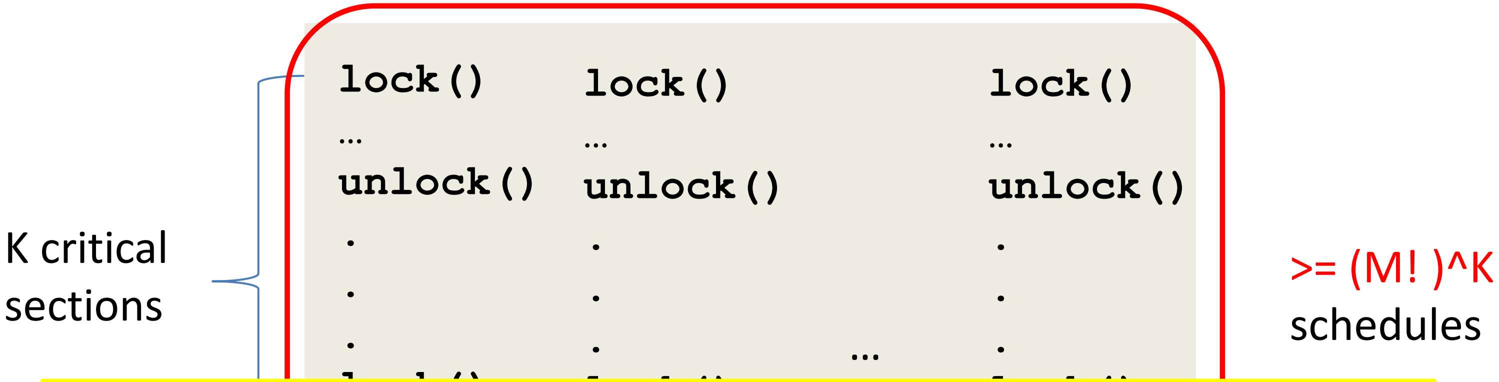
K critical sections

>= (M!)^K schedules

M threads

- Number of schedules: exponential in K, M
- Even more schedules aggregated over all inputs

Why hard?



```
lock()      lock()               lock()
…           …                    …
unlock()    unlock()             unlock()
.           .                    .
.           .                    .
.           .             …      .
```

Finding concurrency bugs

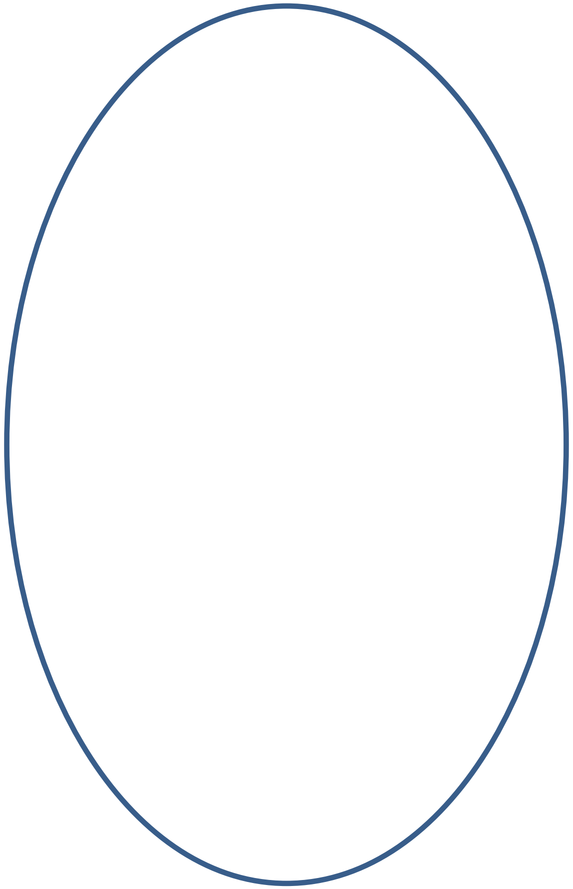
==

finding needles in a haystack

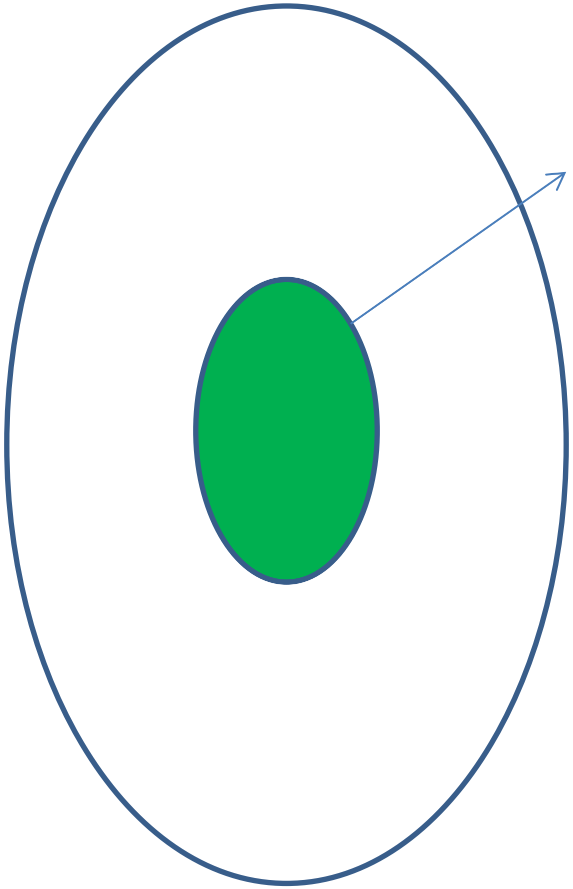
- N
- Even more schedules aggregated over all inputs

How to improve checking coverage?

All possible runtime schedules



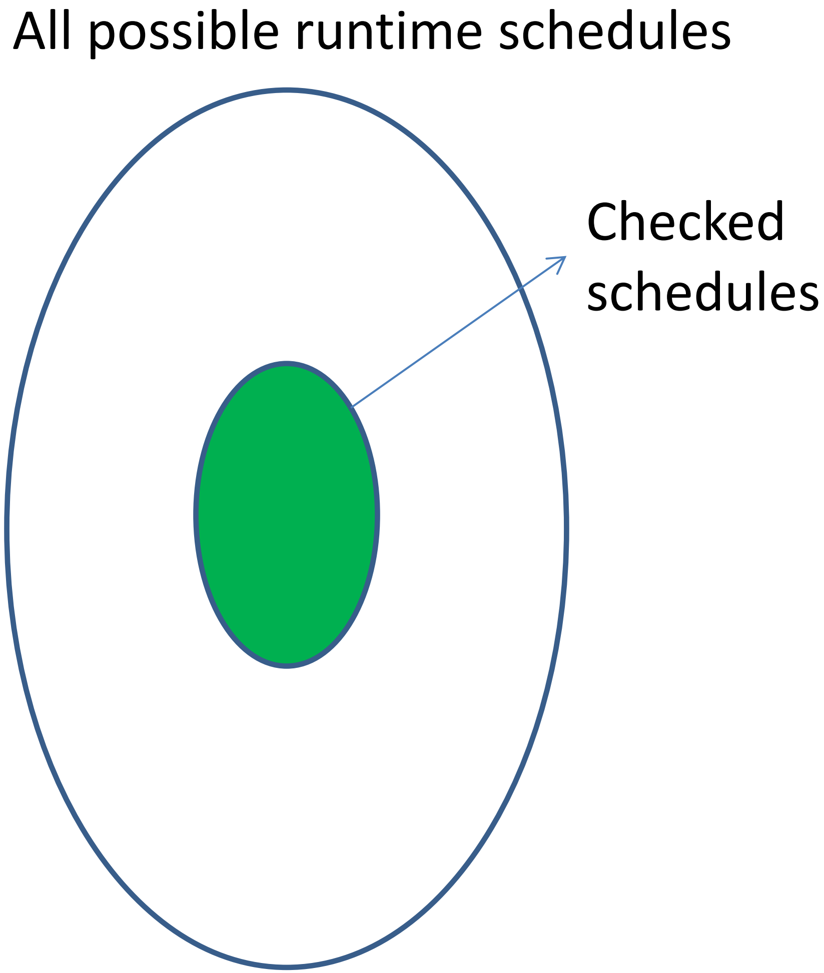
How to improve checking coverage?



All possible runtime schedules

Checked schedules

How to improve checking coverage?

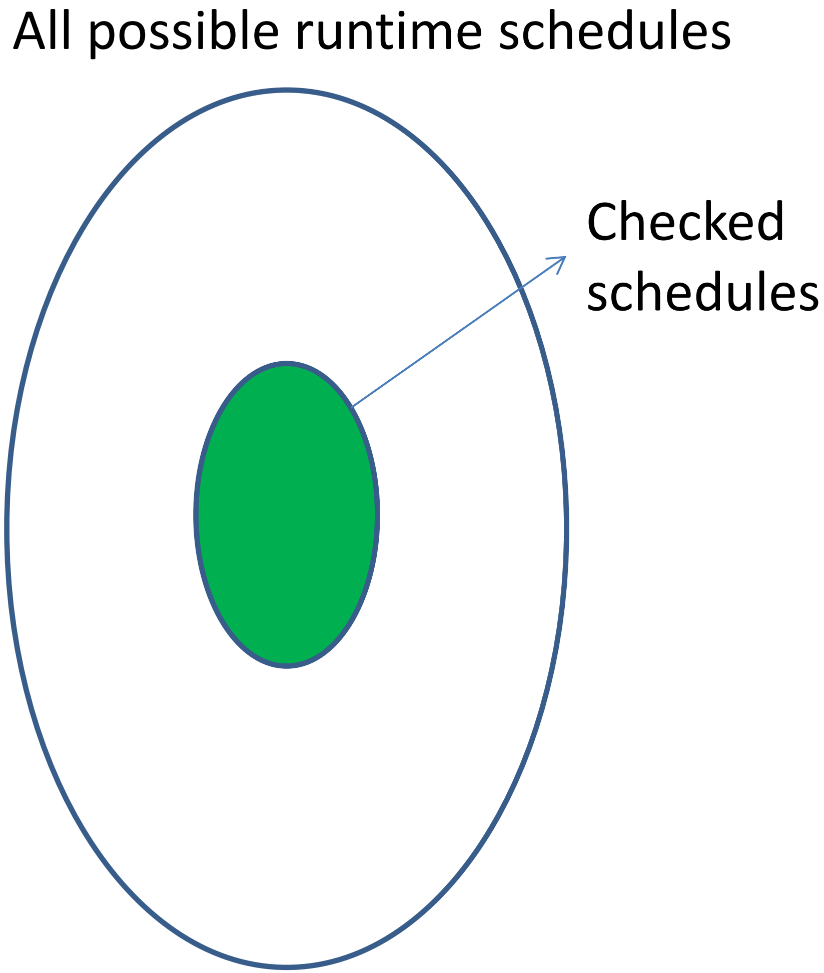


All possible runtime schedules

Checked schedules

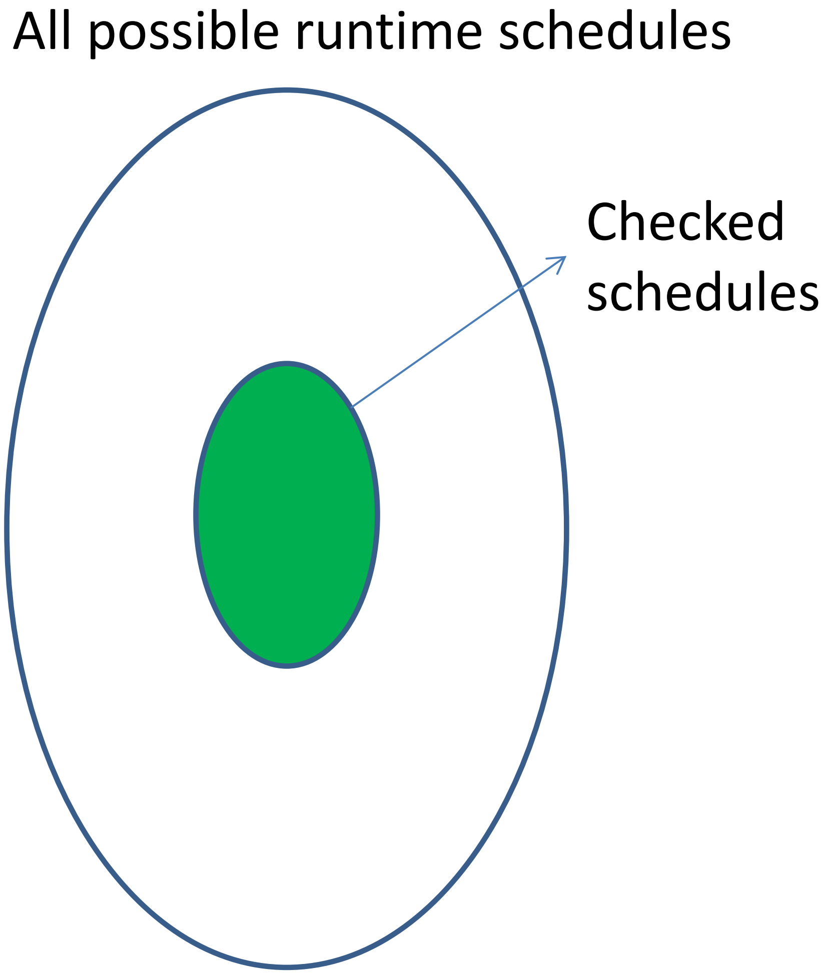
- Coverage = Checked/All

How to improve checking coverage?



- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence

How to improve checking coverage?

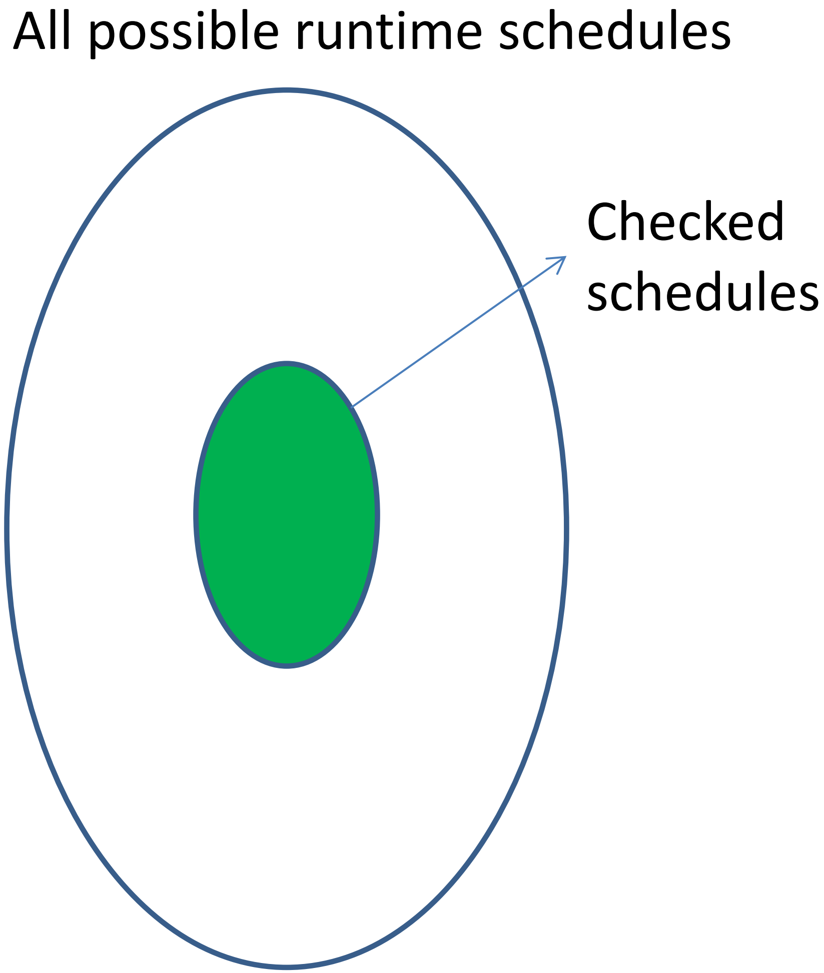


All possible runtime schedules

Checked schedules

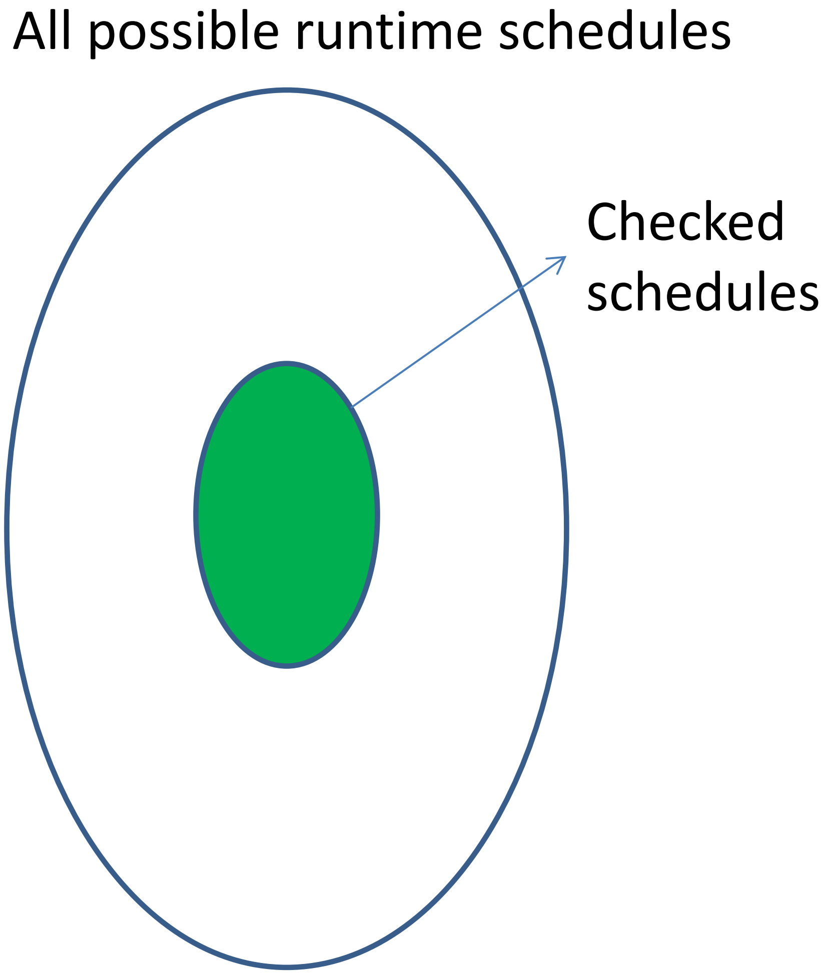
- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
- Equiv. is hard to find

How to improve checking coverage?



