REAL WORLD ANALYSIS OF CONCURRENT PROGRAMS

Samuel Mimram
École Polytechique
Many possible approaches:
- dynamic detection / post mortem techniques
- testing
- static analysis
  - abstract interpretation
  - model checking
  - typing

I will present some approaches:
- obtained by randomly browsing the internet
- does not follow historical order
- does not follow impact order
Race conditions

We want to detect **race conditions**: unprotected concurrent access to memory (one of which is a write).
**Lockset** analysis is based on the idea that every accessed shared variable should have a lock associated to it. It was introduced in


They use a dynamic analysis:

Let \( \text{locks\_held}(t) \) be the set of locks held by thread \( t \).

For each \( v \), initialize \( C(v) \) to the set of all locks.

On each access to \( v \) by thread \( t \),

\[
\text{set } C(v) := C(v) \cap \text{locks\_held}(t);
\]

if \( C(v) = \{ \} \), then issue a warning.
Locksets: Eraser

This has to be improved a bit, but the idea is here:

- **Initialization**: shared variables are frequently initialized without holding a lock.
- **Read-Shared Data**: Some shared variables are written during initialization only and are read-only thereafter. These can be safely accessed without locks.
- **Read-Write Locks**: Read-write locks allow multiple readers to access a shared variable, but allow only a single writer to do so.
Locksets: Eraser

They can run on unmodified binaries and found bugs in
- the SPIN operating system
- the HTTP server and indexing engine of AltaVista
- students homework
- etc.

We see that, of course, this can be turned into a static analysis.
Let’s study locking mechanisms in the Linux kernel...
Mutexes

The Linux kernel provides mutexes:

```c
struct mutex {
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
};
```

where

- `count` is the state:
  - 1: available
  - 0: locked
  - -1: locked with other processes waiting

- `wait_list` is the list of waiting processes
- `wait_lock` protects `wait_list`
Mutexes

The Linux kernel provides mutexes:

```c
struct mutex {
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
};
```

where

- `count` is the state:
  - 1: available
  - 0: locked
  - -1: locked with other processes waiting

- `wait_list` is the list of waiting processes
- `wait_lock` protects `wait_list`

They are not reentrant.
Mutexes

- create a mutex
  ```c
  void mutex_init(struct mutex *lock);
  ```

- sleep until a mutex is available and lock it
  ```c
  void mutex_lock(struct mutex *lock);
  int mutex_lock_interruptible(struct mutex *lock);
  int mutex_trylock(struct mutex *lock);
  ```

- unlock a mutex
  ```c
  void mutex_unlock(struct mutex *lock);
  ```

- check the state of a mutex
  ```c
  int mutex_is_locked(struct mutex *lock);
  ```
Sleeping

When we `sleep`, the current thread gets paused and other can run.

There are other function which are sleepy:

- mutexes: `mutex_lock`
- waiting for I/O: `wait_event / poll_wait / etc.`
- memory allocation: `kmalloc(..., GFP_KERNEL)`
- interaction with userspace: `get_user / put_user`
- explicit scheduling: `schedule`
- etc.
Spinlocks

When we lock a locked mutex, the scheduler might schedule another thread instead, which is costly (context switch). For small and quick portions of code (e.g. modifying one variable), another primitive called spinlocks is available.

It is faster, but restricted to atomic code:

- locking disables preemption,
- optionally disables interrupts,
- the guarded section is supposed to never sleep.
Spinlocks

When we lock a locked mutex, the scheduler might schedule another thread instead, which is costly (context switch). For small and quick portions of code (e.g. modifying one variable), another primitive called spinlocks is available.

It is faster, but restricted to atomic code:

- locking disables preemption,
- optionally disables interrupts,
- the guarded section is supposed to never sleep.

Those are not reentrant.
Spinlocks

When we lock a locked mutex, the scheduler might schedule another thread instead, which is costly (context switch). For small and quick portions of code (e.g. modifying one variable), another primitive called **spinlocks** is available.

It is faster, but restricted to atomic code:
- locking disables preemption,
- optionally disables interrupts,
- the guarded section is supposed to **never** sleep.

Those are not reentrant.

Note: other pieces of code make such assumptions such as interrupt handlers.
Spinlocks

