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
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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mutex_lock(M)</code></td>
<td><code>mutex_lock(M)</code></td>
</tr>
<tr>
<td><code>*obj = ...</code></td>
<td><code>free(obj)</code></td>
</tr>
<tr>
<td><code>mutex_unlock(M)</code></td>
<td><code>mutex_unlock(M)</code></td>
</tr>
</tbody>
</table>

Apache Bug #21287 (simplified)
Concurrency bug example

Apache Bug #21287 (simplified)
Concurrence bug example

Thread 0
mutex_lock(M)
*obj = ...
mutex_unlock(M)

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

Input: everything a program reads from environment
– E.g., main() arguments, data read from file or socket
Concurrent bug example

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock(M)</td>
<td>mutex_lock(M)</td>
<td>mutex_lock(M)</td>
<td>mutex_lock(M)</td>
</tr>
<tr>
<td>*obj = ...</td>
<td>*obj = ...</td>
<td>free(obj)</td>
<td>free(obj)</td>
</tr>
<tr>
<td>mutex_unlock(M)</td>
<td>mutex_unlock(M)</td>
<td>mutex_unlock(M)</td>
<td>mutex_unlock(M)</td>
</tr>
</tbody>
</table>

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

- **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**?

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

<table>
<thead>
<tr>
<th>K critical sections</th>
<th>M threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock() ... unlock()</td>
<td>lock() ... unlock()</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>lock() ... unlock()</td>
<td>lock() ... unlock()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$$\text{lock()} \ldots \text{unlock()}$$
Why **hard**?

<table>
<thead>
<tr>
<th>lock()</th>
<th>lock()</th>
<th>lock()</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>unlock()</td>
<td>unlock()</td>
<td>unlock()</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>lock()</td>
<td>lock()</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>unlock()</td>
<td>unlock()</td>
<td>unlock()</td>
</tr>
</tbody>
</table>

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

$M!$ schedules

$M$ threads

$K$ critical sections
Why hard?

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

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

$$\geq (M!)^K$$

**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

• Coverage = Checked/All
How to improve checking coverage?

- Coverage = Checked/All
- Traditionally: enlarge Checked exploiting equivalence
How to improve checking coverage?

- 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
  - \[\text{DIR SOSP 11}\] (joint w/ MSR Asia) took us 3 years
  - First after \[\text{VeriSoft POPL 97}\]

All possible runtime schedules

Checked schedules
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]

Can we increase coverage without enlarging Checked?
How to improve checking coverage?

All possible runtime schedules

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

Can we increase coverage without enlarging Checked?
High coverage with StableMT

- Enforce round-robin synchronization order

```
lock() lock() lock()
... ... ...
unlock() unlock() unlock()
. . .
. . .
. . .
lock() lock() ...
... ...
unlock() unlock() ...
```
High coverage with StableMT

- Enforce round-robin synchronization order
High coverage with StableMT

- Enforce round-robin synchronization order

Finding concurrency bugs == checking one schedule
High coverage with StableMT

- Enforce round-robin synchronization order

Finding concurrency bugs

\[ \text{K critical sections} \]

\[ \text{== checking one schedule} \]

\[ \text{Simple enough that it feels like cheating } \vphantom{\text{Smile}} \]
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

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

- All inputs \( \rightarrow \) a greatly reduced set of schedules

Key benefits:
- Vastly shrink the haystack \( \rightarrow \) needles much easier to find
- Provide anticipated *stability* (robustness against input or program perturbations)

"What you check is what you run"
"What you can’t check is not run"

Tool + runtime co-design
Stability and determinism are two separate, complementary properties.

Stability is more useful for reliability.
Deterministic multithreading (DMT): one input $\rightarrow$ one schedule
Deterministic multithreading (DMT): one input ➔ one schedule

Traditional multithreading

Stable multithreading

Deterministic multithreading
Deterministic multithreading (DMT): one input $\Rightarrow$ one schedule

- One testing execution validates all future executions on the same input
- Reproducing a concurrency bug requires only the input
Deterministic multithreading (DMT): one input $\Rightarrow$ one schedule

- 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()  ...
...
unlock() unlock() unlock()
Input or program perturbation ➔
different schedules

K critical sections

M threads

lock()  lock()  lock()
...
unlock()  unlock()  unlock()
.
.
.
lock()  lock()  ...
...
unlock()  unlock()  unlock()
Input or program perturbation ➔
different schedules

Input 1

K critical sections

M threads
Input or program perturbation →
different schedules

K critical sections

M threads

Input 1
Input 2

lock() → lock() → lock() →...
unlock() → unlock() → unlock() → ...