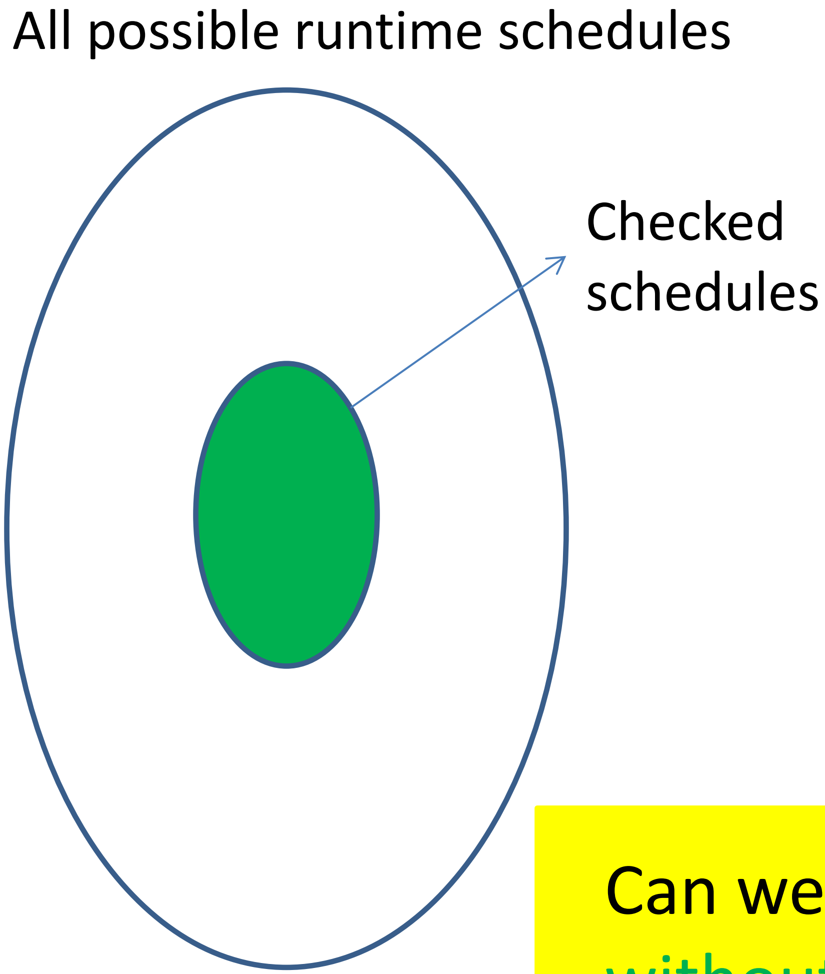
- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
- Equiv. is hard to find
  - [DIR SOSP 11] (joint w/ MSR Asia) took us 3 years

How to improve checking coverage?



- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
- Equiv. is hard to find
  - [DIR SOSP 11] (joint w/ MSR Asia) took us 3 years
  - First after [VeriSoft POPL 97]

How to improve checking coverage?



All possible runtime schedules

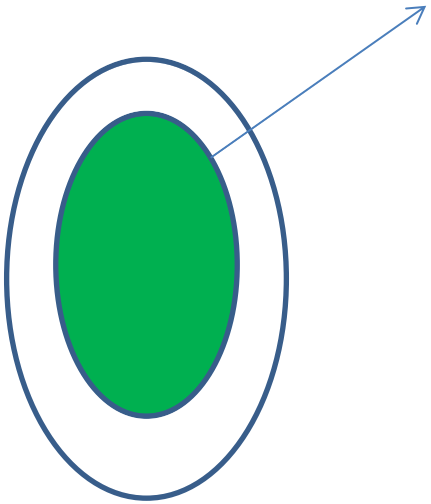
Checked schedules

- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
- Equiv. is hard to find
  - [DIR SOSP 11] (joint w/ MSR Asia) took us 3 years
  - First after [VeriSoft POPL 97]

Can we increase coverage without enlarging Checked?

How to improve checking coverage?

All possible runtime schedules

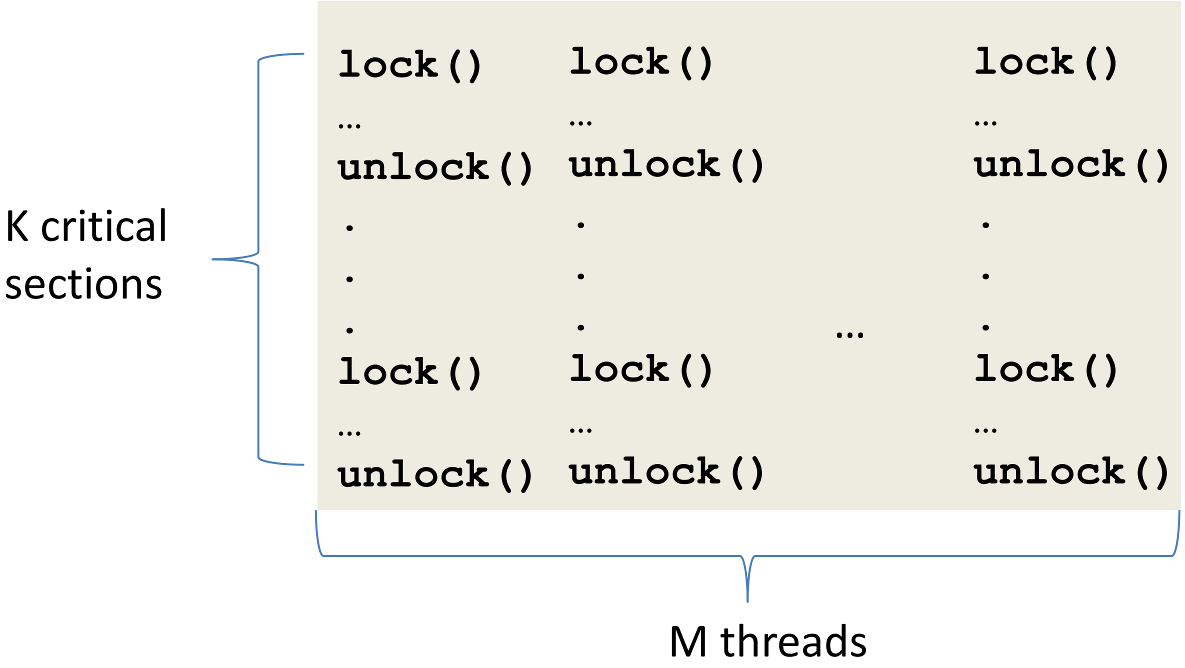


Checked schedules

- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
- Equiv. is hard to find
  - [DIR SOSP 11] (joint w/ MSR Asia) took us 3 years
  - First after [VeriSoft POPL 97]

Can we increase coverage without enlarging Checked?

High coverage with StableMT



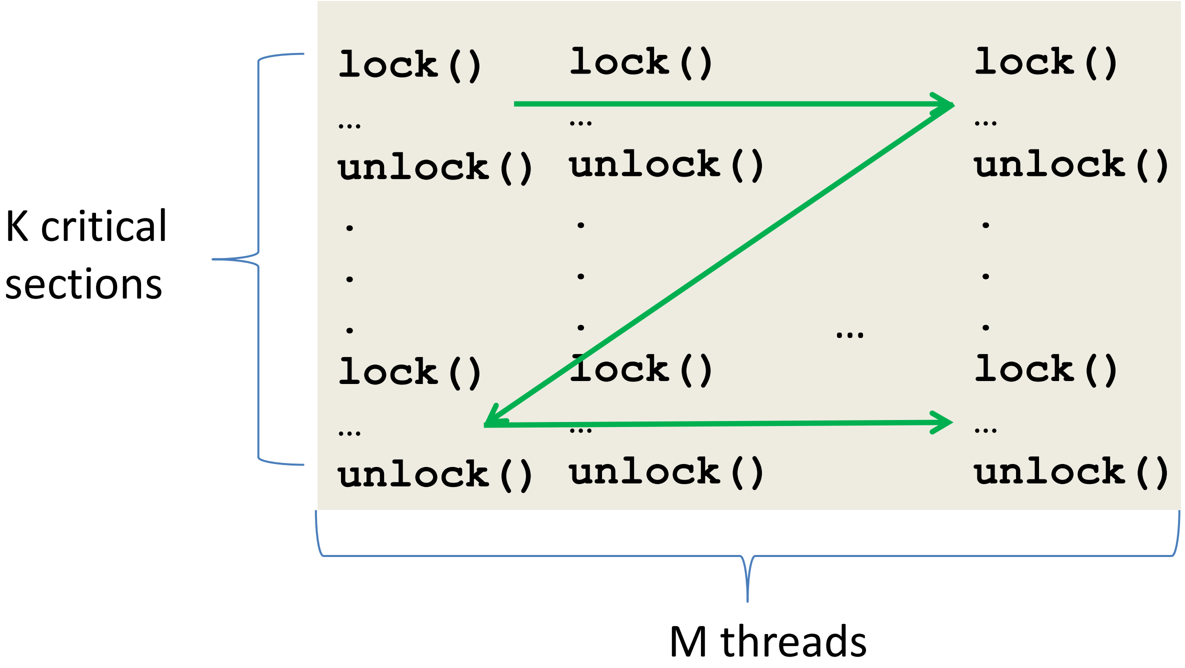
K critical sections

```
lock()     lock()              lock()
…          …                   …
unlock()   unlock()            unlock()
.          .                   .
.          .                   .
.          .             …     .
lock()     lock()              lock()
…          …                   …
unlock()   unlock()            unlock()
```

M threads

- Enforce round-robin synchronization order

High coverage with StableMT



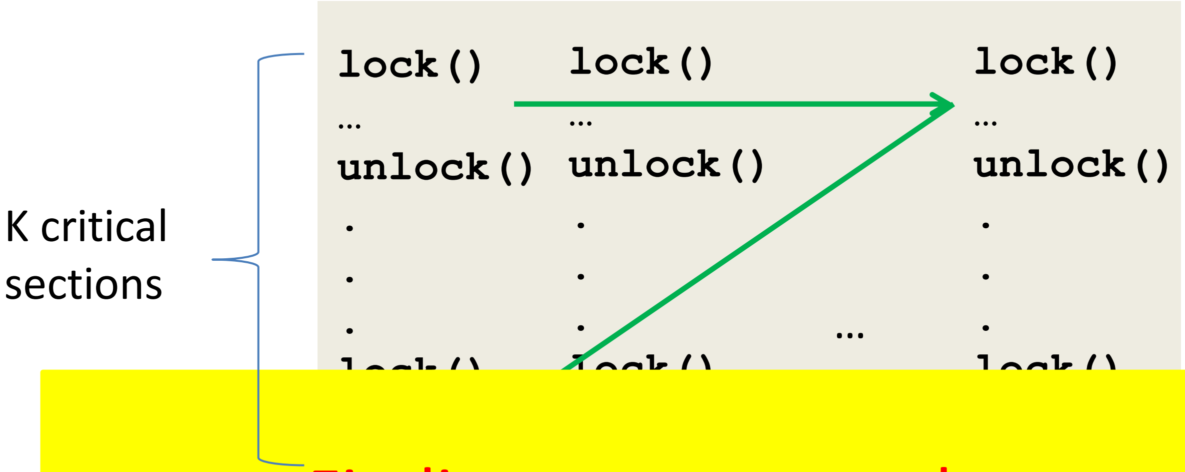
```
lock()     lock()              lock()
…          …                   …
unlock()   unlock()            unlock()
.          .                   .
.          .                   .
.          .          …        .
lock()     lock()              lock()
…          …                   …
unlock()   unlock()            unlock()
```

K critical sections

M threads

- Enforce round-robin synchronization order

High coverage with StableMT



K critical sections

```
lock()     lock()                 lock()
…          …                      …
unlock()   unlock()               unlock()
.          .                      .
.          .                      .
.          .              …       .
lock()     lock()                 lock()
```

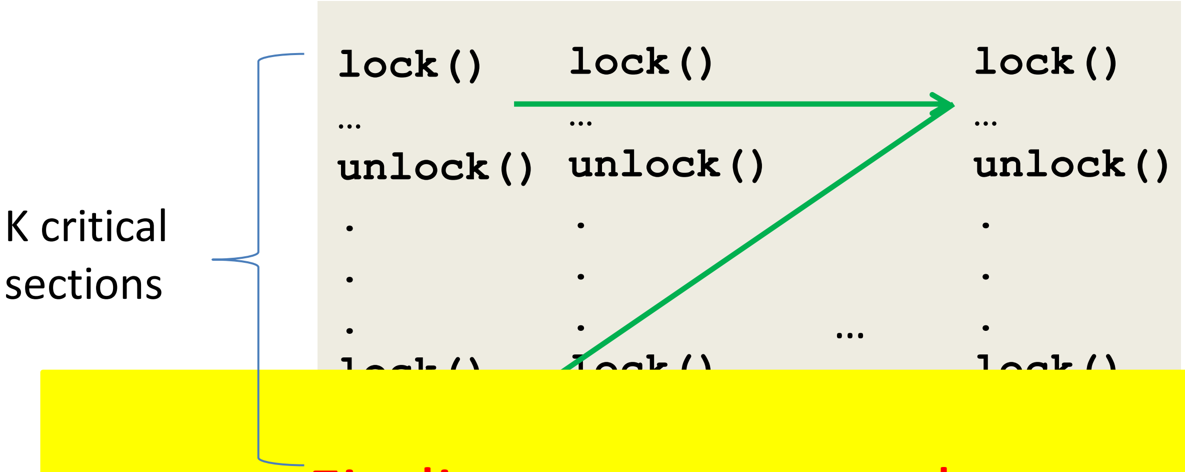
Finding concurrency bugs

==

checking one schedule

- er

High coverage with StableMT



K critical sections

```
lock()      lock()                    lock()
…           …                         …
unlock()    unlock()                  unlock()
.           .                         .
.           .                         .
.           .               …         .
lock()      lock()                    lock()
```

Finding concurrency bugs

==

checking one schedule

Simple enough that it feels like cheating ☺

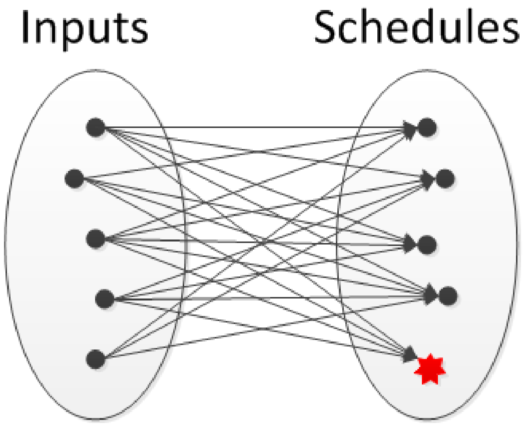
- er

Are all of the exponentially many schedules necessary?

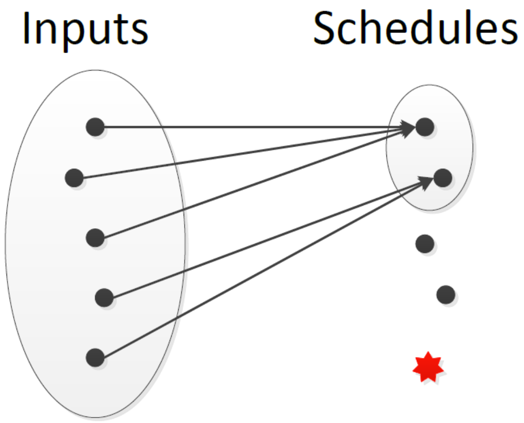
- Insight 1: for many programs, a wide range of inputs shares the same set of schedules [Tern OSDI 10] [Peregrine SOSP 11]

- Insight 2: the overhead of enforcing a schedule on different inputs is low (e.g., 15%) [Tern OSDI 10] [Peregrine SOSP 11]

Stable Multithreading



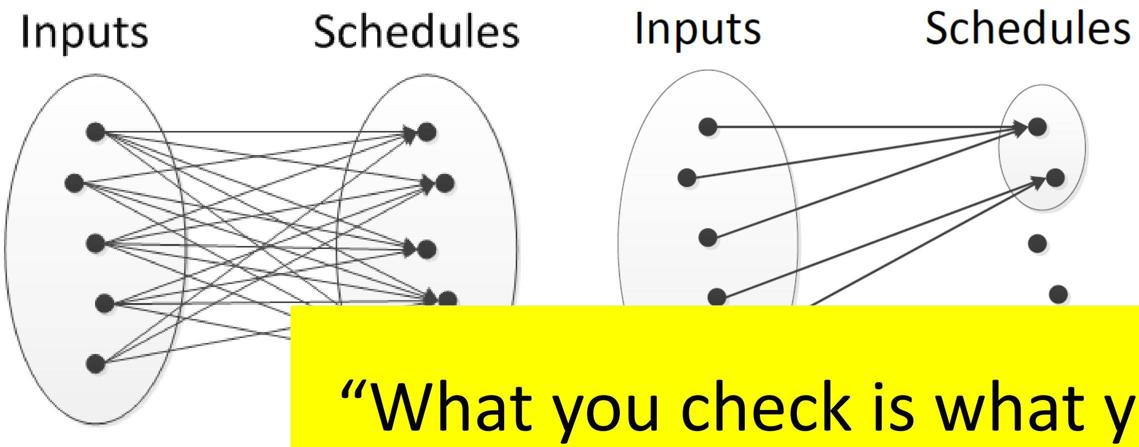
Traditional multithreading



Stable multithreading

- All inputs → a greatly reduced set of schedules
- Key benefits
  - Vastly shrink the haystack → needles much easier to find
  - Provide anticipated *stability* (robustness against input or program perturbations)

Stable Multithreading



Traditio
multith

“What you check is what you run”

“What you can’t check is not run”

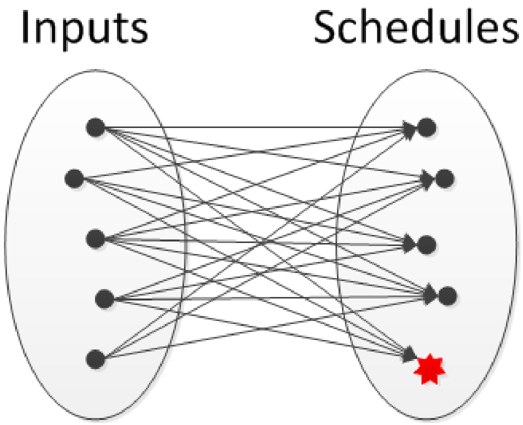
Tool + runtime co-design

- All inpu
- Key be
  - Vastly shrink the haystack ➔ needles much easier to find
  - Provide anticipated *stability* (robustness against input or program perturbations)

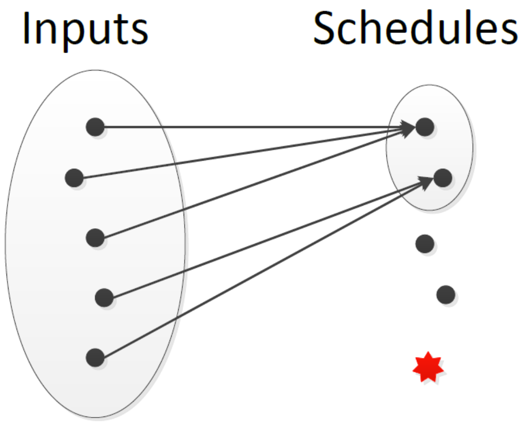
Stability and determinism are two separate, complementary properties.

Stability is more useful for reliability.

