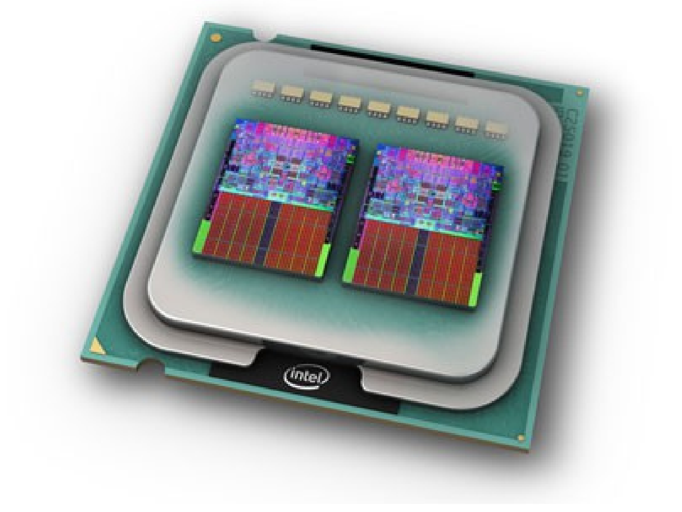
Kernel concurrency bugs

CSE 451, 2015

Pedro Fonseca

Concurrency bugs

- Depend on the interleaving of instructions
  - Non-deterministic
- Hard to avoid concurrency in the multi-core era
  - Finer grained locking
  - New algorithms
- Can have serious consequences
  - Therac-25 accident



Concurrency bug example

```
int x = 0;

void threadA(){
   A = x + 1;
   x = A;
}

void threadB(){
   B = x + 1;
   x = B;
}
```

Concurrency bug example

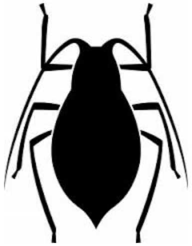
```
int x = 0;

void threadA(){
    A = x + 1;
    x = A;
}

void threadB(){
    B = x + 1;
    x = B;
}
```

1

```
A = x + 1
                B = x + 1
                x = B
x = A
```



Concurrency bug example

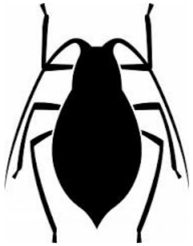
```
int x = 0;

void threadA(){
   A = x + 1;
   x = A;
}

void threadB(){
   B = x + 1;
   x = B;
}
```

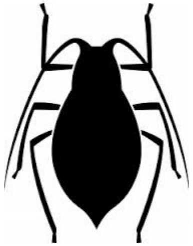
1

```
A = x + 1
                 B = x + 1
                 x = B
x = A
```



2

```
A = x + 1
                 B = x + 1
x = A
                 x = B
```



Concurrency bug example

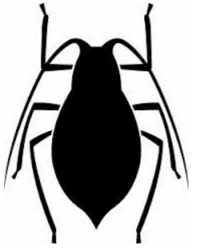
```
int x = 0;

void threadA(){
    A = x + 1;
    x = A;
}

void threadB(){
    B = x + 1;
    x = B;
}
```

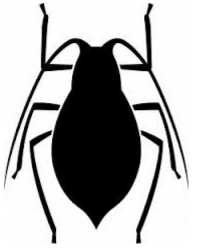
1

A = *x* + 1	
	B = *x* + 1
	x = B
x = A	



2

A = *x* + 1	
	B = *x* + 1
x = A	
	x = B



3

A = *x* + 1	
x = A	
	B = *x* + 1
	x = B

Concurrency bug example

How many interleavings?

```
int x = 0;

void threadA(){
    A = x + 1;
    x = A;
}

void threadB(){
    B = x + 1;
    x = B;
}
```

1

```
A = x + 1
                B = x + 1
                x = B
x = A
```


2

```
A = x + 1
                B = x + 1
x = A
                x = B
```


3

```
A = x + 1
x = A
                B = x + 1
                x = B
```

Concurrency bug example

How many interleavings?

$$\frac{(a + b)!}{a! \times b!}$$

a,b → number of instructions

```
int x = 0;

void threadA(){
    A = x + 1;
    x = A;
}

void threadB(){
    B = x + 1;
    x = B;
}
```

1

```
A = x + 1
                B = x + 1
                x = B
x = A
```


2

```
A = x + 1
                B = x + 1
x = A
                x = B
```


3

```
A = x + 1
x = A
                B = x + 1
                x = B
```

Concurrency bug example

How many interleavings?

$$\frac{(a + b)!}{a! \times b!}$$

a,b → number of instructions

a=b=1	**2**
a=b=2	**6**
a=b=3	**20**
a=b=4	**70**
a=b=5	**252**
a=b=6	**924**
a=b=7	**3432**
a=b=8	**12870**
a=b=9	**48620**
a=b=10	**184756**
a=b=11	**705432**
a=b=12	**2704156**
a=b=13	**10400600**
a=b=14	**40116600**
a=b=15	**155117520**
a=b=16	**601080390**
a=b=17	**2333606220**
a=b=18	**9075135300**
a=b=19	**35345263800**
a=b=20	**137846528820**

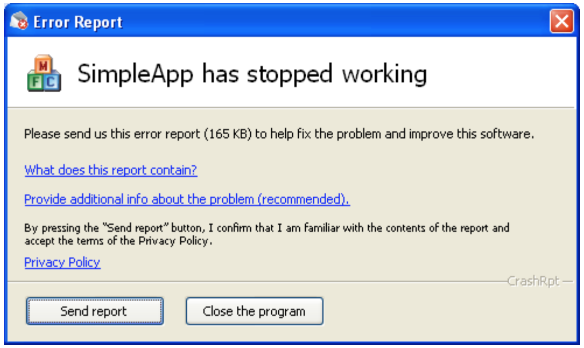
Too many!!