... → ... → ...

lock() → lock() → ...
unlock() → unlock() → ...

lock() →...
unlock() →...
Input or program perturbation ➞

different schedules

K critical sections

M threads

lock()  lock()  lock()
...
unlock()  unlock()  unlock()
...
lock()  lock()  lock()
...
unlock()  unlock()  unlock()
Input or program perturbation ➔ different schedules

K critical sections

M threads

Input 1
Input 2
...

lock() → lock() → lock() → unlock() → unlock() → unlock() → lock() → ...

...
Input or program perturbation ➔

different schedules

Still too many schedules
Unstable!
Deterministic but not stable

- Determinism is a **narrow** property
  - Same input + same program $\implies$ same behavior
  - Input or program changes slightly? Can be **unstable**
Deterministic but not stable

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

Traditional multithreading

Stable, deterministic multithreading

Deterministic, unstable multithreading

Determinism and stability are often mistakenly conflated

- Input or program changes slightly? Can be unstable
Stable but not deterministic

• Determinism is a **binary** property
  – Nondeterministic if one input ⇒ $n > 1$ schedules
Stable but not deterministic

- Determinism is a binary property
  - Nondeterministic if one input $\Rightarrow n > 1$ schedules
Stable but not deterministic

• Determinism is a binary property
  - Nondeterministic if one input \( n > 1 \) schedules

Traditional multithreading

Stable, nondeterministic multithreading

Deterministic multithreading

Nondeterministic but stable \( \Rightarrow \) 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
Key challenge: how to compute the schedules to map inputs to

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

lock()
unlock()
...
lock()
unlock()
Key challenge: how to compute the schedules to map inputs to

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

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

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

```
lock()
unlock()
...
lock()
unlock()
comp(...)
lock()
unlock()
comp(...)
```
Our 1\textsuperscript{st} 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 1\textsuperscript{st} attempt: \textit{record and reuse} synchronization schedules

• On new input, run program as is to record reasonably fast synchronization schedule

• Compute relaxed, quickly checkable \textit{precondition} of the schedule to capture dependencies on input

• Reuse schedule on inputs satisfying \textit{precondition}
Our 1\textsuperscript{st} attempt: \textit{record and reuse synchronization schedules}

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

```java
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

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

if(y == 1)
    ...; // no synch
else
    ...; // no synch
```
Our 1\textsuperscript{st} attempt: record and reuse synchronization schedules

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

\begin{verbatim}
if(x == 1) {
    lock();
    unlock();
} else
    ...; // no synch

if(y == 1)
    ...; // no synch
else
    ...; // no synch
\end{verbatim}

Precondition should constrain \textit{x}, but \textbf{not} \textit{y}
Our 1\textsuperscript{st} attempt: record and reuse synchronization schedules

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

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

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

Solution: \textit{symbolic execution} to track constraints and \textit{precondition slicing} to remove unnecessary constraints

Precondition should constrain x, but \textbf{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

```c
x = 1;
if(x) {
  lock();
  unlock();
}
x = 0;
```
The problem of data races

• May cause execution to deviate from schedule

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

x = 0;
```
The problem of data races

- May cause execution to deviate from schedule

```c
x = 1;
if (x) {
    lock();
    unlock();
}
x = 0;
```
The problem of data races

- May cause execution to deviate from schedule

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

```c
a[x] = 1;
a[x] = 0;
if(a[y]) {
    lock();
    unlock();
}
```
The problem of data races

- May cause execution to deviate from schedule

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

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

a[x] = 1;
a[x] = 0;
if(a[y]) {
    lock();
    unlock();
}
```
The problem of data races

- May cause execution to deviate from schedule

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

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

Our 1\textsuperscript{st} 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;
  }

c 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;
}

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

// 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);
  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);
  }
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);
}
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 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 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);
}
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
read_block          (waiting)             (waiting)
  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    consumer 2
read_block (waiting) (waiting)
Enqueue   (woken up)
read_block  Dequeue
Enqueue                     (woken up, idle)
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 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
create 2 consumer threads;
for each file block {
    char *block = read_block();
    Enqueue;
}

consumer 1 (waiting)
read_block
Enqueue (woken up)
compress
Dequeue
Enqueue
compress
Dequeue
(compressed)

consumer 2 (waiting)
read_block
Dequeue
wait
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);
}

- 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;