Deterministic multithreading (DMT): one input → one schedule

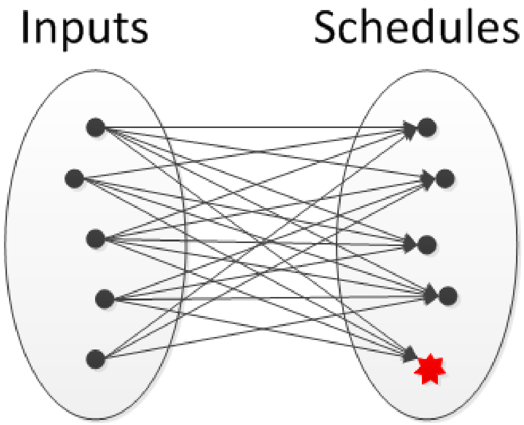


Traditional multithreading

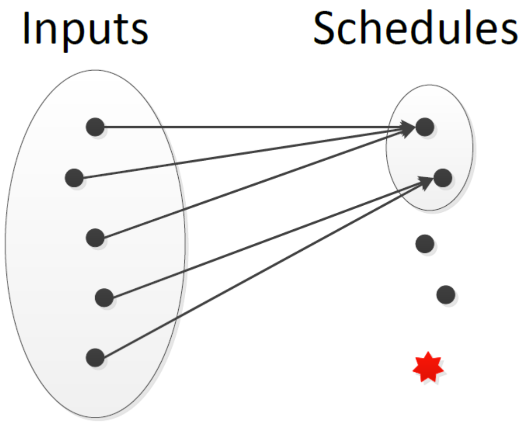


Stable multithreading

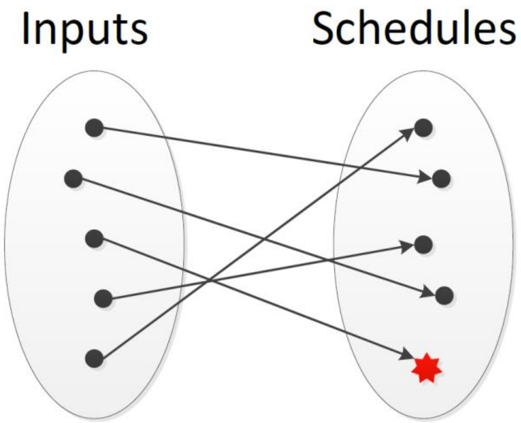
Deterministic multithreading (DMT): one input → one schedule



Traditional multithreading

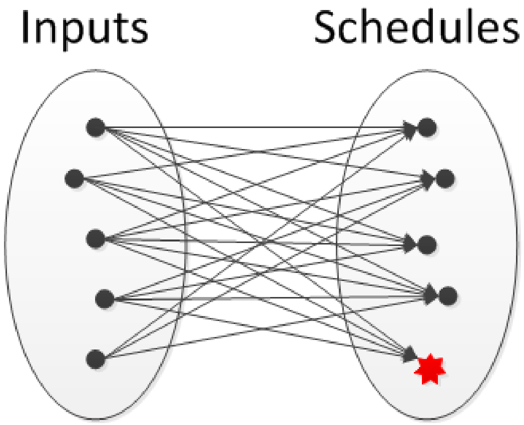


Stable multithreading

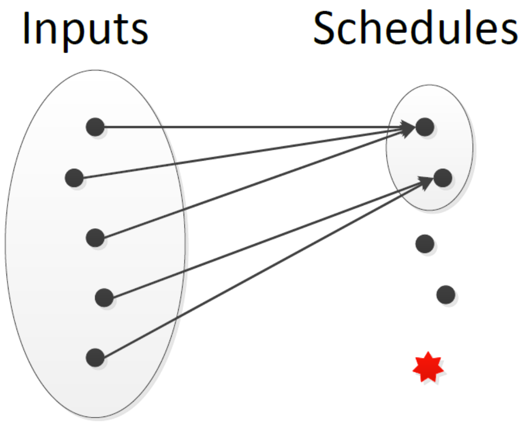


Deterministic multithreading

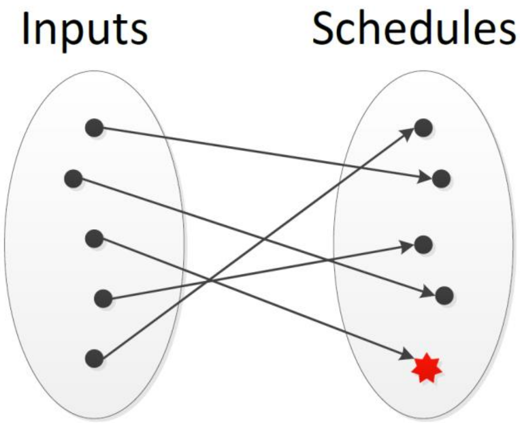
Deterministic multithreading (DMT): one input → one schedule



Traditional multithreading



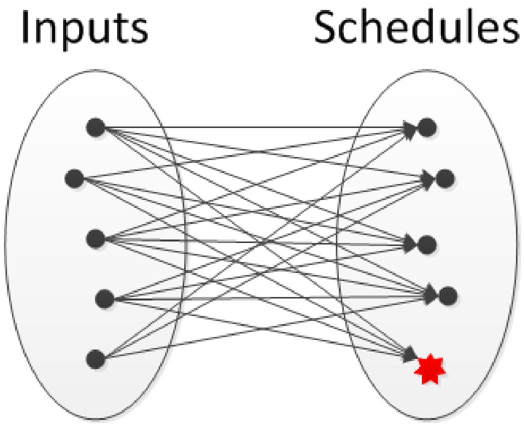
Stable multithreading



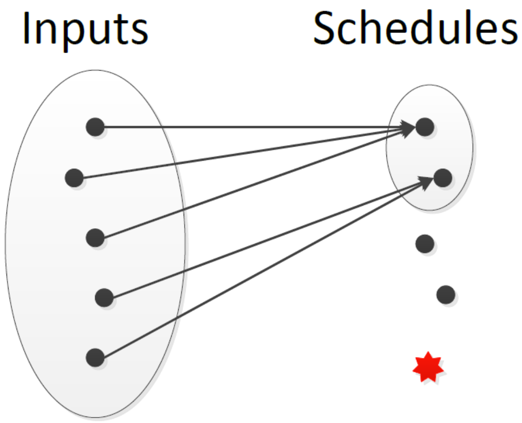
Deterministic multithreading

- One testing execution validates all future executions on the same input
- Reproducing a concurrency bug requires only the input

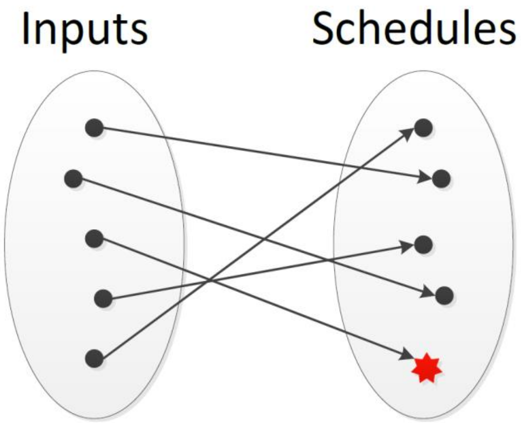
Deterministic multithreading (DMT): one input → one schedule



Traditional multithreading



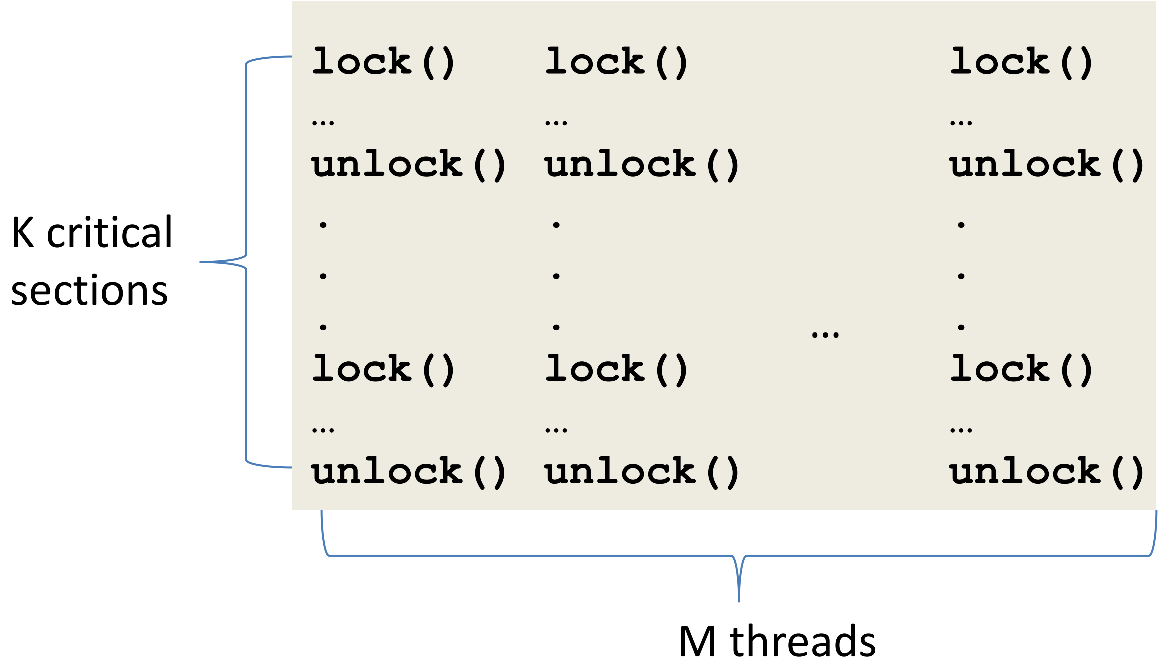
Stable, deterministic multithreading



Deterministic multithreading

- One testing execution validates all future executions on the same input
- Reproducing a concurrency bug requires only the input

Input or program perturbation → different schedules

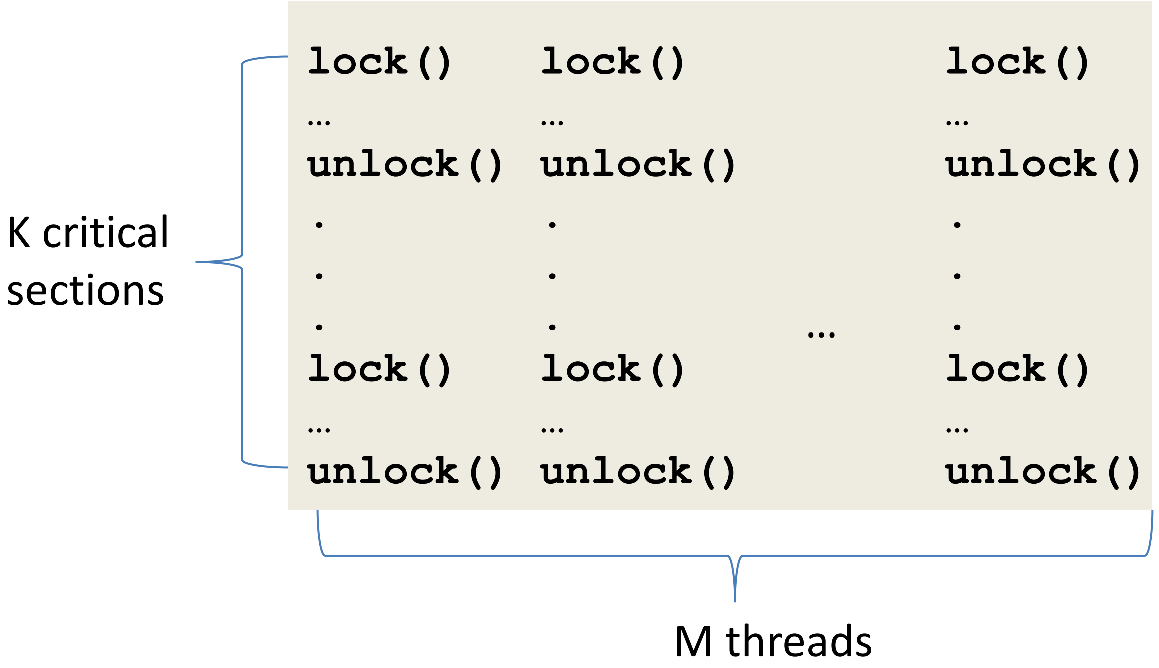


```
lock()     lock()                lock()
…          …                     …
unlock()   unlock()              unlock()
.          .                     .
.          .                     .
.          .               …     .
lock()     lock()                lock()
…          …                     …
unlock()   unlock()              unlock()
```

K critical sections

M threads

Input or program perturbation ➔ different schedules



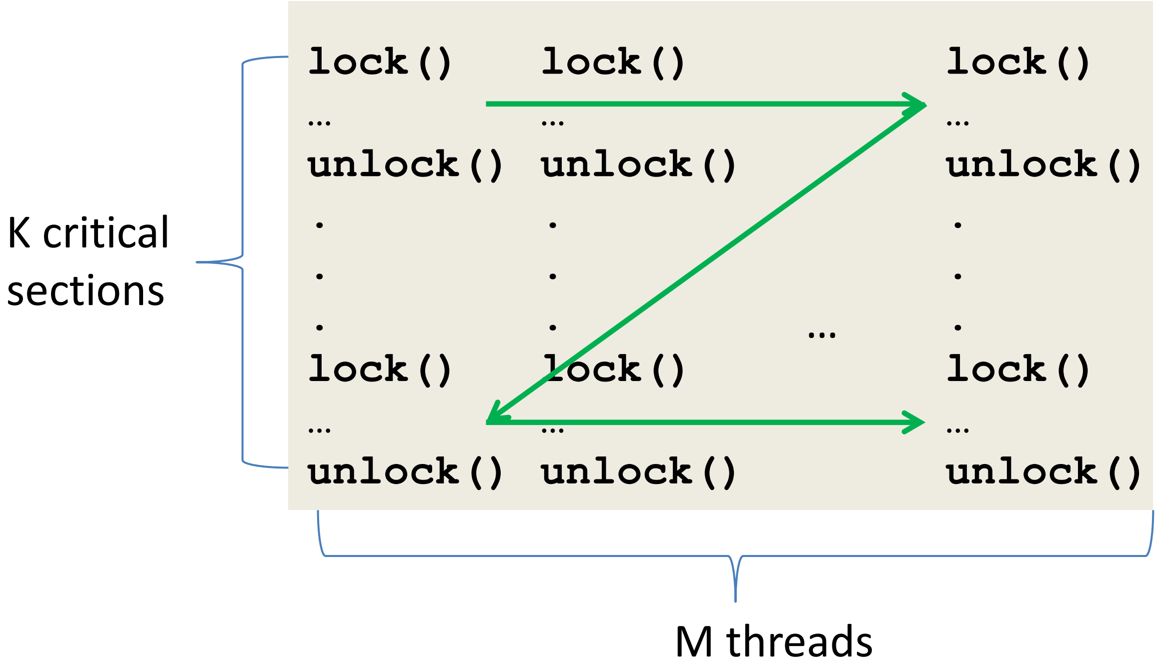
```
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
.         .                   .
.         .                   .
.         .         …         .
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
```

K critical sections

M threads

Input 1

Input or program perturbation ➔ different schedules



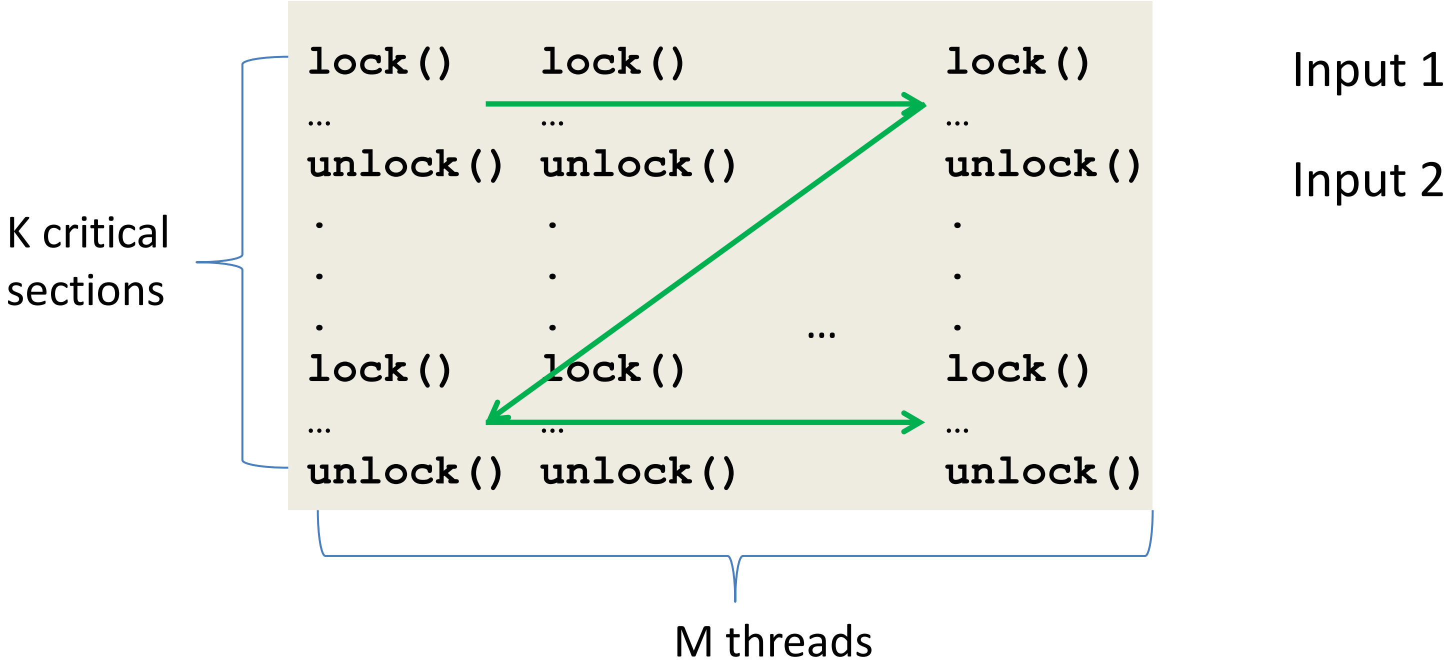
```
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
.         .                   .
.         .                   .
.         .          …        .
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
```

K critical sections

M threads

Input 1

Input or program perturbation ➔ different schedules



```
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
.         .                   .
.         .                   .
.         .          …        .
lock()    lock()              lock()
…         …                   …
unlock()  unlock()            unlock()
```

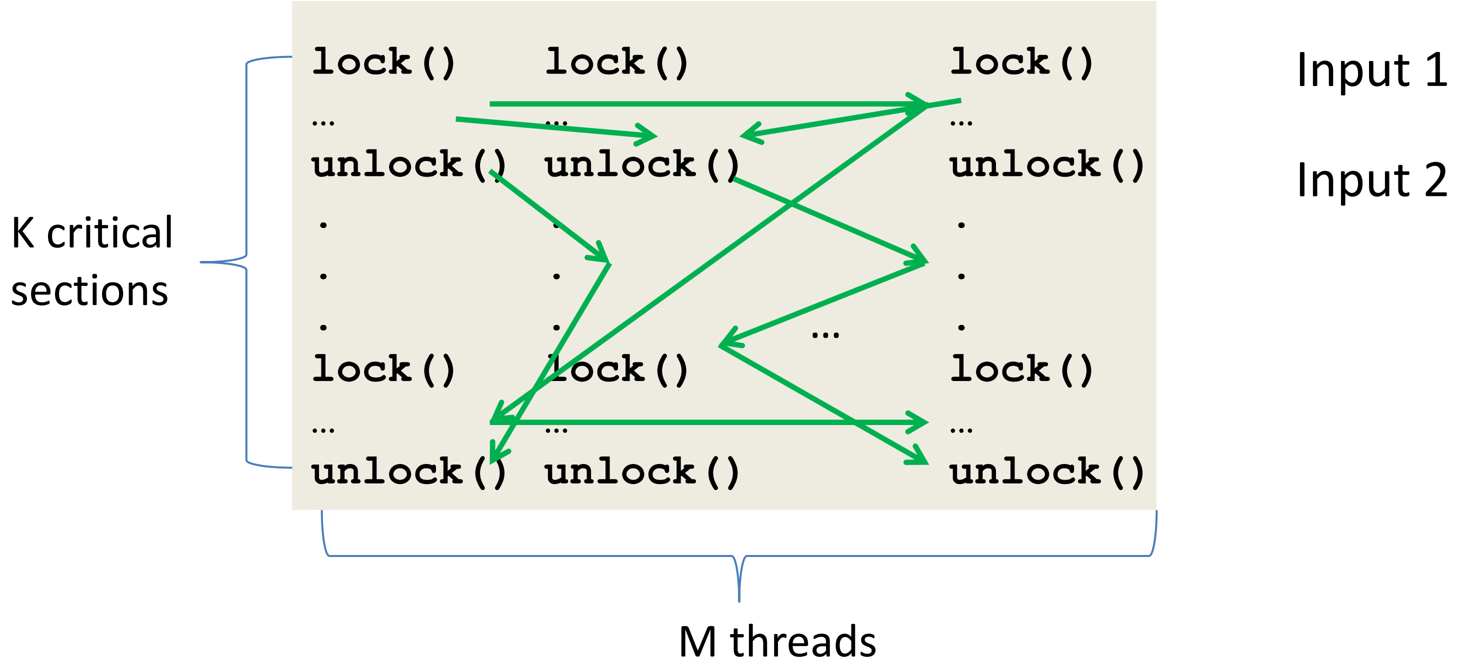
K critical sections

M threads

Input 1

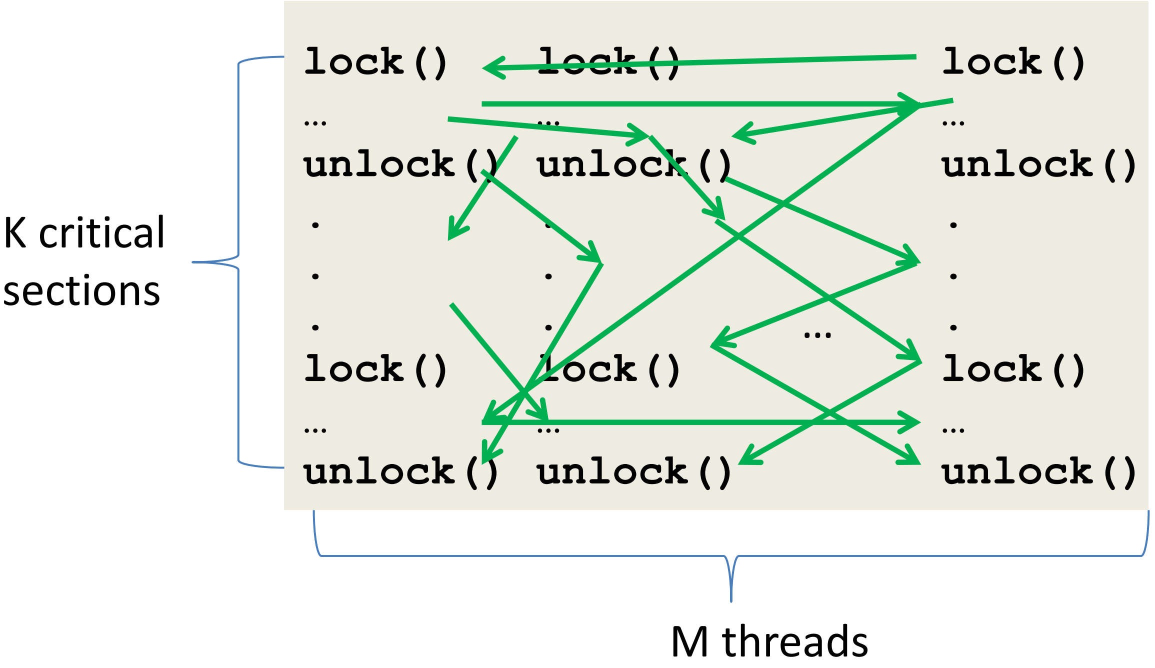
Input 2

Input or program perturbation ➔ different schedules



```
lock()     lock()            lock()
…          …                 …
unlock()   unlock()          unlock()
.          .                 .
.          .                 .
.          .        …        .
lock()     lock()            lock()
…          …                 …
unlock()   unlock()          unlock()
```