Only a subset of executions exposes

- Often the subset of executions is **really tiny**
  - Concurrency bugs may go unnoticed for years!
- Small environment differences may expose them more easily
  - Different kernels, different libraries, different workloads, different hardware
  - Bugs may never show up during testing and always show up on users computers!
- Non-determinism can be really painful!

Concurrency bugs can have ~~m~~any effect!

- Crash
- Hang
- Wrong results



Concurrency bug example

```
int x = 0;

void threadA(){
    Lock_x.acquire();
    A = x + 1;
    x = A;
    Lock_x.release();
}

void threadB(){
    Lock_x.acquire();
    B = x + 1;
    x = B;
    Lock_x.release();
}
```

Concurrency bug example

```
int x = 0;

void threadA(){
    Lock_x.acquire();
    A = x + 1;
    x = A;
    Lock_x.release();
}

void threadB(){
    Lock_x.acquire();
    B = x + 1;
    x = B;
    Lock_x.release();
}
```

1

```
A = x + 1
                B = x + 1
                x = B
x = A
```

2

```
A = x + 1
                B = x + 1
x = A
                x = B
```

3

```
A = x + 1
x = A
                B = x + 1
                x = B
```

Concurrency bug example

```
int x = 0;

void threadA(){
    Lock_x.acquire();
    A = x + 1;
    x = A;
    Lock_x.release();
}

void threadB(){
    Lock_x.acquire();
    B = x + 1;
    x = B;
    Lock_x.release();
}
```

1

```
A = x + 1
                B = x + 1
                x = B
x = A
```

2

```
A = x + 1
                B = x + 1
x = A
                x = B
```

3

```
A = x + 1
x = A
                B = x + 1
                x = B
```

What about this solution?

```
int x = 0;

void threadA(){

   x = x + 1;

}

void threadB(){

   x = x + 1;

}
```

What about this solution?

```
int x = 0;

void threadA(){

    x = x + 1;

}

void threadB(){

    x = x + 1;

}
```

Still buggy!

What about this solution?

One C statement → Many instructions

```
00000000004004ed threadA
  4004ed:   55                     push   %rbp
  4004ee:   48 89 e5               mov    %rsp,%rbp
  4004f1:   8b 05 45 0b 20 00      mov    0x200b45(%rip),%eax    # 60103c <x>
  4004f7:   83 c0 01               add    $0x1,%eax
  4004fa:   89 05 3c 0b 20 00      mov    %eax,0x200b3c(%rip)    # 60103c <x>
  400500:   5d                     pop    %rbp
  400501:   c3                     retq

0000000000400502 threadB
  400502:   55                     push   %rbp
  400503:   48 89 e5               mov    %rsp,%rbp
  400506:   8b 05 30 0b 20 00      mov    0x200b30(%rip),%eax    # 60103c <x>
  40050c:   83 c0 01               add    $0x1,%eax
  40050f:   89 05 27 0b 20 00      mov    %eax,0x200b27(%rip)    # 60103c <x>
  400515:   5d                     pop    %rbp
  400516:   c3                     retq
```

What about this code?

```
int x = 0;

void threadA(){

   x = CONSTANT_A;

}

void threadB(){

   x = CONSTANT_B;

}
```

What about this code?

Still not a good idea!

```
int x = 0;

void threadA(){

   x = CONSTANT_A;

}

void threadB(){

   x = CONSTANT_B;

}
```

What about this code?

```
int x = 0;

void threadA(){

   x = CONSTANT_A;

}

void threadB(){

   x = CONSTANT_B;

}
```

Still not a good idea!

a) Compiler might still emit multiple instructions

What about this code?

```
int x = 0;

void threadA(){

    x = CONSTANT_A;

}

void threadB(){

    x = CONSTANT_B;

}
```

Still not a good idea!

a) Compiler might still emit multiple instructions

and...

b) Some instructions are not atomic

What about this code?

```
int x = 0;

void threadA(){

   x = CONSTANT_A;

}

void threadB(){

   x = CONSTANT_B;

}
```

Still not a good idea!

a) Compiler might still emit multiple instructions

and...

b) Some instructions are not atomic

and...

c) C standard says don't do it

What about this code?

```
int x = 0;

void threadA(){

    x = CONSTANT_A;

}

void threadB(){

    x = CONSTANT_B;

}
```

Still not a good idea!*

a) Compiler might still emit multiple instructions

and...

b) Some instructions are not atomic

and...

c) C standard says don't do it

* Kernel developers sometimes do it though

Kernel concurrency bugs

- Bugs that depend on the instruction interleavings
  - **Triggered only by a subset** of the interleavings
- Plenty of kernel concurrency bugs in kernels!

The bug is a race ...
particular machi...
within **10 minut**...
timing such tha...
machine is in trouble and needs to be rebooted.

Three of the five 3.4.9 machines [...] **locked up.** I've tried reproducing the issue, but so far I've been unsuccessful [...]

Linux kernel mailing list (5/1/2013)

[The bug] was quite hard to decode as the reproduction time is between **2 days and 3 weeks and intrusive tracing makes it less likely** [...]