\textbf{soba_init(2);}\
f for each file block {
\textbf{char *block = read_block();}\n\textbf{Enqueue;}
}

consumer thread:

while(1) {
\textbf{Wait or Dequeue}\
\textbf{soba_wait();}\ncom press(block);\n}

107
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;
  }

c 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;
\textbf{soba} \textit{init}(2);
for each file block {
\begin{verbatim}
  char *block = read_block();
  Enqueue;
\end{verbatim}
}

consumer thread:
while(1) {
\begin{verbatim}
  Wait or Dequeue
  \textbf{soba} \textit{wait}();
  compress(block);
\end{verbatim}
}
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;
\texttt{soba\_init}(2);
for each file block {
    char *block = \texttt{read\_block}();
    Enqueue;
}

consumer thread:
while(1) {
    Wait or Dequeue \texttt{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;
\texttt{soba\_init(2);}
for each file block {
  char *block = read\_block();
  Enqueue;
}

consumer thread:
while(1) {
  Wait or Dequeue
  \texttt{soba\_wait()};
  compress(block);
}
Parrot schedule with hints

main thread:
create 2 consumer threads;
\texttt{soba\_init(2);} 
for each file block {
   char *block = read\_block();
   Enqueue;
}

consumer thread:
while(1) {
   Wait or Dequeue \texttt{soba\_wait()};
   compress(block);
}
main thread:
create 2 consumer threads;
\textbf{soba\_init}(2);
for each file block {
    char *block = read\_block();
    Enqueue;
}

consumer thread:
while(1) {
    Wait or Dequeue \textbf{soba\_wait}();
    compress(block);
}
Parrot schedule with hints

main thread:
create 2 consumer threads;
\textbf{soba\_init}(2);
for each file block {
  char *\textbf{block} = \textbf{read\_block}();
  Enqueue;
}

consumer thread:
while(1) {
  Wait or \textbf{Dequeue}
  \textbf{soba\_wait}();
  compress(\textbf{block});
}
main thread:
create 2 consumer threads;
\textbf{\texttt{soba\_init}(2)};
for each file block {
cchar *block = \texttt{read\_block}();
Enqueue;
}

consumer thread:
while(1) {
Wait or Dequeue
\textbf{\texttt{soba\_wait}();}
compress(block);
}

Run in parallel!

- 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 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

<table>
<thead>
<tr>
<th></th>
<th># programs requiring hints</th>
<th># lines of hints</th>
<th>Overhead w/o hints</th>
<th>Overhead w/ hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft barrier</td>
<td>81</td>
<td>87</td>
<td>484%</td>
<td>9.0%</td>
</tr>
<tr>
<td>Performance critical section</td>
<td>9</td>
<td>22</td>
<td>830%</td>
<td>42.1%</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td>109</td>
<td>510%</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th></th>
<th># programs requiring hints</th>
<th># lines of hints</th>
<th>Overhead w/o hints</th>
<th>Overhead w/ hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft barrier</td>
<td>81</td>
<td>87</td>
<td>484%</td>
<td>9.0%</td>
</tr>
<tr>
<td>Performance critical section</td>
<td>9</td>
<td>22</td>
<td>830%</td>
<td>42.1%</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td>109</td>
<td>510%</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th></th>
<th># programs requiring hints</th>
<th># lines of hints</th>
<th>Overhead w/o hints</th>
<th>Overhead w/ hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft barrier</td>
<td>81</td>
<td>87</td>
<td>484%</td>
<td>9.0%</td>
</tr>
<tr>
<td>Performance critical section</td>
<td>9</td>
<td>22</td>
<td>830%</td>
<td>42.1%</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td>109</td>
<td>510%</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th></th>
<th># programs requiring hints</th>
<th># lines of hints</th>
<th>Overhead w/o hints</th>
<th>Overhead w/ hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft barrier</td>
<td>81</td>
<td>87</td>
<td>484%</td>
<td>9.0%</td>
</tr>
<tr>
<td>Performance critical section</td>
<td>9</td>
<td>22</td>
<td>830%</td>
<td>42.1%</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td>109</td>
<td>510%</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th></th>
<th># programs requiring hints</th>
<th># lines of hints</th>
<th>Overhead w/o hints</th>
<th>Overhead w/ hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft barrier</td>
<td>81</td>
<td>87</td>
<td>484%</td>
<td>9.0%</td>
</tr>
<tr>
<td>Performance critical section</td>
<td>9</td>
<td>22</td>
<td>830%</td>
<td>42.1%</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td>109</td>
<td>510%</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

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