Input or program perturbation ➔ different schedules



```
lock()      lock()            lock()
…           …                 …
unlock()    unlock()          unlock()
.           .                 .
.           .                 .
.           .          …      .
lock()      lock()            lock()
…           …                 …
unlock()    unlock()          unlock()
```

K critical sections

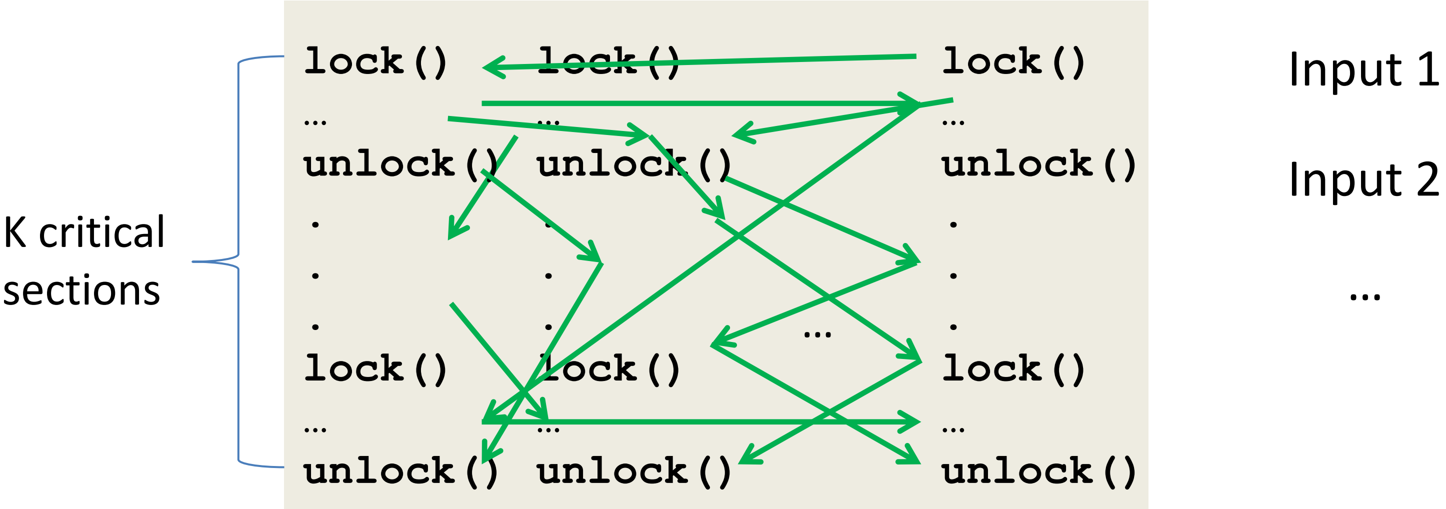
M threads

Input 1

Input 2

…

Input or program perturbation ➔ different schedules



K critical sections

```
lock()     lock()            lock()
…          …                 …
unlock()   unlock()          unlock()
.          .                 .
.          .                 .
.          .        …        .
lock()     lock()            lock()
…          …                 …
unlock()   unlock()          unlock()
```

Input 1

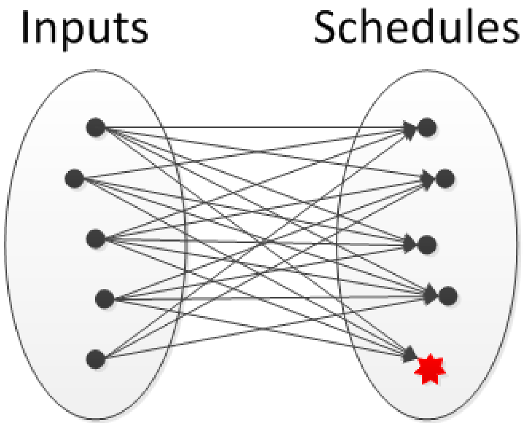
Input 2

…

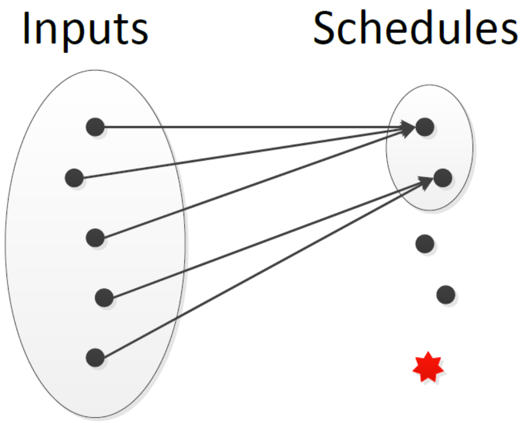
Still too many schedules

Unstable!

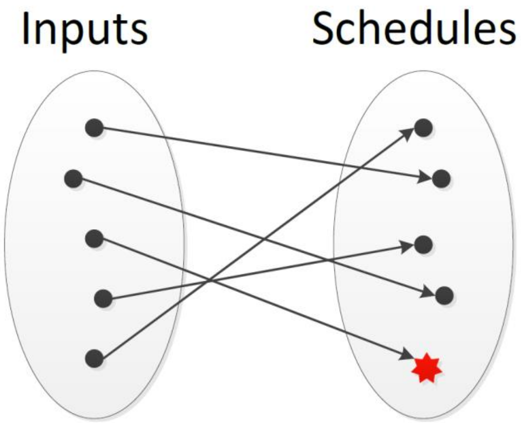
Deterministic but not stable



Traditional
multithreading



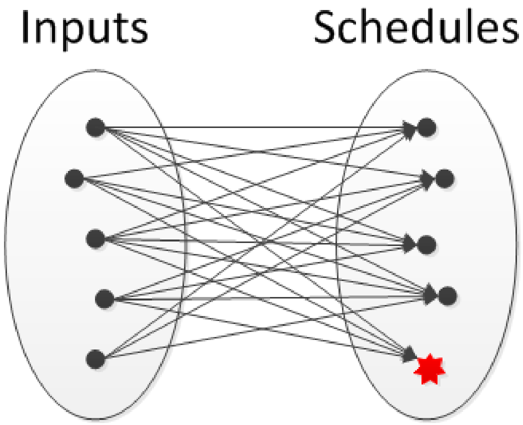
Stable, deterministic
multithreading



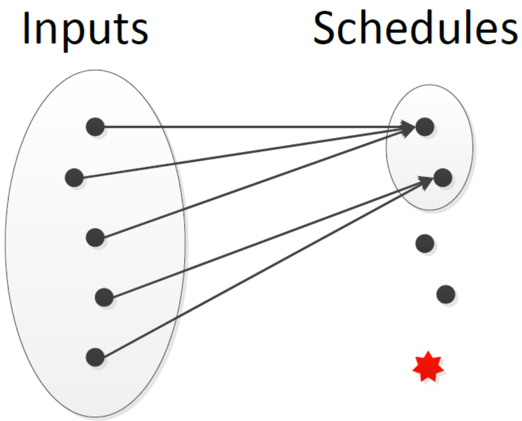
Deterministic, unstable
multithreading

- Determinism is a narrow property
  - Same input + same program ➔ same behavior
  - Input or program changes slightly? Can be unstable

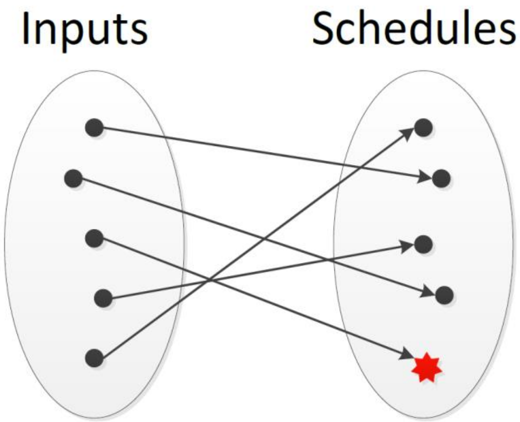
Deterministic but not stable



Traditional multithreading



Stable, deterministic multithreading

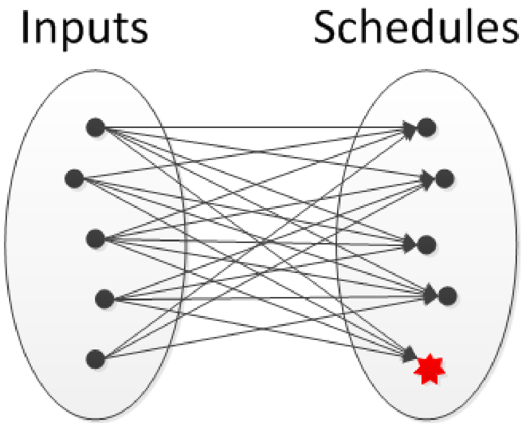


Deterministic, unstable multithreading

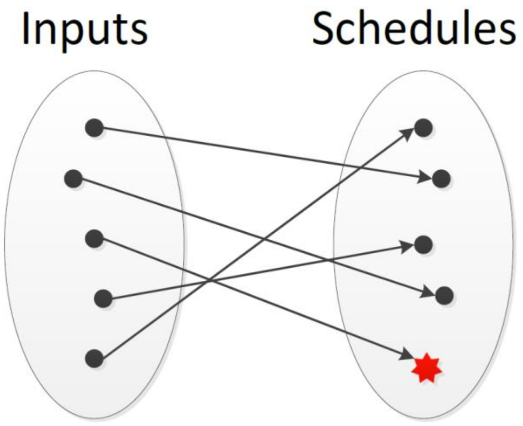
- Determ
  - Same ior
  - Input or program changes slightly? Can be unstable

Determinism and stability are often mistakenly conflated

Stable but not deterministic



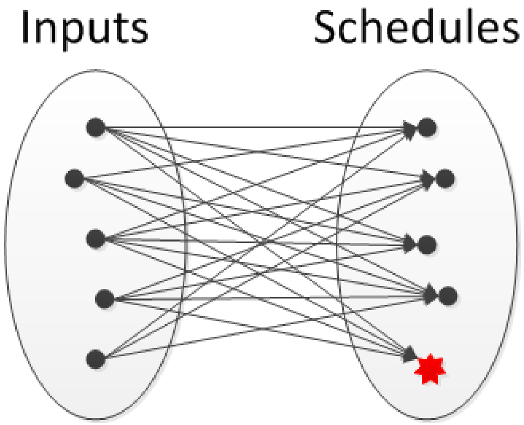
Traditional multithreading



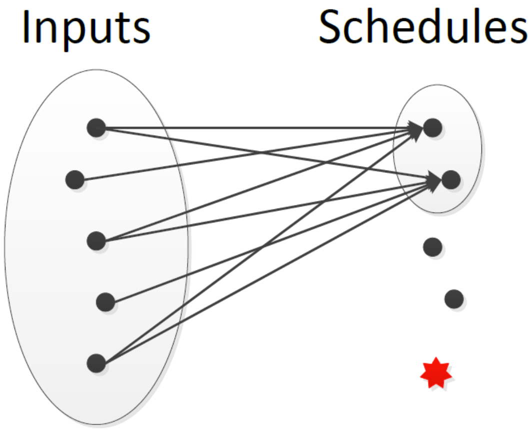
Deterministic multithreading

- Determinism is a binary property
  - Nondeterministic if one input → n > 1 schedules

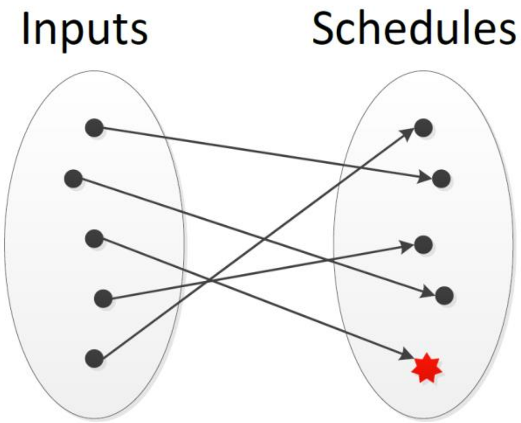
Stable but not deterministic



Traditional
multithreading



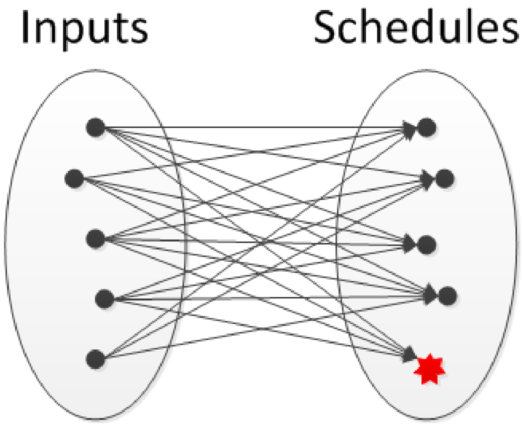
Stable, nondeterministic
multithreading



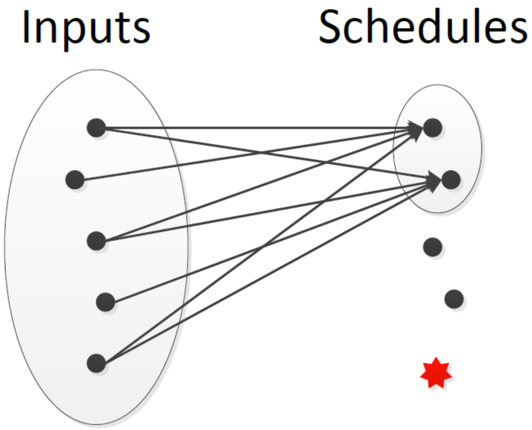
Deterministic
multithreading

- Determinism is a binary property
  - Nondeterministic if one input ➔ n > 1 schedules

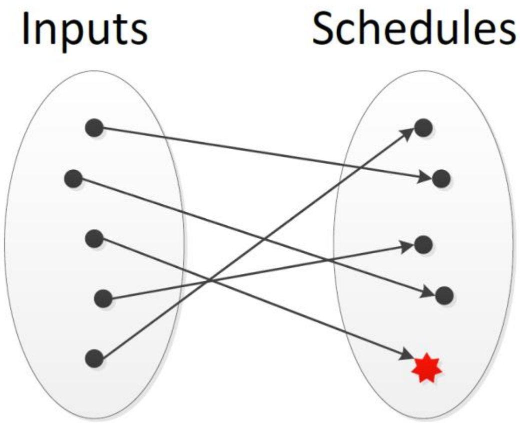
Stable but not deterministic



Traditional multithreading



Stable, nondeterministic multithreading



Deterministic multithreading

- Det
  - N

  es

Nondeterministic but stable → easy to be made reliable through checking

How to build StableMT systems

Key challenge: how to compute the schedules to map inputs to

- Requirements on the schedules
  - Stability: process many inputs
  - Performance: reasonably fast

Key challenge: how to compute the schedules to map inputs to

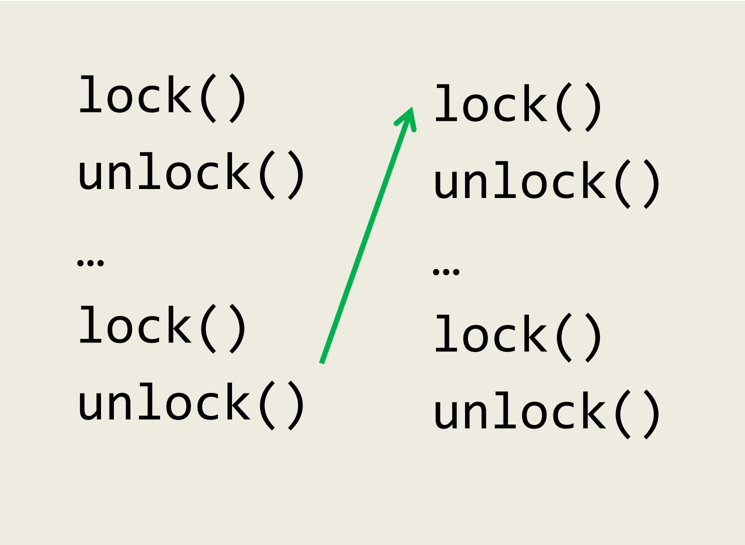
- Requirements on the schedules
  - Stability: process many inputs
  - Performance: reasonably fast

hard!

Key challenge: how to compute the schedules to map inputs to

- Requirements on the schedules
  - Stability: process many inputs
  - Performance: reasonably fast

hard!