Linux 3.4.41 change log

Approaches to explore interleavings

- Stress testing approach
  - Hope to find the interleaving
- Systematic approach
  - Take full control of the interleavings
  - ~~Existing tools focus on user mode applications~~

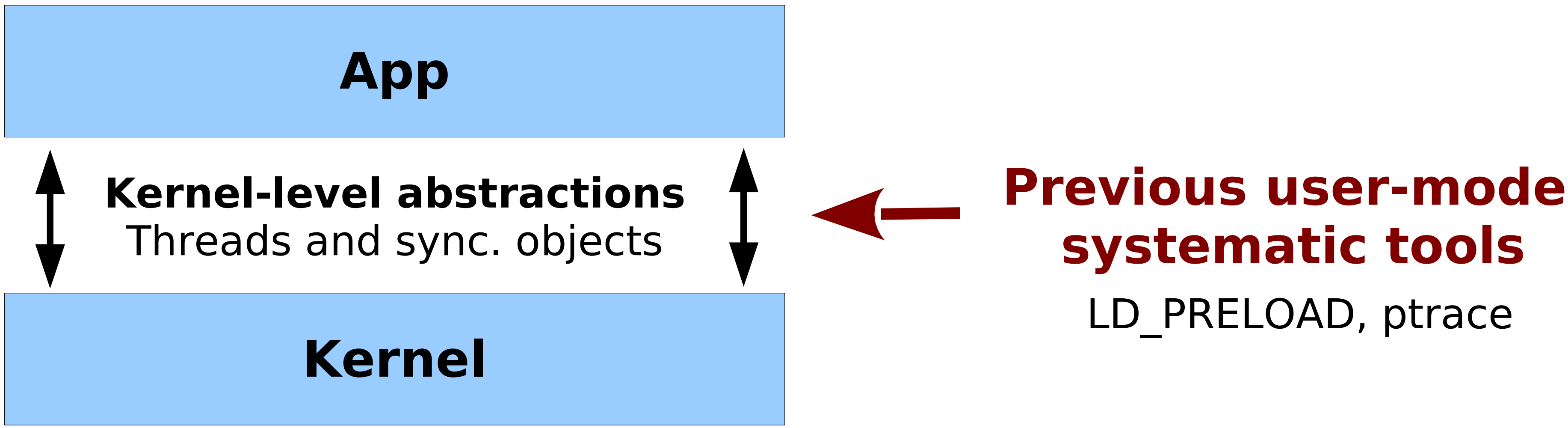
Focus on operating system kernels

SKI

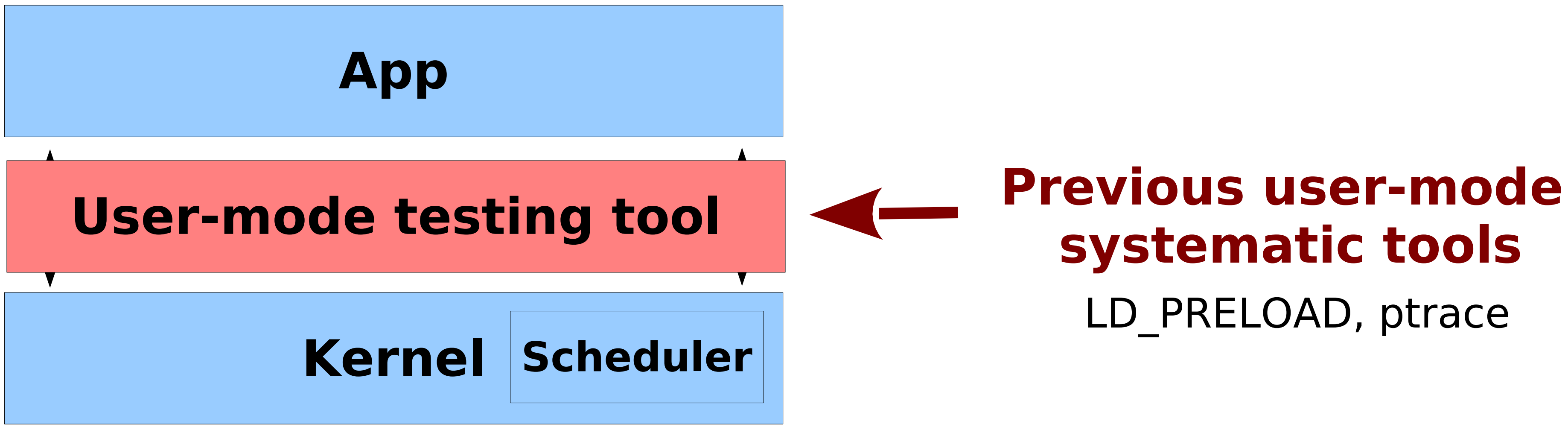
Finding kernel concurrency bugs

- **Testing applications versus kernels**
- Our approach
- Implementation
- Evaluation