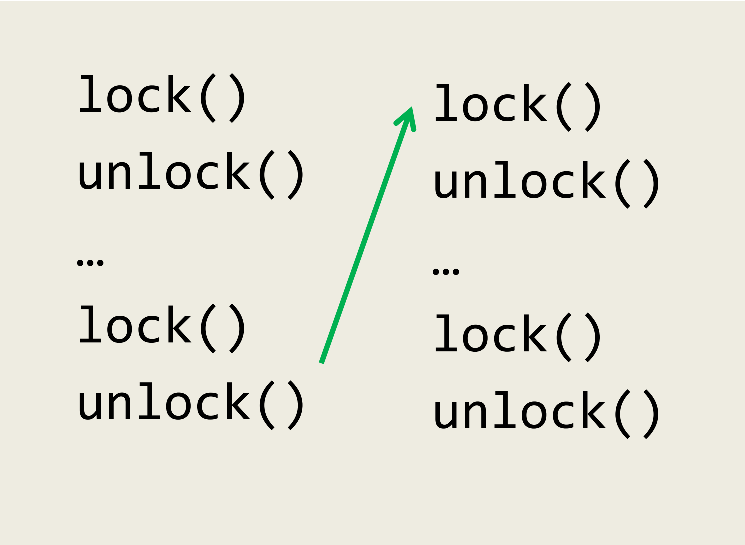


```
lock()          lock()
unlock()        unlock()
…               …
lock()          lock()
unlock()        unlock()
```

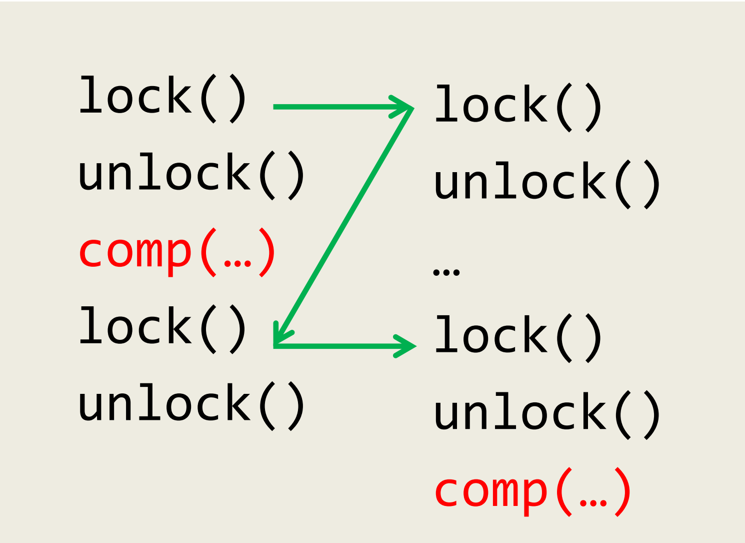
Key challenge: how to compute the schedules to map inputs to

- Requirements on the schedules
  - Stability: process many inputs
  - Performance: reasonably fast

hard!



```
lock()          lock()
unlock()        unlock()
…               …
lock()          lock()
unlock()        unlock()
```



```
lock()          lock()
unlock()        unlock()
comp(…)         …
lock()          lock()
unlock()        unlock()
                comp(…)
```

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

```
if(x == 1) {
  lock();
  unlock();
} else
  …; // no synch
```

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

```
if(x == 1) {
  lock();
  unlock();
} else
  …; // no synch
```

```
if(y == 1)
  …; // no synch
else
  …; // no synch
```

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

```
if(x == 1) {
  lock();
  unlock();
} else
  …; // no synch
```

```
if(y == 1)
  …; // no synch
else
  …; // no synch
```

Precondition should constrain x, but not y

Our 1st attempt: record and reuse synchronization schedules

- On new input, run program as is to record reasonably fast synchronization schedule
- Compute relaxed, quickly checkable precondition of the schedule to capture dependencies on input
- Reuse schedule on inputs satisfying precondition

```
i
}
  ...; // no synch
```

Solution: symbolic execution to track constraints and precondition slicing to remove unnecessary constraints

Precondition should constrain x, but not y

The problem of data races

- May cause execution to deviate from schedule

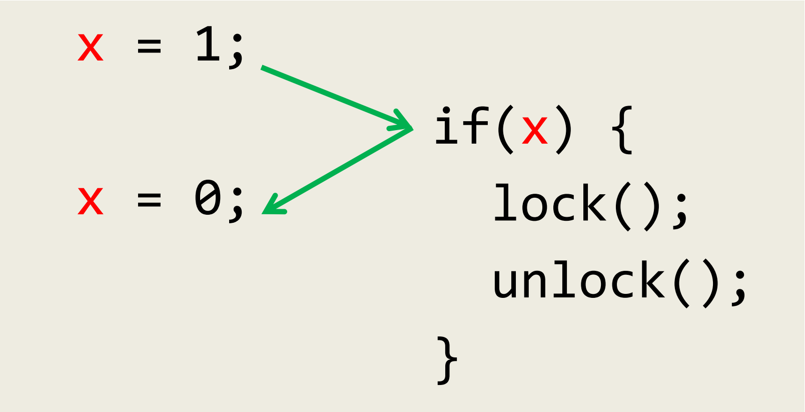
The problem of data races

- May cause execution to deviate from schedule

```
x = 1;
            if(x) {
x = 0;        lock();
              unlock();
            }
```

The problem of data races

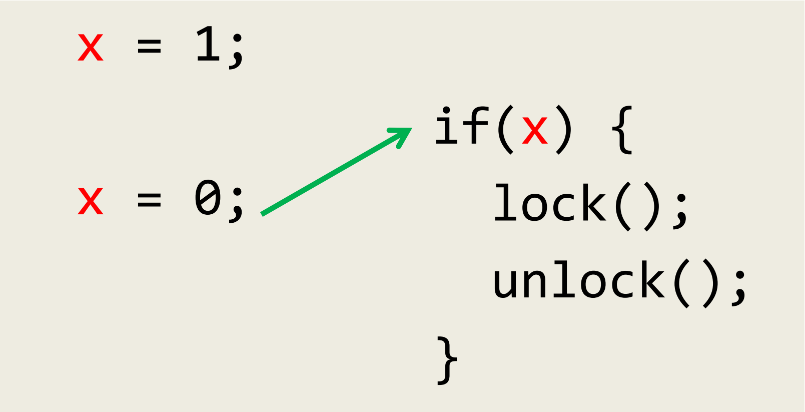
- May cause execution to deviate from schedule



```
x = 1;
            if(x) {
x = 0;        lock();
              unlock();
            }
```

The problem of data races

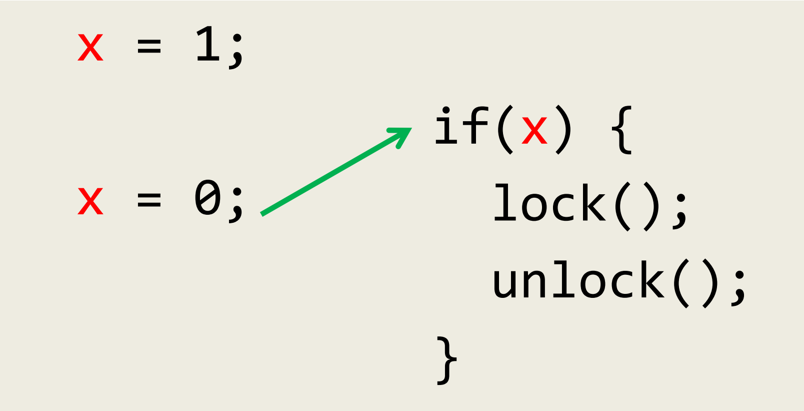
- May cause execution to deviate from schedule



```
x = 1;
      if(x) {
x = 0;
       lock();
       unlock();
      }
```

The problem of data races

- May cause execution to deviate from schedule

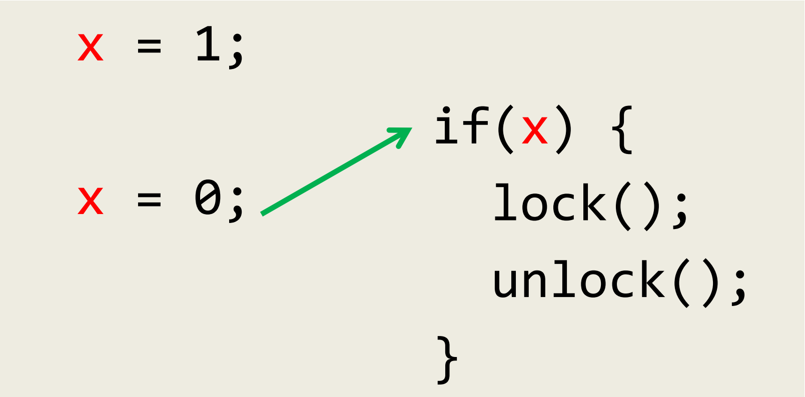


```
x = 1;
             if(x) {
x = 0;         lock();
               unlock();
             }
```

```
a[x] = 1;
             if(a[y]) {
a[x] = 0;      lock();
               unlock();
             }
```

The problem of data races

- May cause execution to deviate from schedule



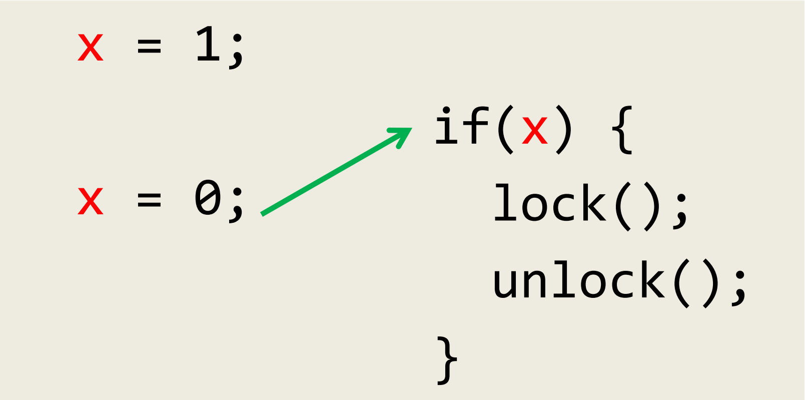
```
  x = 1;
            if(x) {
  x = 0;      lock();
              unlock();
            }
```

```
  a[x] = 1;
            if(a[y]) {
  a[x] = 0;   lock();
              unlock();
            }
```

Solution: custom race detector to detect such races, then custom instrumentor to deterministically resolve races at runtime

The problem of data races

- May cause execution to deviate from schedule



```
x = 1;
              if(x) {
x = 0;          lock();
                unlock();
              }
```

```
a[x] = 1;
              if(a[y]) {
a[x] = 0;       lock();
                unlock();
              }
```

Solution: custom race detector to detect such races, then custom instrumentor to deterministically resolve races at runtime

Our 1st attempt: sophisticated enough that it needed [Tern OSDI 10] [Loom OSDI 10] [Peregrine SOSP 11] to explain

Attempts by others

- Ignore thread load imbalance [Dthreads SOSP 11] ➔ sometimes pathological slowdown (e.g., 100x) because parallel computations are serialized
- Fine-grained load balancing with instruction counts [DMP ASPLOS 09] [Kendo ASPLOS 09] [CoreDet ASPLOS 10] ➔ unstable

Attempts by others

- Ignore thread load imbalance [Dthreads SOSP 11] ➔ sometimes pathological slowdown (e.g., 100x) because parallel computations are serialized
- Fine-grained load balancing with instruction counts [DMP ASPLOS 09] [Kendo ASPLOS 09] [CoreDet ASPLOS 10] ➔ unstable

Seems a very hard challenge,
but there's a simple solution!

Insight

- Empirical study of 100+ programs
- Most threads spend majority of time in a small # of core computations
  - Obvious in retrospect: another example of 80-20 rule
- Balance core computations ➔ small overhead

Insight

- Empirical study of 100+ programs
- Most threads spend majority of time in a small # of core computations
  - Obvious in retrospect: another example of 80-20 rule
- Balance core computations ➔ small overhead

Coarse-grained load balancing
is good enough!

Performance hints in Parrot

[Parrot SOSP 13]

- By default, the Parrot thread runtime runs synchronizations round-robin
- When necessary, developers add performance hints to their code for speed
  - Soft barrier: “coschedule these computations”
  - Performance critical section: “get through this code section fast”
- Evaluation on 100+ programs shows that hints are easy to add and make executions fast
- https://github.com/columbia/smt-mc/

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }
```

```
pthread_mutex_lock(&mu);
enqueue(q, block);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mu);
```

```
consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }
```

```
pthread_mutex_lock(&mu);
enqueue(q, block);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mu);
```

```
consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }
```

```
pthread_mutex_lock(&mu);
enqueue(q, block);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mu);
```

```
consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

```
pthread_mutex_lock(&mu);
// termination logic elided
while (empty(q))
  pthread_cond_wait(&cv, &mu);
char *block = dequeue(q);
pthread_mutex_unlock(&mu);
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }
```

```
pthread_mutex_lock(&mu);
enqueue(q, block);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mu);
```

```
consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

```
pthread_mutex_lock(&mu);
// termination logic elided
while (empty(q))
  pthread_cond_wait(&cv, &mu);
char *block = dequeue(q);
pthread_mutex_unlock(&mu);
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

- Schedules are data-independent, so that same schedule can compress any file regardless of file contents

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

- Schedules are data-independent, so that same schedule can compress any file regardless of file contents
- Core computations: compress()

Example based on PBZip2

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

- Schedules are data-independent, so that same schedule can compress any file regardless of file contents
- Core computations: compress()

```
$ LD_PRELOAD=parrot.so ./a.out file_with_two_blocks
```

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

main   consumer 1   consumer 2

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

```
main      consumer 1    consumer 2
          (waiting)     (waiting)
```

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

main	consumer 1 (waiting)	consumer 2 (waiting)
read_block Enqueue		

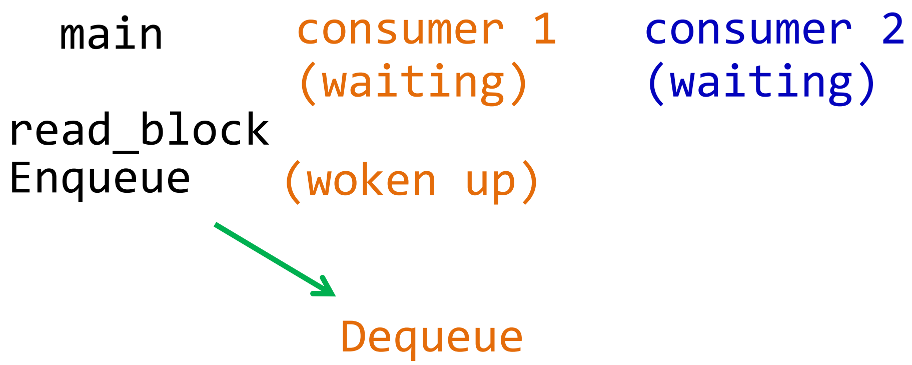
Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```

main	consumer 1 (waiting)	consumer 2 (waiting)
read_block		
Enqueue	(woken up)	

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

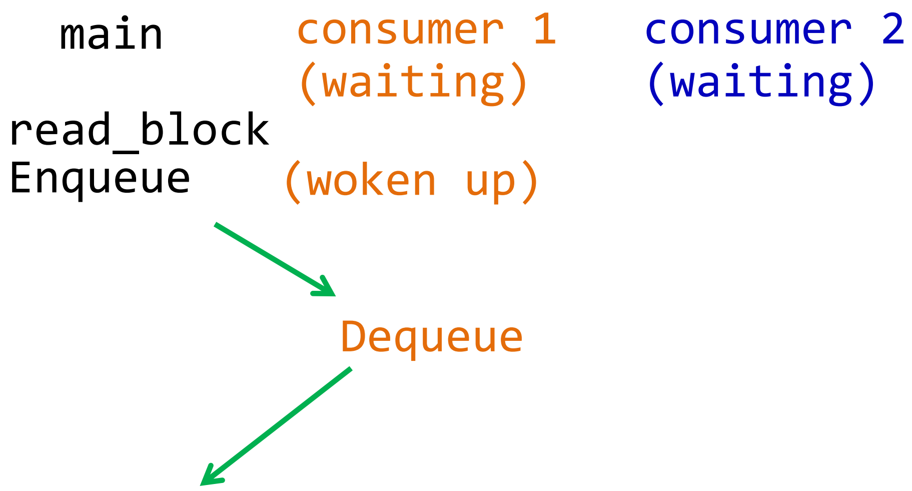
read_block
Enqueue

(woken up)

Dequeue

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

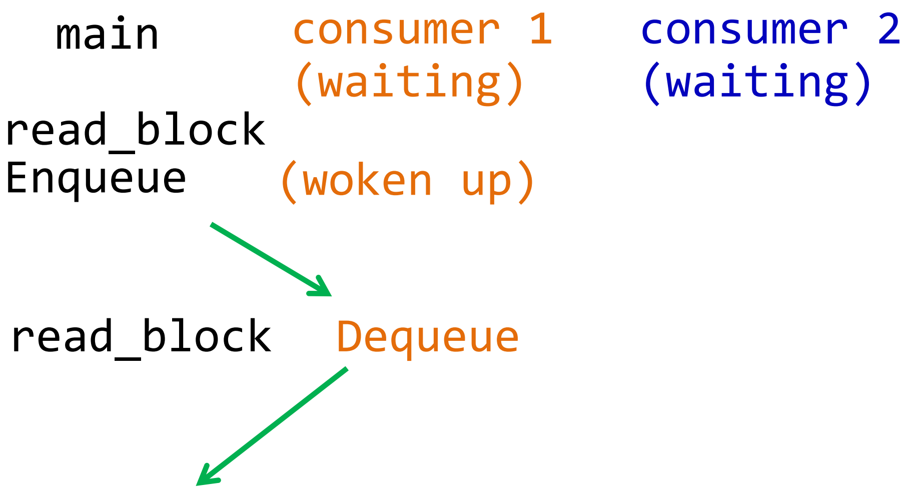
read_block
Enqueue

(woken up)

Dequeue

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

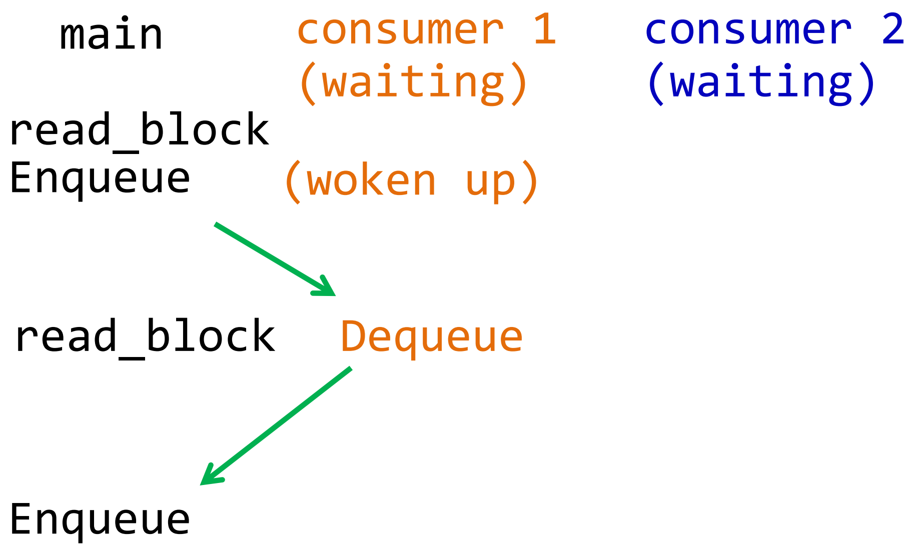
(woken up)

read_block

Dequeue

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

(woken up)