User-mode tools

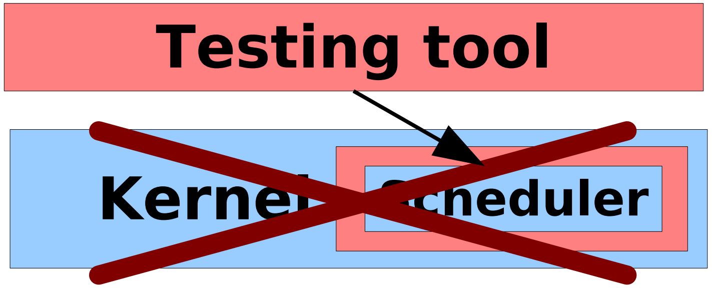


User-mode tools



Kernel-mode challenges

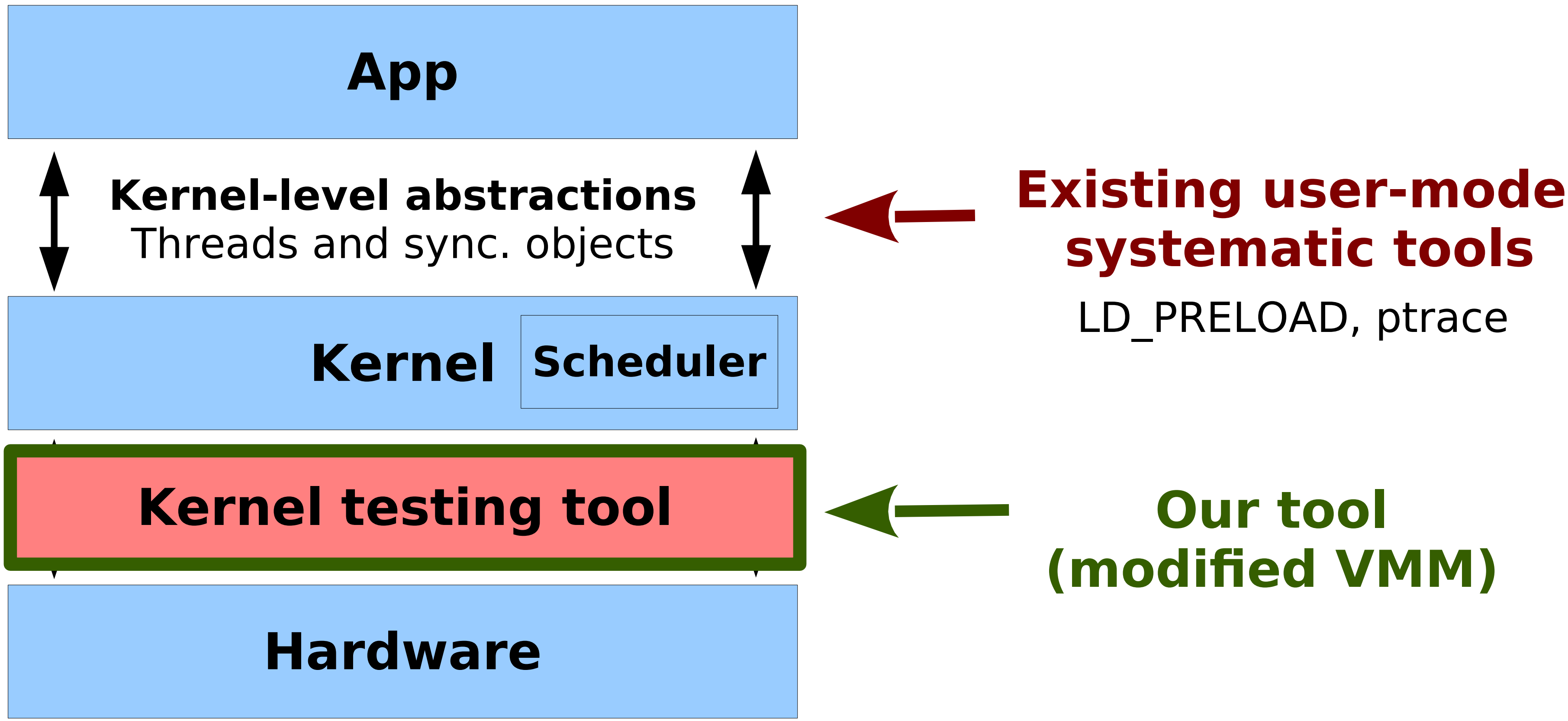
- Kernel doesn't have a good instrumentation interface



- An alternative would be to modify the kernel
  - But kernel modifications:
    - Change the tested software
    - Are non-trivial
    - Hinder portability