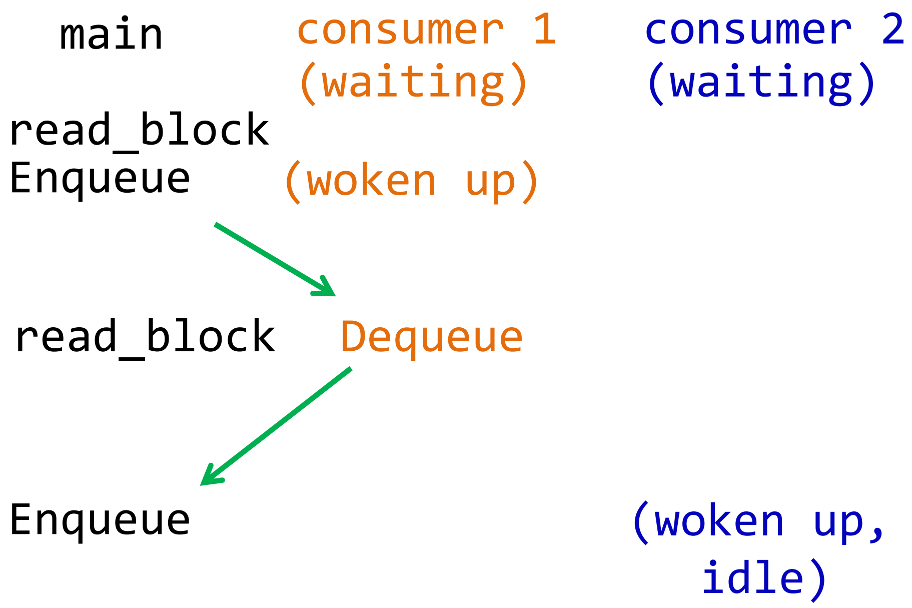
read_block

Dequeue

Enqueue

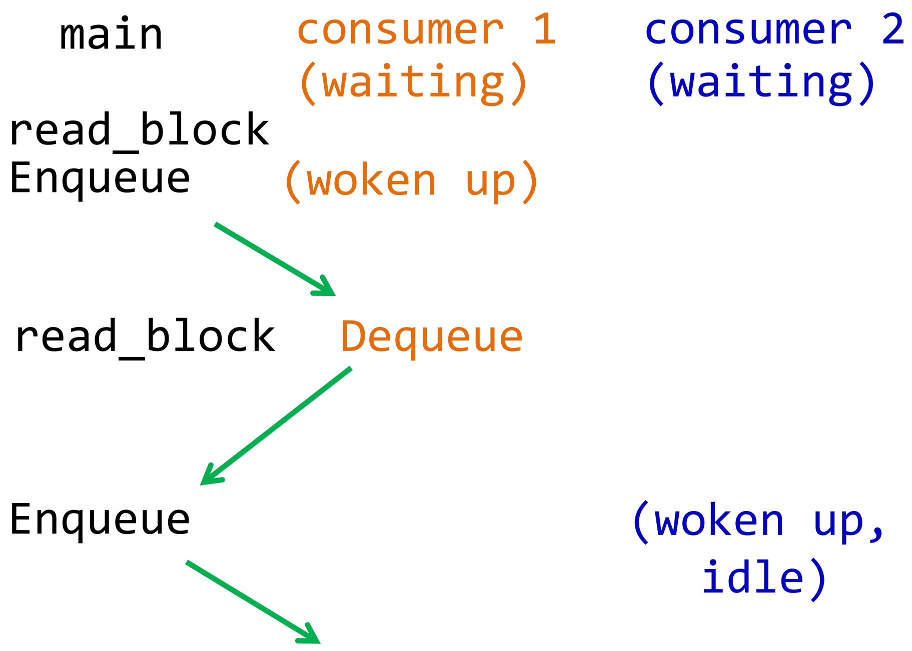
Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

(woken up)

read_block

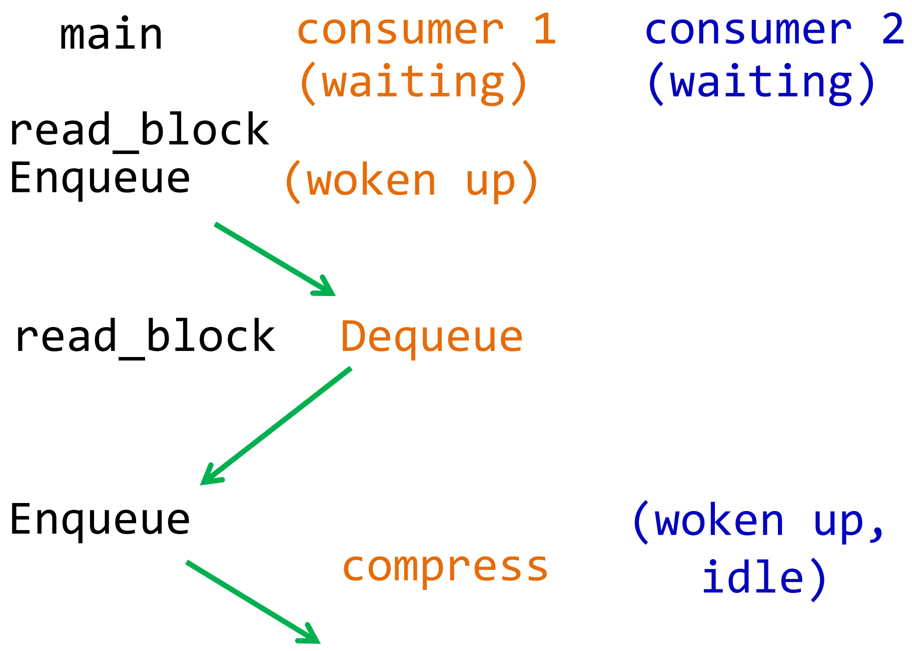
Dequeue

Enqueue

(woken up,
idle)

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

(woken up)

read_block

Dequeue

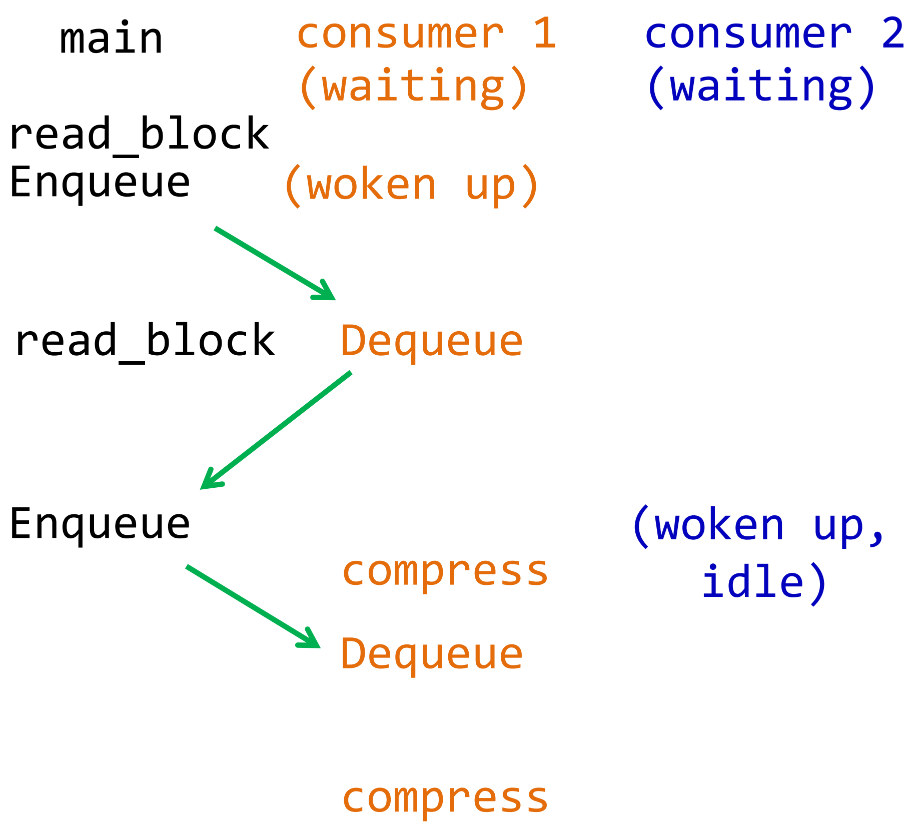
Enqueue

compress

(woken up,
idle)

Schedule ignoring load imbalance

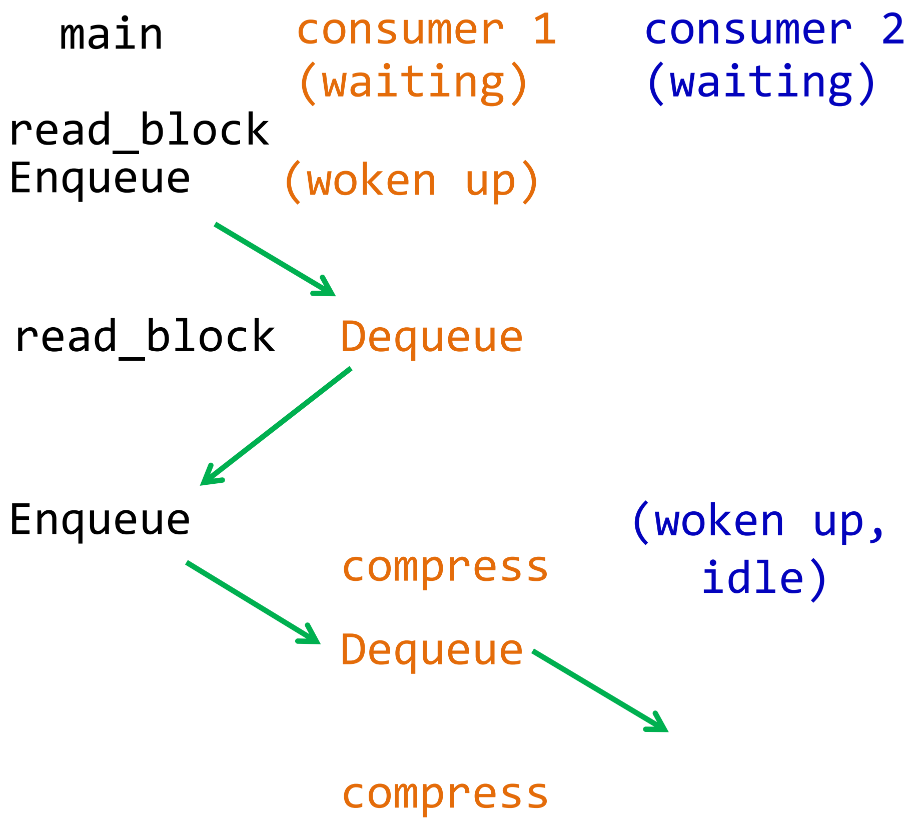
```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main	consumer 1 (waiting)	consumer 2 (waiting)
read_block Enqueue	(woken up)	
read_block	Dequeue	
Enqueue		(woken up, idle)
	compress	
	Dequeue	
	compress	

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1 (waiting)

consumer 2 (waiting)

read_block

Enqueue

(woken up)

read_block

Dequeue

Enqueue

compress

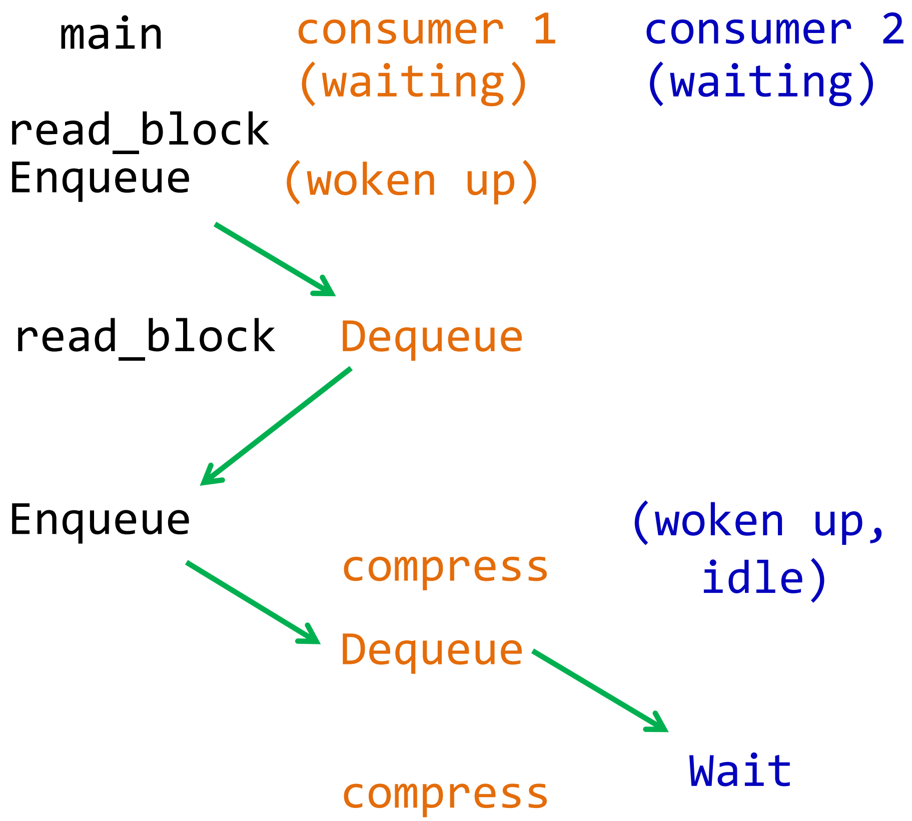
(woken up, idle)

Dequeue

compress

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Deque;
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

(woken up)

read_block

Dequeue

Enqueue

(woken up,
idle)

compress

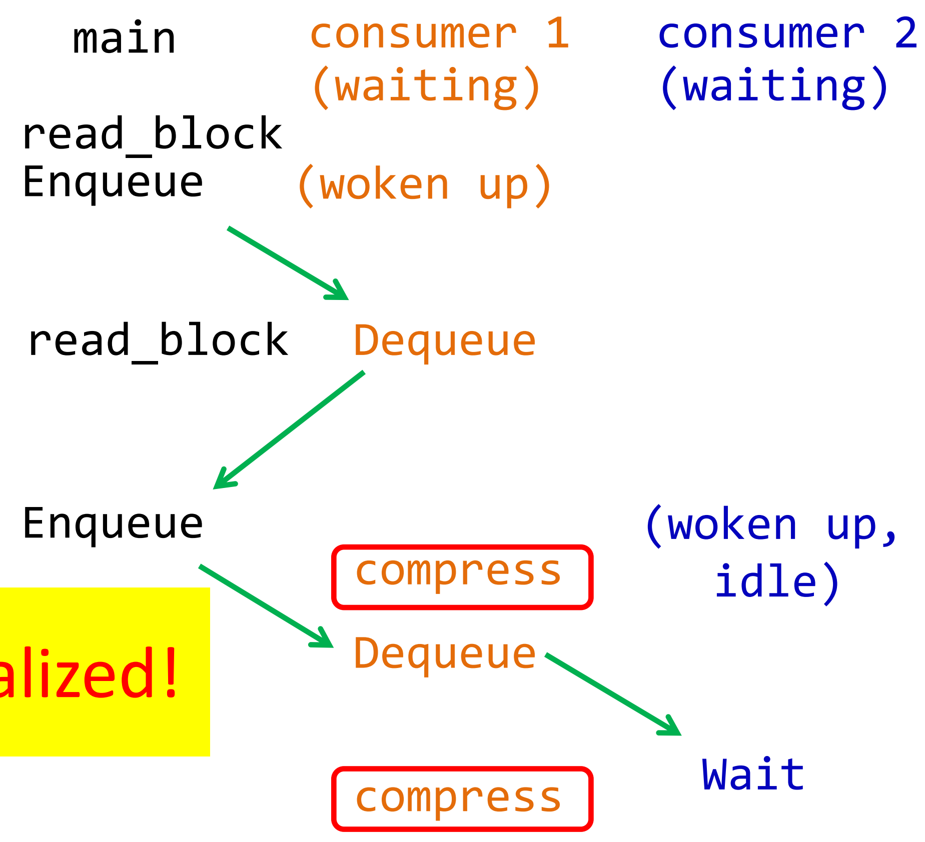
Dequeue

Wait

compress

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



main

consumer 1 (waiting)

consumer 2 (waiting)

read_block

Enqueue

(woken up)

read_block

Dequeue

Enqueue

compress

(woken up, idle)

Dequeue

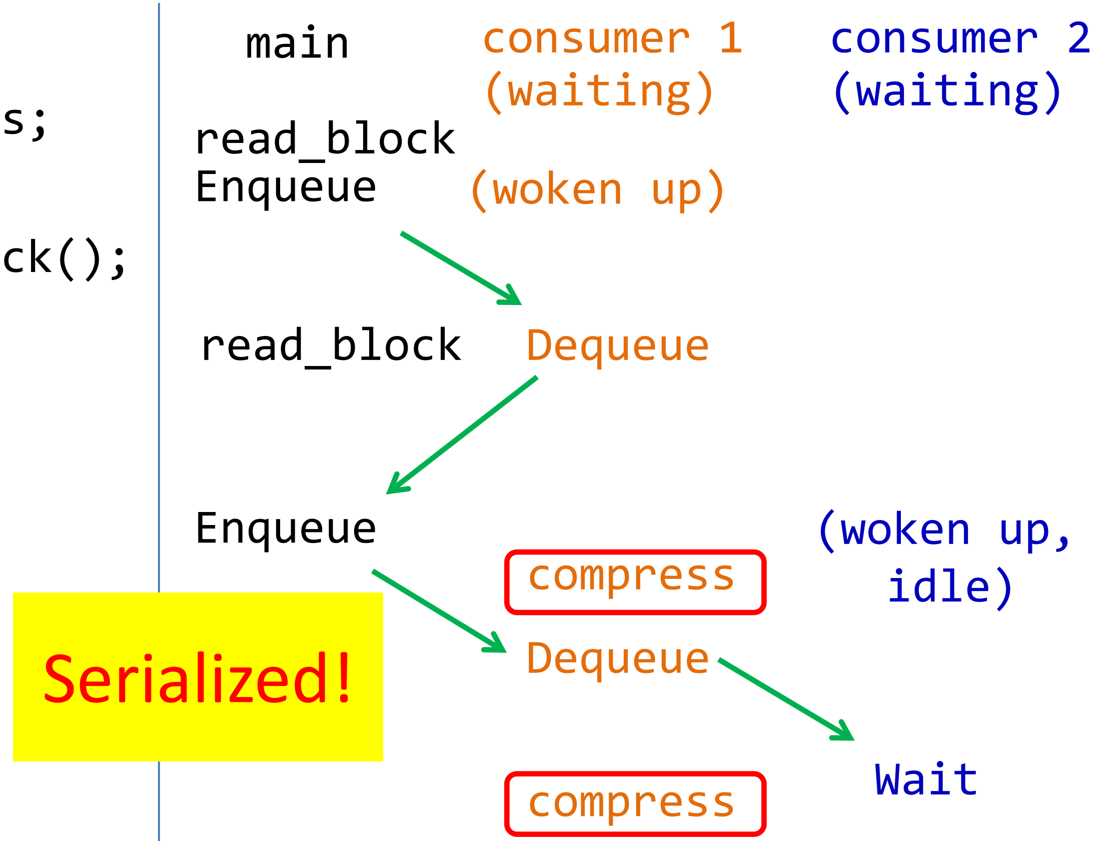
Wait

compress

Serialized!

Schedule ignoring load imbalance