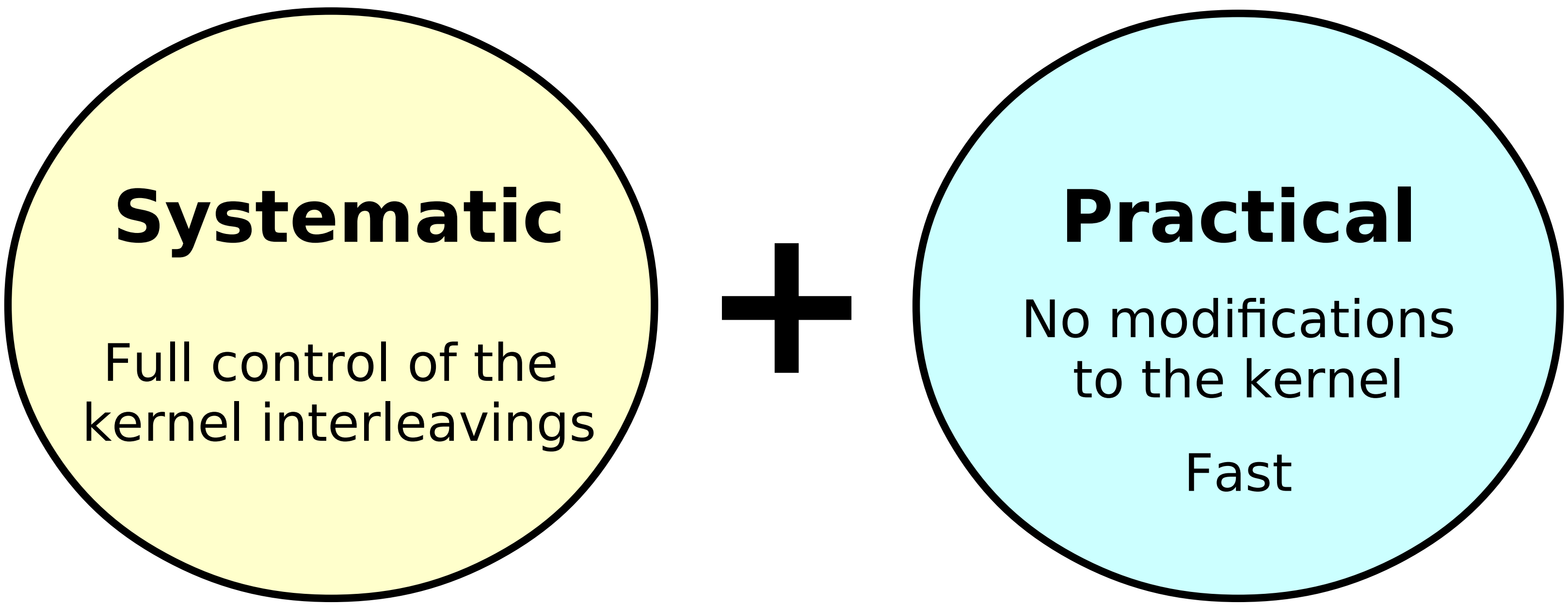
Avoid kernel modifications

User-mode *versus* kernel-mode



SKI

Finding kernel concurrency bugs

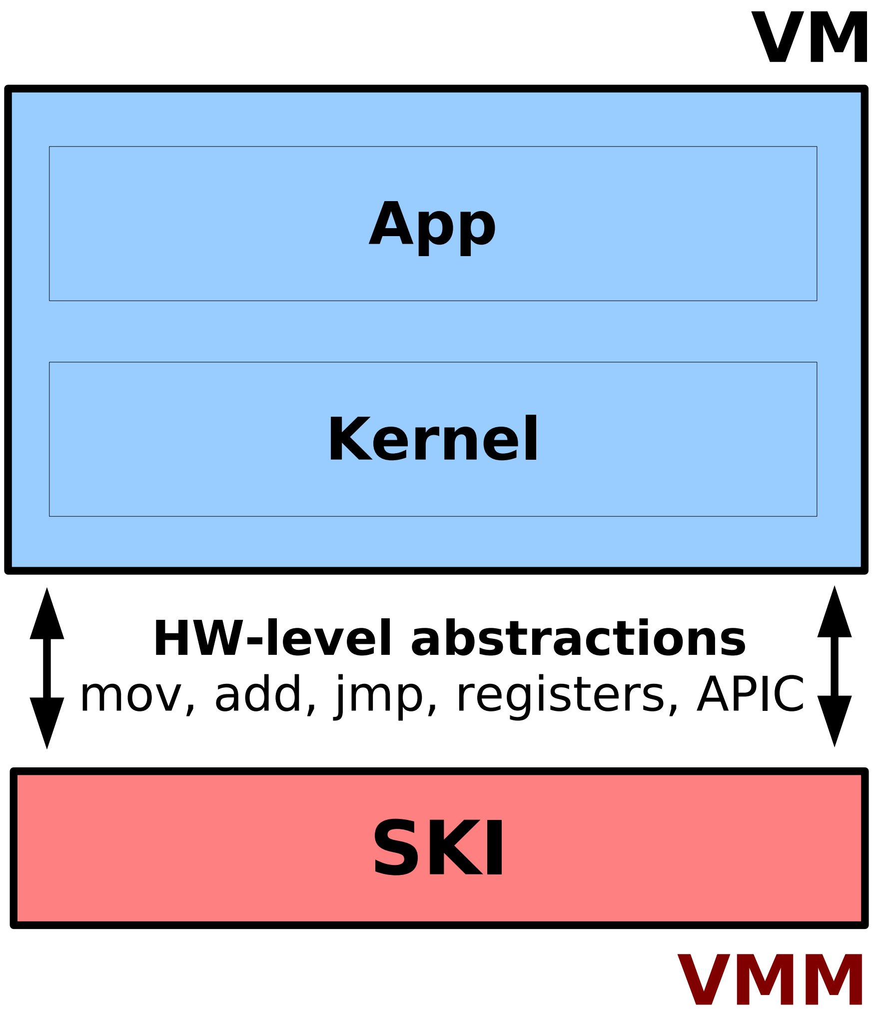


SKI

Finding kernel concurrency bugs

- Challenges testing the kernel code
- **SKI's approach**
- Implementation
- Evaluation