```
main thread:
  create 2 consumer threads;
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue;
    compress(block);
  }
```



- Observed 770% overhead on 16 cores in prior system Dthreads [Dthreads SOSP 11]

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

main   consumer 1   consumer 2

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

main	consumer 1 (waiting)	consumer 2 (waiting)

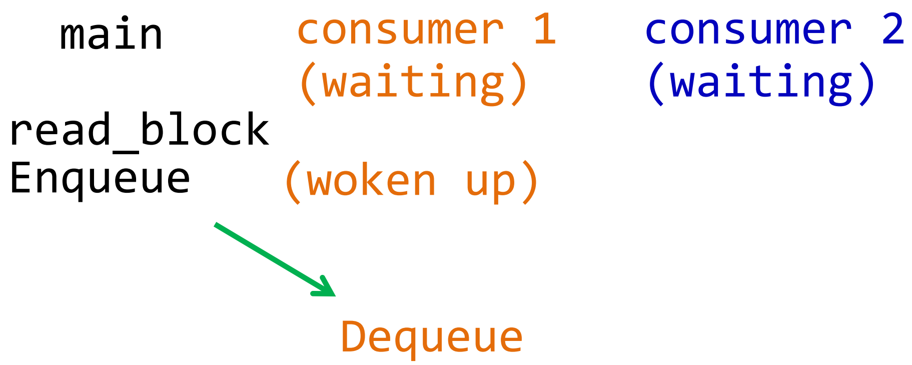
Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

main	consumer 1	consumer 2
	(waiting)	(waiting)
read_block		
Enqueue	(woken up)	

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

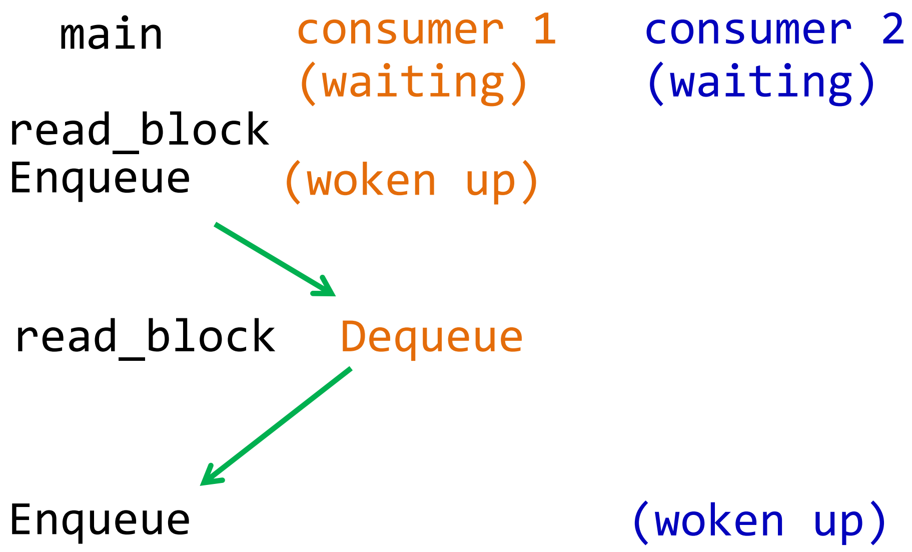
read_block
Enqueue

(woken up)

Dequeue

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main

consumer 1
(waiting)

consumer 2
(waiting)

read_block
Enqueue

(woken up)

read_block

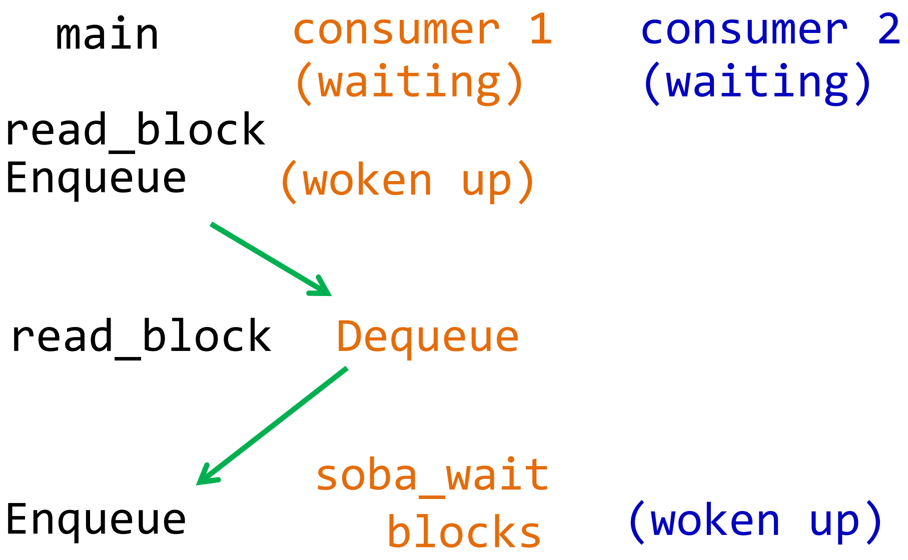
Dequeue

Enqueue

(woken up)

Parrot schedule with hints

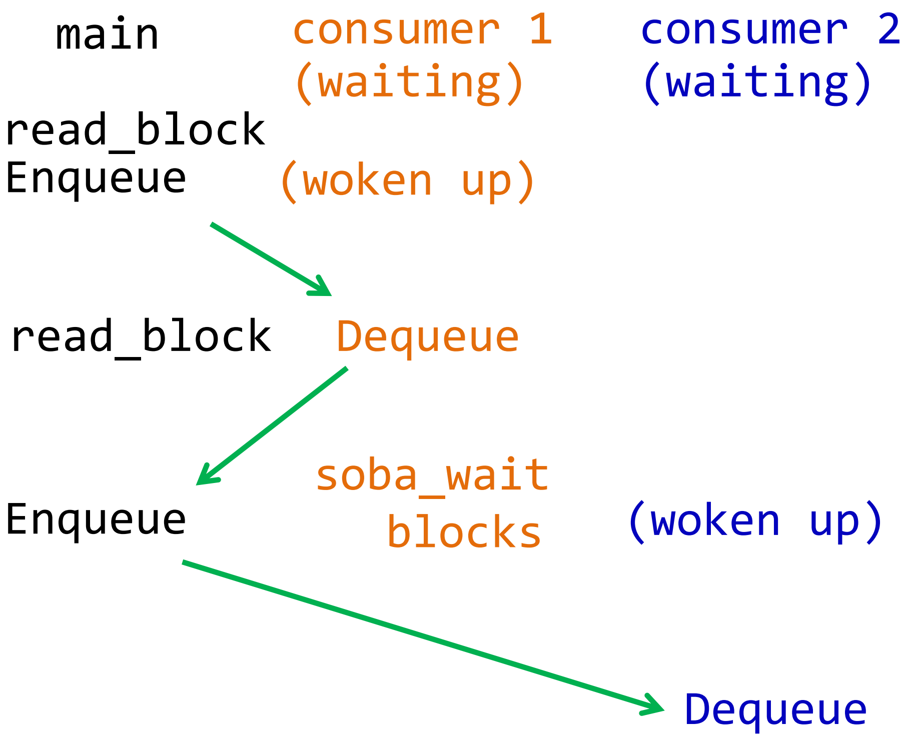
```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main	consumer 1	consumer 2
	(waiting)	(waiting)
read_block Enqueue	(woken up)
read_block	Dequeue
Enqueue	soba_wait blocks	(woken up)

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main

consumer 1 (waiting)

consumer 2 (waiting)

read_block

Enqueue

(woken up)

read_block

Dequeue

Enqueue

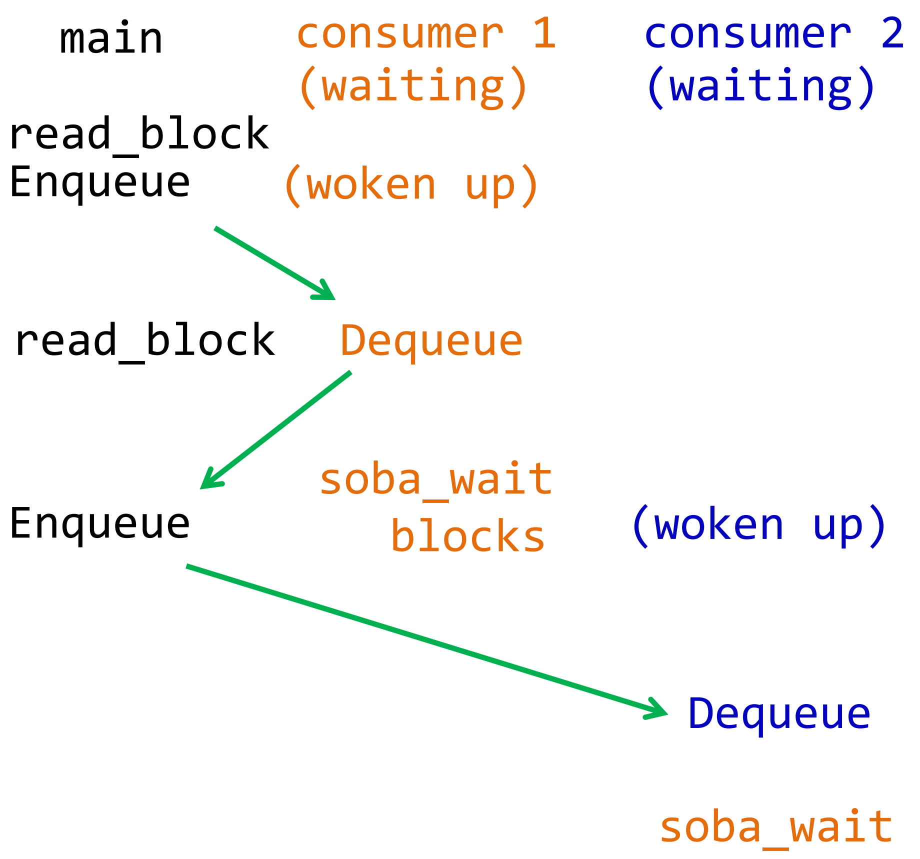
soba_wait blocks

(woken up)

Dequeue

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main

consumer 1 (waiting)

consumer 2 (waiting)

read_block

Enqueue

(woken up)

read_block

Dequeue

Enqueue

soba_wait blocks

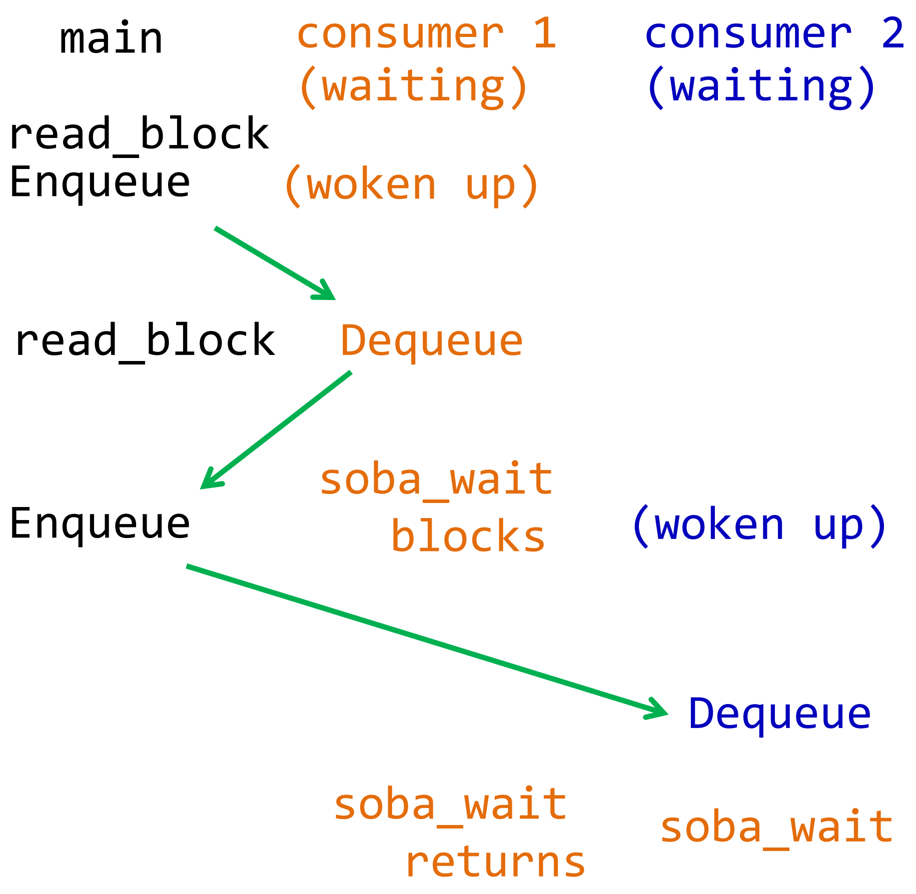
(woken up)

Dequeue

soba_wait

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main | consumer 1 (waiting) | consumer 2 (waiting)

read_block

Enqueue | (woken up)

read_block | Dequeue

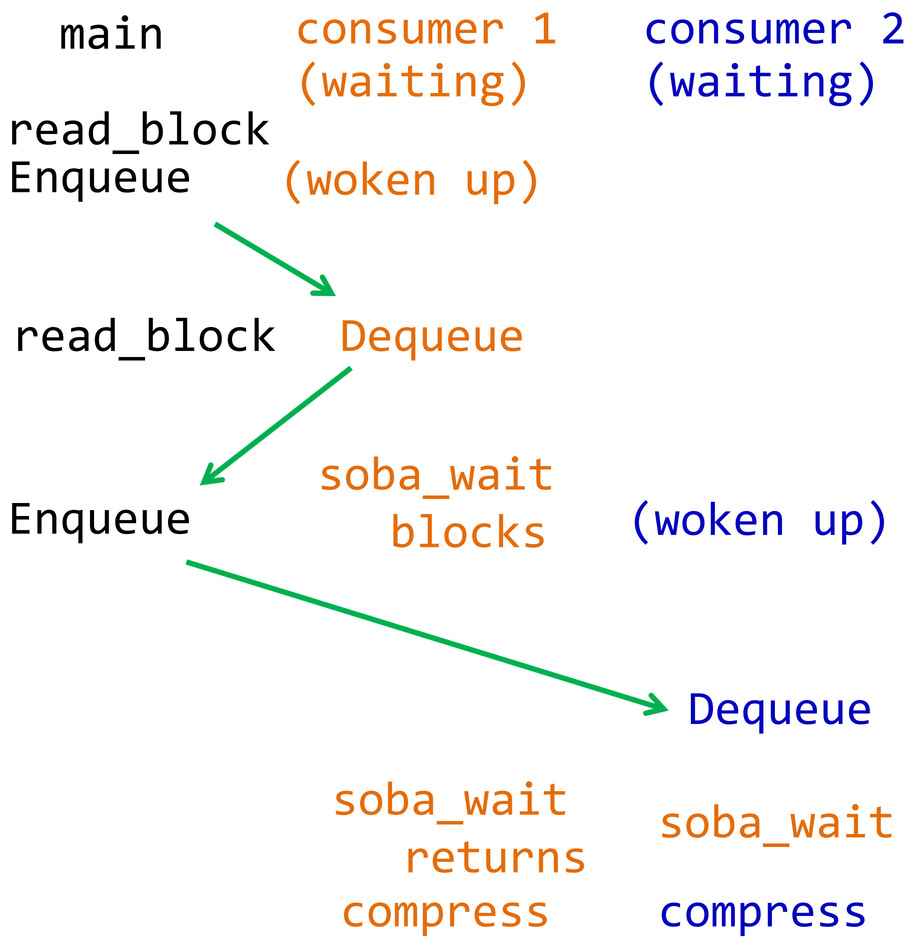
Enqueue | soba_wait blocks | (woken up)

Dequeue

soba_wait returns | soba_wait

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main
consumer 1 (waiting)
consumer 2 (waiting)

read_block
Enqueue
(woken up)

read_block
Dequeue

Enqueue
soba_wait blocks
(woken up)

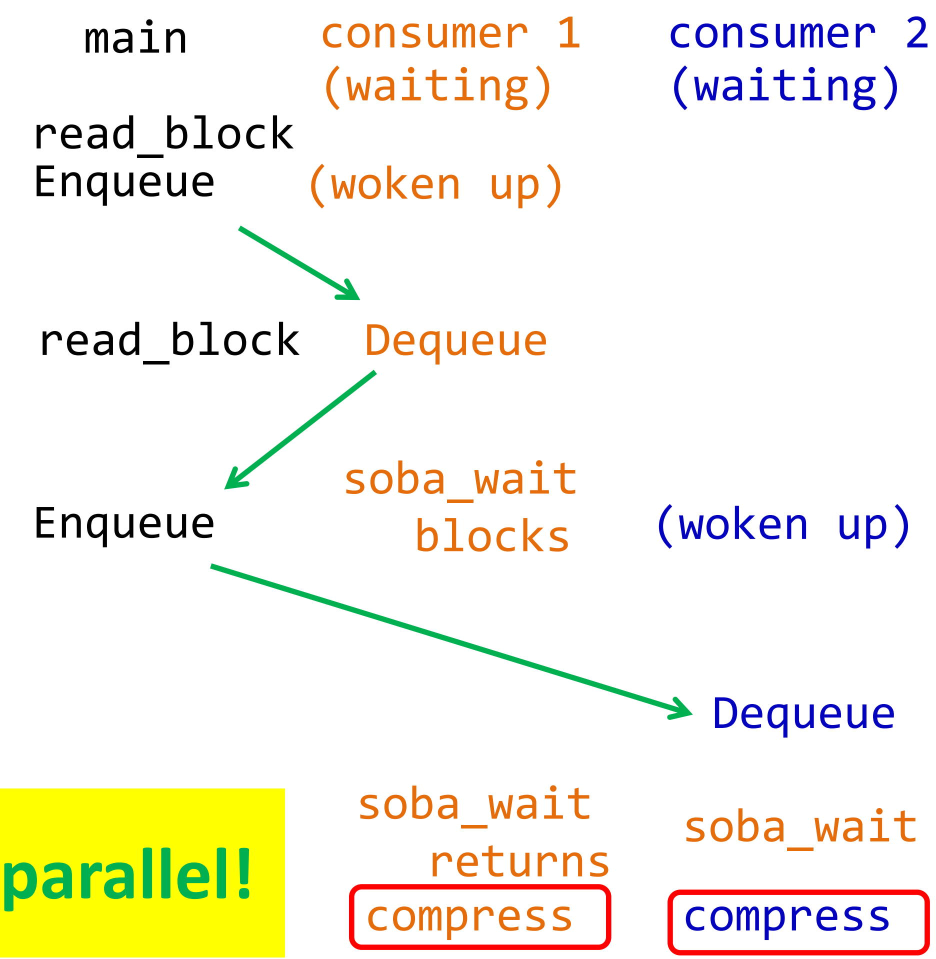
Dequeue

soba_wait returns
compress

soba_wait
compress

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```



main

consumer 1 (waiting)

consumer 2 (waiting)

read_block

Enqueue

(woken up)

read_block

Dequeue

Enqueue

soba_wait blocks

(woken up)

Dequeue

soba_wait returns

compress

soba_wait

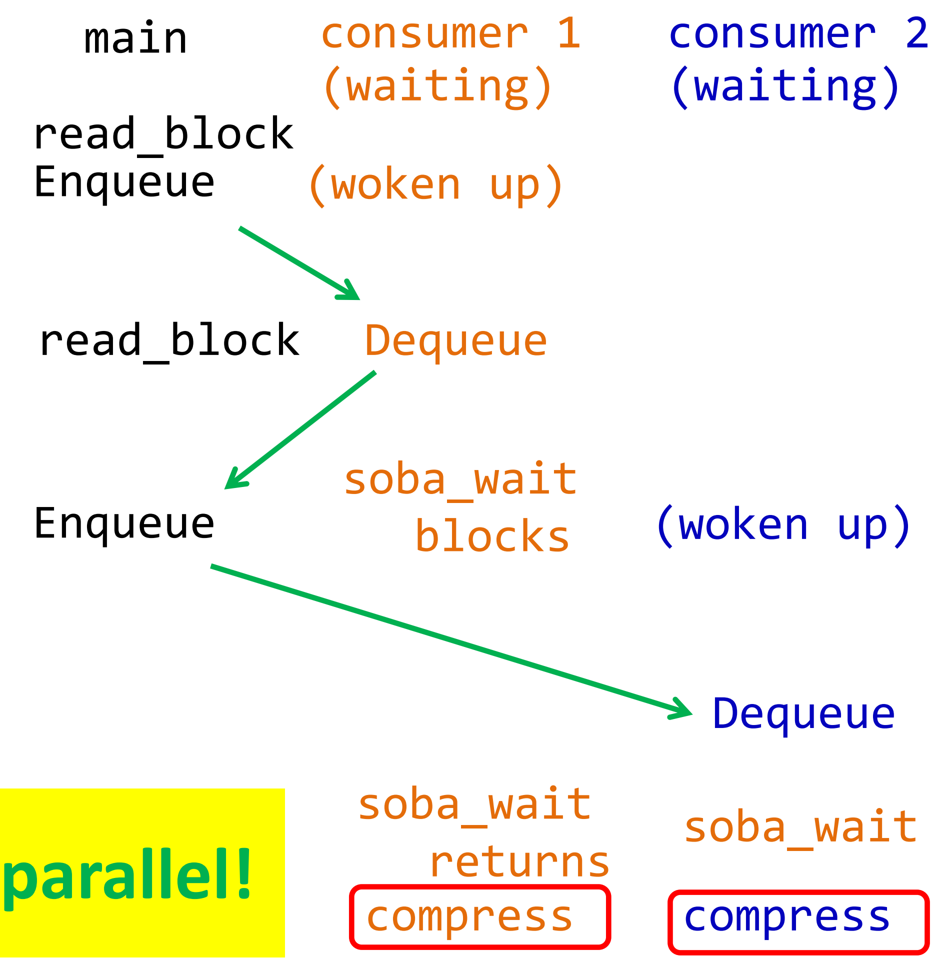
compress

Run in parallel!