SKI's approach

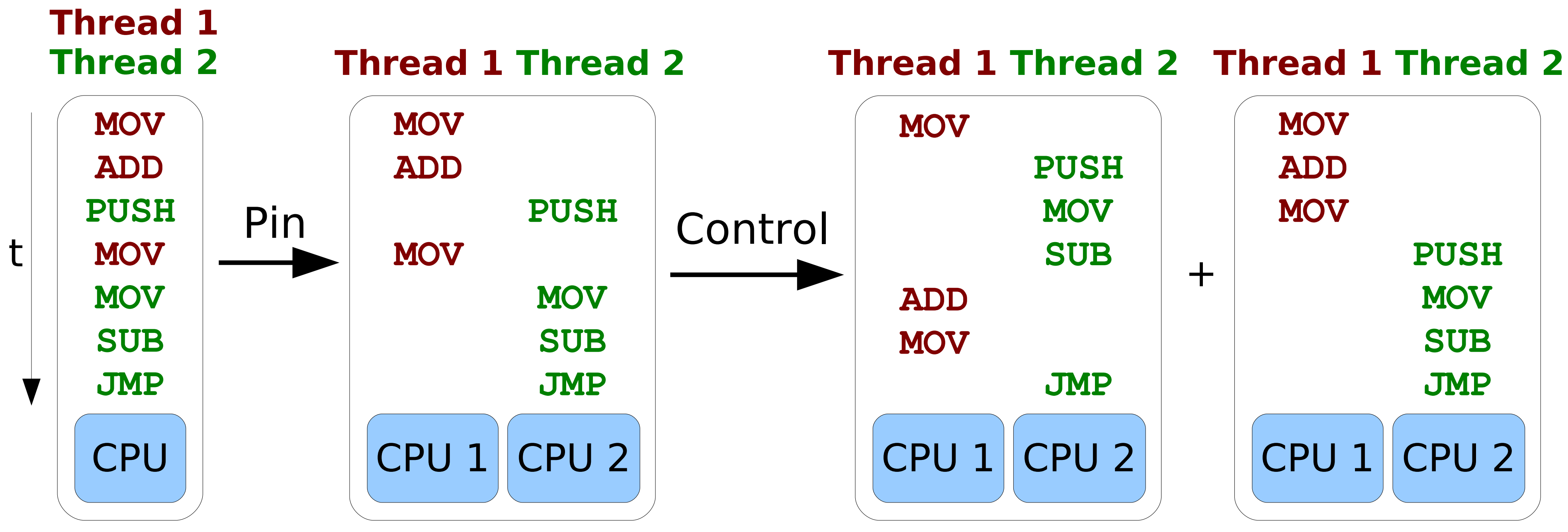


Challenges

1. How to control the schedules?
2. Which contexts are schedulable?
3. Which schedules to choose?

1. How to control the kernel schedules?

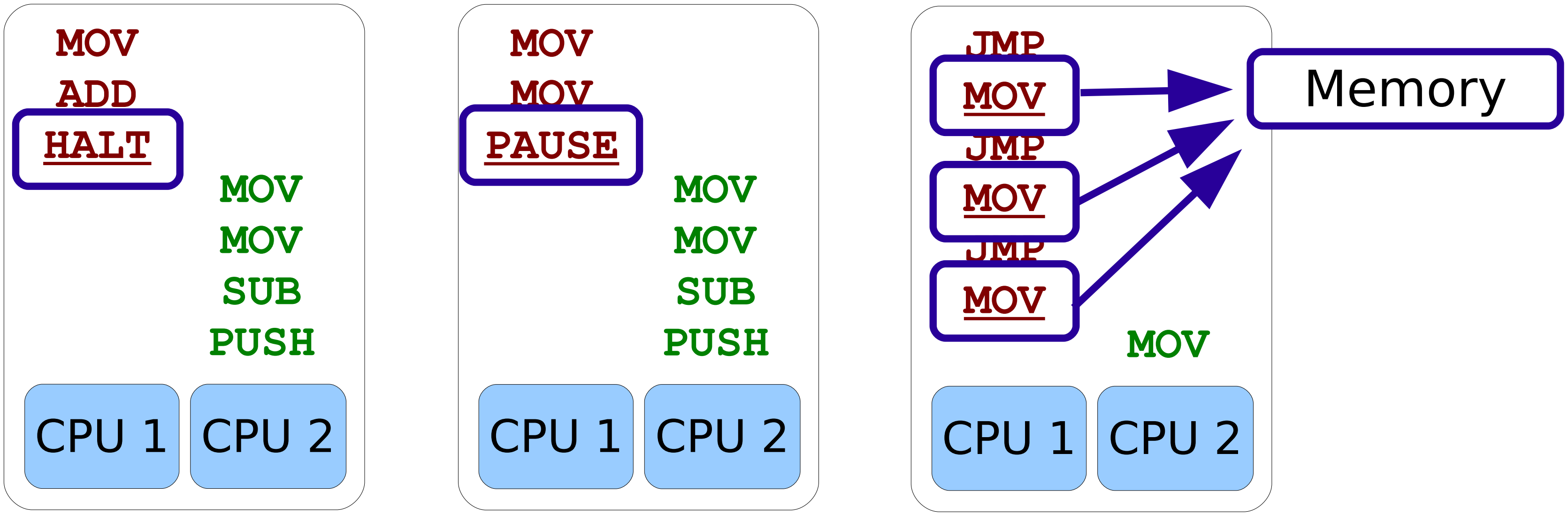
- Pin each tested thread to a different CPU (thread affinity)
- Pause and resume CPUs to control schedules



Leverage thread affinity and control CPUs

2. Which contexts are schedulable?

- Execution of some instructions are good hints
- Memory access patterns can also provide hints



Rely on VMM introspection

3. Which schedules to choose?

- PCT: User-mode scheduling algorithm [ASPLOS'10]
  - Run the highest priority live threads
  - Create schedule diversity
- Generalize with interrupt support
  - Detect arrival / end
  - Control dispatch
- Reduce interleaving space

Generalize user-mode systematic testing algorithms

SKI

Finding kernel concurrency bugs

- Challenges testing kernel code
- SKI's approach
- **Implementation**
- Evaluation

Implementation

- Implemented SKI by modifying QEMU (VMM)
  - No kernel changes required
- Built a user-mode library (VM)
  - Flags start/end of tests and sends results to VMM
  - Used library to implement several test-cases
    - e.g., file system tests
- Implemented several optimizations

Detecting and diagnosing bugs with SKI

- SKI supports different types of bug detectors
  - Crash and assertion violations
  - Data races
  - Semantic bugs (e.g. disk corruption)
- SKI produces detailed execution traces

SKI

Finding kernel concurrency bugs

- Challenges testing kernel code
- SKI's approach
- Implementation
- **Evaluation**

1. Regression testing

2. Finding previously unknown bugs

1. Regression testing: setup

- Searched for previously reported bugs
  - In kernel bugzilla, mailing lists, git logs
  - Well documented reports and diverse set of bugs
- Created SKI test suites for these bugs
  - By adapting the stress tests in the bug reports

1. Regression testing: results

Bug	Kernel	Component	Detector
A	Linux 2.6.28	Anonymous pipes	Crash
B	Linux 3.2	Inotify + FAT32	Crash
C	Linux 3.6.1	Proc + Ext4	Semantic
D	FreeBSD 8.0	Sockets	Semantic

1. Regression testing: results

Bug	Kernel	Component	Detector
A	Linux 2.6.28	Anonymous pipes	Crash
B	Linux 3.2	Inotify + FAT32	Crash
C	Linux 3.6.1	Proc + Ext4	Semantic
D	FreeBSD 8.0	Sockets	Semantic