Parrot schedule with hints

```
main thread:
  create 2 consumer threads;
  soba_init(2);
  for each file block {
    char *block = read_block();
    Enqueue;
  }

consumer thread:
  while(1) {
    Wait or Dequeue
    soba_wait();
    compress(block);
  }
```

Run in parallel!



main	consumer 1	consumer 2
	(waiting)	(waiting)
read_block Enqueue	(woken up)	
read_block	Dequeue	
Enqueue	soba_wait blocks	(woken up)
		Dequeue
	soba_wait returns compress	soba_wait compress

- 0.8% overhead

Performance hint API

```
// soft barrier; doesn’t increase # of schedules
void soba_init(int count, void *chan = NULL, int
  deterministic_timeout = 20);
void soba_wait(void *chan = NULL);

// performance critical section; increase # of
// schedules, but can check!
void pcs_enter();
void pcs_exit();
```

Evaluation questions

- How fast is Parrot?
- How easy is it to add hints?
- How much can Parrot improve reliability?

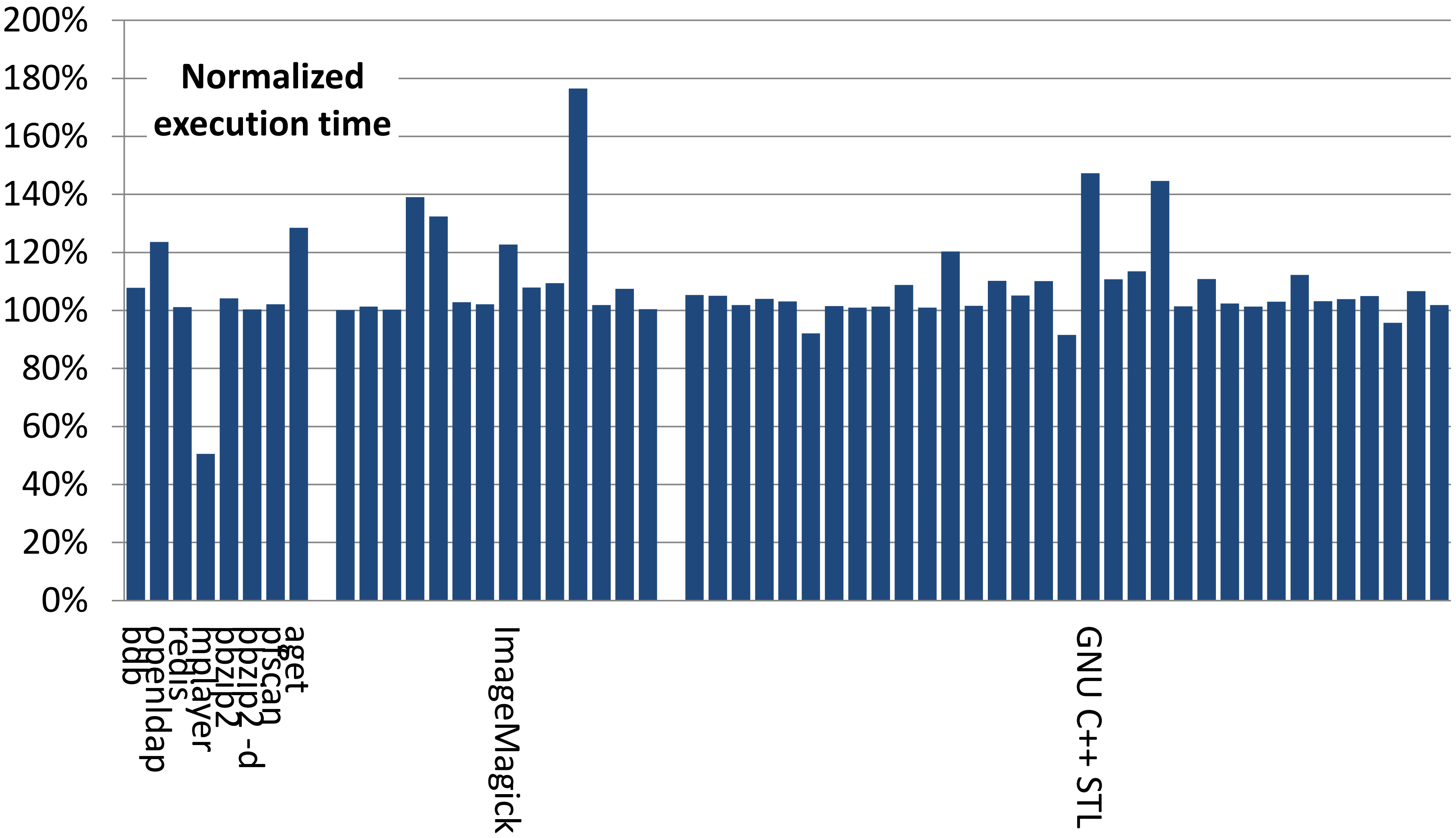
Evaluation questions

- How fast is Parrot?
- How easy is it to add hints?
- How much can Parrot improve reliability?

Evaluation Setup

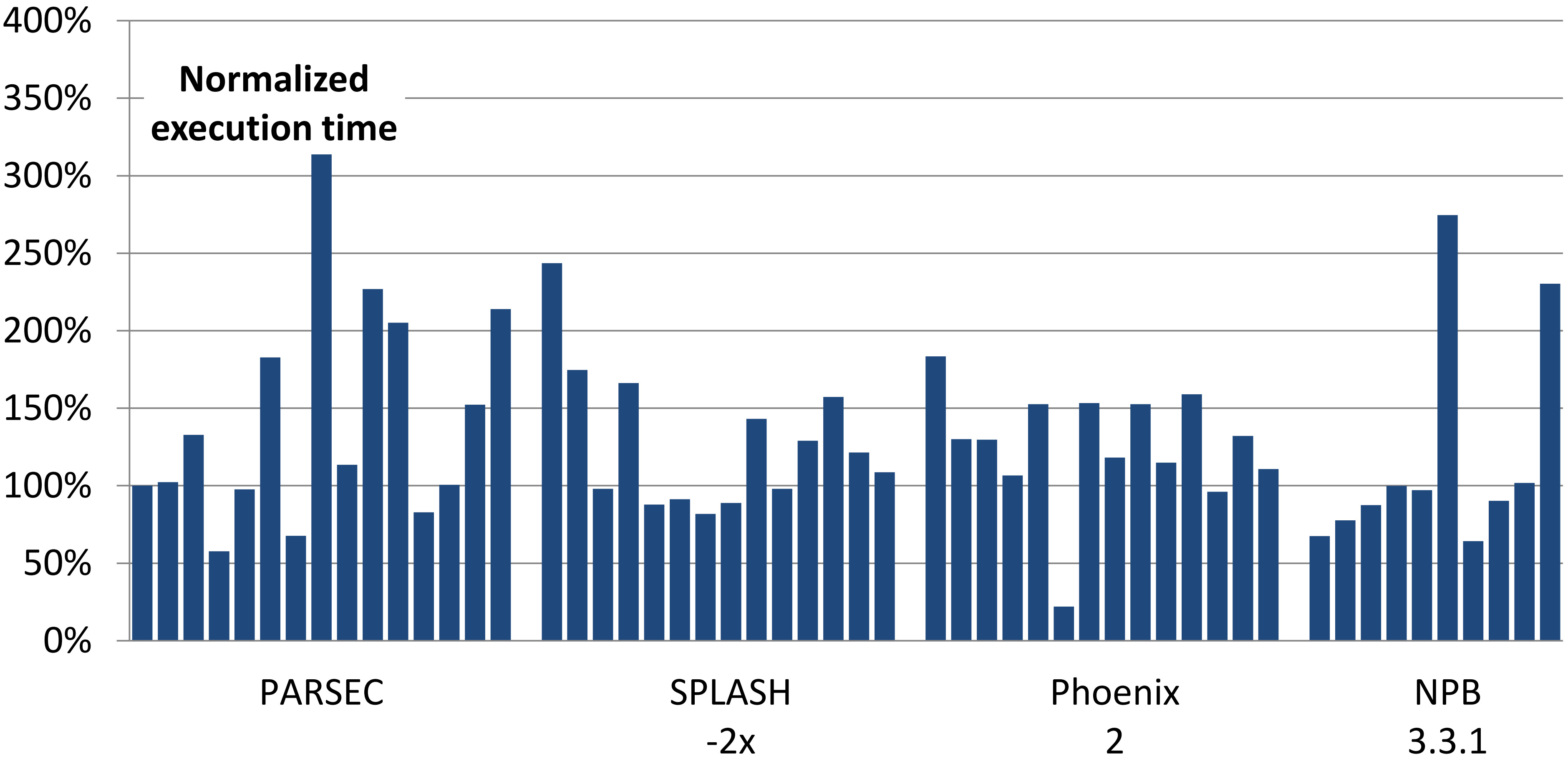
- A diverse set of of 108 programs
  - 55 Real-world programs: BerkeleyDB, OpenLDAP, Redis, MPlayer, ImageMagick, STL, PBZip2, pfscan, aget
  - 53 programs from 4 complete synthetic benchmark suites: PARSEC, SPLASH2X, PHOENIX, NPB.
  - Diverse: Pthreads, OpenMP, data partition, fork-join, pipeline, map-reduce, and workpile.
- Maximum allowed cores (24-core Xeon)
- Largest allowed or representative workloads

Overhead (real-world programs): small



- Mean overhead: 6.9% for real-world, 19.0% for synthetic, and 12.7% for all

Overhead (synthetic benchmarks): small



- Mean overhead: 6.9% for real-world, 19.0% for synthetic, and 12.7% for all

Evaluation questions

- How fast is Parrot?
- How easy is it to add hints?
- How much can Parrot improve reliability?

Hints: easy to add, effective

	# programs requiring hints	# lines of hints	Overhead w/o hints	Overhead w/ hints
Soft barrier	81	87	484%	9.0%
Performance critical section	9	22	830%	42.1%
Total	90	109	510%	11.9%

- Average to 1.2 lines per program
- A few hints in common libs benefit many programs
- 0.5--2 hours per program added by mostly MS students who didn't write the programs

Hints: easy to add, effective

	# programs requiring hints	# lines of hints	Overhead w/o hints	Overhead w/ hints
Soft barrier	81	87	484%	9.0%
Performance critical section	9	22	830%	42.1%
Total	90	109	510%	11.9%

- Average to 1.2 lines per program
- A few hints in common libs benefit many programs
- 0.5--2 hours per program added by mostly MS students who didn't write the programs

Hints: easy to add, effective

	# programs requiring hints	# lines of hints	Overhead w/o hints	Overhead w/ hints
Soft barrier	81	87	484%	9.0%
Performance critical section	9	22	830%	42.1%
Total	90	109	510%	11.9%

- Average to 1.2 lines per program
- A few hints in common libs benefit many programs
- 0.5--2 hours per program added by mostly MS students who didn’t write the programs

Hints: easy to add, effective

	# programs requiring hints	# lines of hints	Overhead w/o hints	Overhead w/ hints
Soft barrier	81	87	484%	9.0%
Performance critical section	9	22	830%	42.1%
Total	90	109	510%	11.9%

- Average to 1.2 lines per program
- A few hints in common libs benefit many programs
- 0.5--2 hours per program added by mostly MS students who didn't write the programs

Hints: easy to add, effective

	# programs requiring hints	# lines of hints	Overhead w/o hints	Overhead w/ hints
Soft barrier	81	87	484%	9.0%
Performance critical section	9	22	830%	42.1%
Total	90	109	510%	11.9%

- Average to 1.2 lines per program
- A few hints in common libs benefit many programs
- 0.5--2 hours per program added by mostly MS students who didn’t write the programs

Evaluation questions

- How fast is Parrot?
- How easy is it to add hints?
- How much can Parrot improve reliability?

Model checking: higher coverage

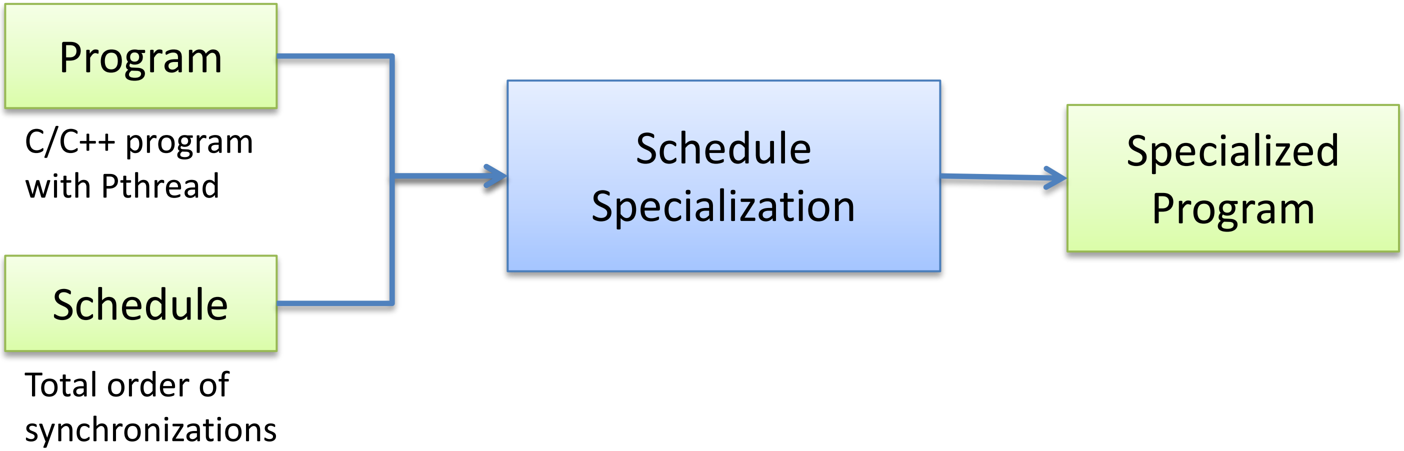
- Integrated Parrot with dBug [dBug SPIN 11] because it's open-source, runs on Linux, implements dynamic partial order reduction [DPOR POPL 05], can estimate number of possible schedules [Knuth]

- Parrot increases coverage by 10^{6}---10^{19734} (not a typo ;) for 53 programs
- Parrot increases number of verified programs from 43 to 99

Static analysis: more precise

[Specialization PLDI 12]

- Specialize a program according to a schedule
- Resultant program contains schedule info, improving precision of stock analysis



Static Race Detector

# of False Positives

Program	w/o StableMT	w/ StableMT
aget	72	0
PBZip2	125	0
fft	96	0
blackscholes	3	0
swaptions	165	0
streamcluster	4	0
canneal	21	0
bodytrack	4	0
ferret	6	0
raytrace	215	0
cholesky	31	7
radix	53	14
water-spatial	2447	1799
lu-contig	18	18
barnes	370	369
water-nsquared	354	333
ocean	331	292

Static Race Detector

# of False Positives

Program	w/o StableMT	w/ StableMT
aget	72	0
PBZip2	125	0
fft	96	0
blackscholes	3	0
swaptions	165	0
streamcluster	4	0
canneal	21	0
bodytrack	4	0
ferret	6	0
raytrace	215	0
cholesky	31	7
radix	53	14
water-spatial	2447	1799
lu-contig	18	18
barnes	370	369
water-nsquared	354	333
ocean	331	292

Static Race Detector

# of False Positives

Program	w/o StableMT	w/ StableMT
aget	72	0
PBZip2	125	0
fft	96	0
blackscholes	3	0
swaptions	165	0
streamcluster	4	0
canneal	21	0
bodytrack	4	0
ferret	6	0
raytrace	215	0
cholesky	31	7
radix	53	14
water-spatial	2447	1799
lu-contig	18	18
barnes	370	369
water-nsquared	354	333
ocean	331	292

Static Race Detector

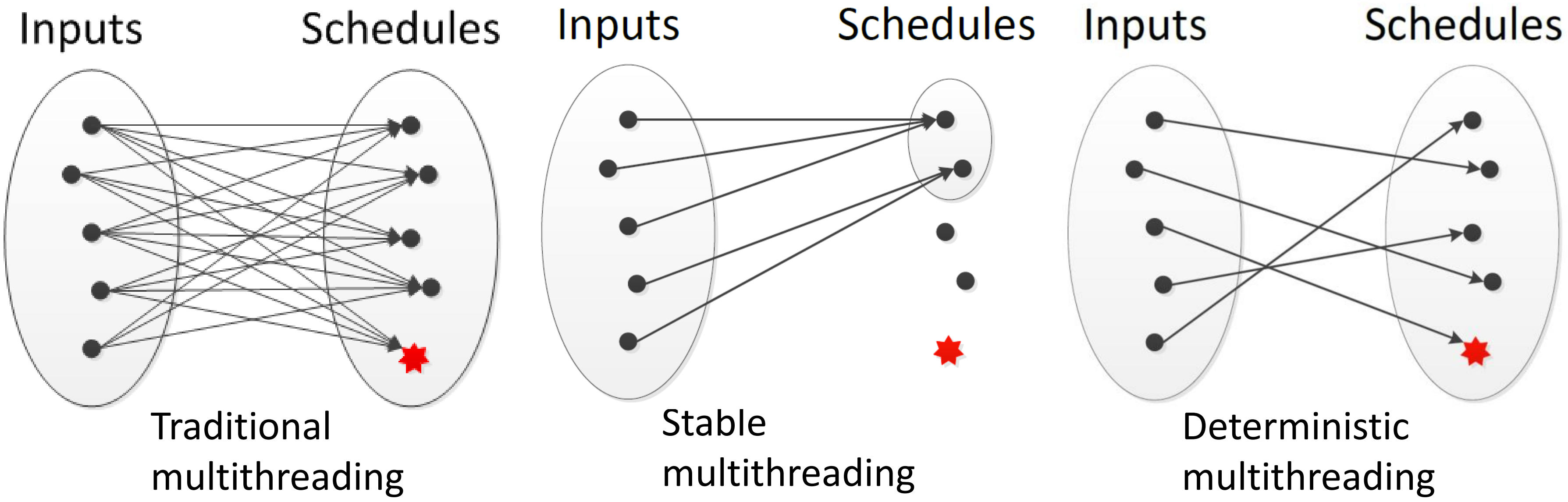
# of False Positives

Program	w/o StableMT	w/ StableMT
aget	72	0
PBZip2	125	0
fft	96	0
blackscholes	3	0
swaptions	165	0
streamcluster	4	0
canneal	21	0
bodytrack	4	0
ferret	6	0
raytrace	215	0
cholesky	31	7
radix	53	14
water-spatial	2447	1799
lu-contig	18	18
barnes	370	369
water-nsquared	354	333
ocean	331	292

Previously Unknown Harmful Races Detected

- 4 in aget
- 2 in radix
- 1 in fft

Conclusion



- Root cause of the multithreading difficulties: ~~nondeterminism~~ too many schedules
- *Stable Multithreading (StableMT)*: a radical approach to vastly reducing schedules for reliability with low overhead [Tern OSDI 10] [Peregrine SOSP 11] [Specialization PLDI 12] [Parrot SOSP 13] [HotPar 13] [CACM 14]