Diverse properties

1. Regression testing: results

Bug	Kernel	Component	Detector
A	Linux 2.6.28	Anonymous pipes	Crash
B	Linux 3.2	Inotify + FAT32	Crash
C	Linux 3.6.1	Proc + Ext4	Semantic
D	FreeBSD 8.0	Sockets	Semantic

1. Regression testing: results

Bug	Kernel	Component	Detector
A	Linux 2.6.28	Anonymous pipes	Crash
B	Linux 3.2	Inotify + FAT32	Crash
C	Linux 3.6.1	Proc + Ext4	Semantic
D	FreeBSD 8.0	Sockets	Semantic

SKI is portable

1. Regression testing: results

				SKI	
Bug	**Kernel**	**Component**	**Detector**	**Schedules**	**Throughput (sched/h)**
A	Linux 2.6.28	Anonymous pipes	Crash	28	302,000
B	Linux 3.2	Inotify + FAT32	Crash	53	169,300
C	Linux 3.6.1	Proc + Ext4	Semantic	51	218,700
D	FreeBSD 8.0	Sockets	Semantic	3519	501,400

1. Regression testing: results

SKI can expose bugs in seconds

				SKI	
Bug	**Kernel**	**Component**	**Detector**	**Schedules**	**Throughput (sched/h)**
A	Linux 2.6.28	Anonymous pipes	Crash	28	302,000
B	Linux 3.2	Inotify + FAT32	Crash	53	169,300
C	Linux 3.6.1	Proc + Ext4	Semantic	51	218,700
D	FreeBSD 8.0	Sockets	Semantic	3519	501,400

1. Regression testing: results

				SKI		Stress tests
Bug	**Kernel**	**Component**	**Detector**	**Schedules**	**Throughput (sched/h)**	**Schedules**
A	Linux 2.6.28	Anonymous pipes	Crash	28	302,000	NA (>24h)
B	Linux 3.2	Inotify + FAT32	Crash	53	169,300	200,000 (4h)
C	Linux 3.6.1	Proc + Ext4	Semantic	51	218,700	800 (1 min)
D	FreeBSD 8.0	Sockets	Semantic	3519	501,400	NA (>24h)

1. Regression testing: results

Some stress tests were ineffective

				SKI		Stress tests
Bug	**Kernel**	**Component**	**Detector**	**Schedules**	**Throughput (sched/h)**	**Schedules**
A	Linux 2.6.28	Anonymous pipes	Crash	28	302,000	NA (>24h)
B	Linux 3.2	Inotify + FAT32	Crash	53	169,300	200,000 (4h)
C	Linux 3.6.1	Proc + Ext4	Semantic	51	218,700	800 (1 min)
D	FreeBSD 8.0	Sockets	Semantic	3519	501,400	NA (>24h)

2. Finding previously unknown bugs

- Created a SKI test suit for file systems
  - Adapted the existing *fsstress* test suit
  - Tested several file systems
- Bug detectors
  - Crashes, warnings, data races, semantic errors (*fsck*)
- Tested recent versions of Linux

2. Finding previously unknown bugs

Bug	Linux	FS	Detector / Failure	Status
1	3.11.1	Btrfs	Crash (Null-pointer)	Fixed
2	3.11.1	Btrfs	Crash (Null-pointer) + Warning	Fixed
3	3.11.1	Btrfs	Warning	Fixed
4	3.11.1	Btrfs	Fsck (References not found)	Reported
5	3.11.1+p	Btrfs	Crash (Null-pointer)	Fixed
6	3.12.2	Btrfs	Warning	Fixed
7	3.13.5	Logfs	Crash (Null-pointer)	Reported
8	3.13.5	Logfs	Crash (Invalid paging)	Reported
9	3.13.5	Jfs	Crash (Assertion violation)	Reported
10	3.13.5	Ext4	Data race	Fixed
11	3.13.5	VFS	Data race	Reported

2. Finding previously unknown bugs

Bug	Linux	FS	Detector / Failure	Status
1	3.11.1	Btrfs	Crash (Null-pointer)	Fixed
2	3.11.1	Btrfs	Crash (	
3	3.11.1	Btrfs		
4	3.11.1	Btrfs	Fsck (References not found)	Reported
5	3.11.1+p	Btrfs	Crash (Null-pointer)	Fixed
6	3.12.2	Btrfs	Warning	Fixed
7	3.13.5	Logfs	Crash (Null-pointer)	Reported
8	3.13.5	Logfs	Crash (Invalid paging)	Reported
9	3.13.5	Jfs	Crash (Assertion violation)	Reported
10	3.13.5	Ext4	Data race	Fixed
11	3.13.5	VFS	Data race	Reported

Official Linux releases

2. Finding previously unknown bugs

Bug	Linux	FS	Detector / Failure	Status
1	3.11.1	Btrfs	Crash (Null-pointer)	Fixed
2	3.11.1	Btrfs	[illegible]	[illegible]
3	3.11.1	Btrfs	[illegible]	[illegible]
4	3.11.1	Btrfs	Fsck (References not found)	Reported
5	3.11.1+p	Btrfs	Crash (Null-pointer)	Fixed
6	3.12.2	Btrfs	Warning	Fixed
7	3.13.5	Logfs	Crash (Null-pointer)	Reported
8	3.13.5	Logfs	Crash (Invalid paging)	Reported
9	3.13.5	Jfs	Crash (Assertion violation)	Reported
10	3.13.5	Ext4	Data race	Fixed
11	3.13.5	VFS	Data race	Reported

Requested by developers

2. Finding previously unknown bugs

Bug	Linux	FS	Detector / Failure	Status
1	3.11.1	Btrfs	Crash (Null-pointer)	Fixed
2	3.11.1	Btrfs	Crash (Null-pointer) + Warning	Fixed
3	3.11.1	Btrfs	Warning	Fixed
4	3.11.1	Btrfs	Fsck (References not found)	Reported
5	3.11.1+p	Btrfs	Crash (Null-pointer)	Fixed
6	3.12.2	Btrfs	Warning	Fixed
7	3.13.5	Logfs	Crash (Null-pointer)	Reported
8	3.13.5	Logfs	Crash (Invalid paging)	Reported
9	3.13.5	Jfs	Crash (Assertion violation)	Reported
10	3.13.5	Ext4	Data race	Fixed
11	3.13.5	VFS	Data race	Reported

Important file systems

2. Finding previously unknown bugs

Bug	Linux	FS	Detector / Failure	Status
1	3.11.1	Btrfs	Crash (Null-pointer)	Fixed
2	3.11.1	Btrfs	Crash (Null-pointer) + Warning	Fixed
3	3.11.1	Btrfs	Warning	Fixed
4	3.11.1	Btrfs	Fsck (References not found)	Reported
5	3.11.1+p	Btrfs	Crash (Null-pointer)	Fixed
6	3.12.2	Btrfs	Warning	Fixed
7	3.13.5	Logfs	Crash (Null-pointer)	Reported
8	3.13.5	Logfs	Crash (Invalid paging)	Reported
9	3.13.5	JF[illegible]	Crash (Assertion violation)	Reported
10	3.13.5	Ext4	Data race	Fixed
11	3.13.5	VFS	Data race	Reported

Data loss

Current limitations and future work

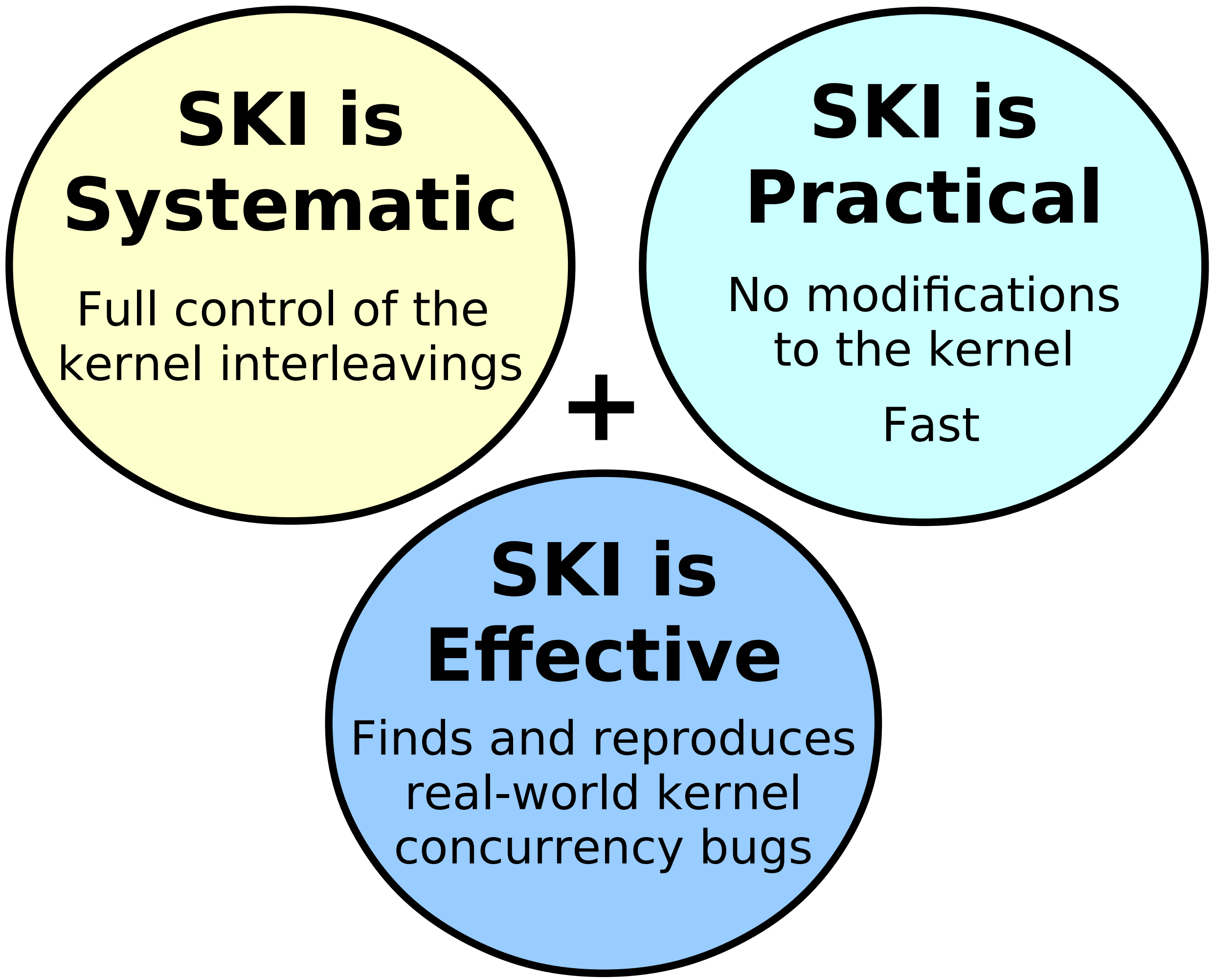
- Bugs in kernel scheduler code
  - SKI pins tested threads

    → Represent a small set of bugs
- Bugs in device drivers
  - SKI supports a large set of devices but not all

    → Implement SKI with binary instrumentation techniques
- Bugs that depend on weak memory models
  - SKI currently implements a strong memory model

    → Generalize SKI to also expose these bugs

SKI: Finding kernel concurrency bugs



SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration
Pedro Fonseca, Rodrigo Rodrigues and Björn B. Brandenburg
OSDI'14

Take-away: concurrency is hard!