note: mirlin[1083] exited with preempt_count 1
BUG: scheduling while atomic: mirlin/1083/0x40000002
Modules linked in: g_cdc_ms musb_hdrc nop_usb_xceiv irqk edmak dm365mmay
Backtrace:
[<c002a5a0>] (dump_backtrace+0x0/0x110) from [<c028e56c>] (dump_stack+0x0/0x110)
   r6:c1099460 r5:c04ea000 r4:00000000 r3:20000013
[<c028e554>] (dump_stack+0x0/0x1c) from [<c00337b8>] (__schedule_bug+0x0/0x1c)
[<c0033760>] (__schedule_bug+0x0/0x64) from [<c028e864>] (schedule+0x84/0x378)
   r4:c10992c0 r3:00000000
[<c028e7e0>] (schedule+0x0/0x378) from [<c0033a80>] (__cond_resched+0x0/0x38)
[<c0033a58>] (__cond_resched+0x0/0x38) from [<c028ec6c>] (__cond_resched+0x0/0x38)
   r4:00013000 r3:00000001
[<c028ec38>] (__cond_resched+0x0/0x44) from [<c0082f64>] (unmap_vmas+0x570/0x620)
[<c00829f4>] (unmap_vmas+0x0/0x620) from [<c0085c10>] (exit_mmap+0xc0/0x1ec)
[<c0085b50>] (exit_mmap+0x0/0x1ec) from [<c0037610>] (mmput+0x40/0xfc)
   r9:00000001 r8:80000005 r6:c04ea000 r5:00000000 r4:c0427300
[<c00375d0>] (mmput+0x0/0xfc) from [<c003b5e4>] (exit_mm+0x150/0x158)
   r5:c10992c0 r4:c0427300
[<c003b494>] (exit_mm+0x0/0x158) from [<c003cd44>] (do_exit+0x198/0x67c)
   r7:c03120d1 r6:c10992c0 r5:0000000b r4:c10992c0
...
Spinlocks

- **initialization**
  
  ```c
  void spin_lock_init(spinlock_t *lock);
  ```

- **locking**
  
  ```c
  void spin_lock(spinlock_t *lock);
  ```

- **releasing**
  
  ```c
  void spin_unlock(spinlock_t *lock);
  ```
Spinlocks implementation

```assembly
locked:       ; The lock variable.
    dd 0

spin_lock:
    mov   eax, 1    ; Set the EAX register to 1.
    xchg  eax, [locked] ; Atomically swap the EAX register with the lock variable.
    ; This will always store 1 to the lock, leaving the previous
    ; value in the EAX register.
    test  eax, eax  ; Test EAX with itself. Among other things, this will
    ; set the processor’s Zero Flag if EAX is 0.
    ; If EAX is 0, then the lock was unlocked and we just locked it.
    ; Otherwise, EAX is 1 and we didn’t acquire the lock.
    jnz   spin_lock ; Jump back to the MOV instruction if the Zero Flag is
    ; not set; the lock was previously locked, and so
    ; we need to spin until it becomes unlocked.
    ret   ; The lock has been acquired, return to the calling function.

spin_unlock:
    mov   eax, 0    ; Set the EAX register to 0.
    xchg  eax, [locked] ; Atomically swap the EAX register with the lock variable.
    ret   ; The lock has been released.
```
Detecting scheduling while atomic

Let’s find some bugs in the kernel!

Detecting scheduling while atomic

A function **may sleep** if

- it calls a sleepy function (*wait_event*, etc.)
- it calls a function which may sleep

This is easy to infer by “abstract interpretation” (= simple propagation)!
A portion of code is **spinlocked** if it is of the form

```c
spin_lock(...);

... // no spinlock-related function
spin_unlock(...);
```
Detecting scheduling while atomic

A crude approximation can be obtained by computing an over-estimation of the number of locked variables:

\[ N(c) : \mathbb{N} \rightarrow \mathbb{N} \]

defined by

1. \( N(a; b)(n) = N(b)(N(a)(n)) \)
2. \( N(\text{spin\_lock}(...))(n) = n + 1 \)
3. \( N(\text{spin\_unlock}(...))(n) = n - 1 \)
4. \( N(f(...))(n) = n \)
5. \( N(\text{if} a \text{ then } b \text{ else } c)(n) = \max(N(b)(n), N(c)(n)) \)
6. \( N(\text{while } a \ b)(n) = \max(n, N(b)(n) \times \infty) \)
Detecting scheduling while atomic

A crude approximation can be obtained by computing an over-estimation of the number of locked variables:

\[ N(c) : \mathbb{N} \rightarrow \mathbb{N} \]

defined by

\[ N(a; b)(n) = N(b)(N(a)(n)) \]
\[ N(\text{spin\_lock}(\ldots))(n) = n + 1 \]
\[ N(\text{spin\_unlock}(\ldots))(n) = n - 1 \]
\[ N(f(\ldots))(n) = n \]
\[ N(\text{if} a \text{ then } b \text{ else } c)(n) = \max(N(b)(n), N(c)(n)) \]
\[ N(\text{while } a \ b)(n) = \max(n, N(b)(n) \times \infty) \]

This is really too crude, one can do better if we suppose that loops are conservative, and take breaks and returns in account, but you get the idea.
Detecting scheduling while atomic

If we combine the two we can detect potential scheduling in spinlocked regions.
Double locks

It can be adapted in order to detect "double locks":

```
spin_lock(x);
...
spin_lock(x);
```

Instead of counting the number of locks, we can remember about which lock has been taken: `locksets`.

This is difficult, so we should remember about some information about the locks:

- global locks,
- for non-global locks, we abstract those by the type of the structure the lock belongs to.
Bugs that you can find

In `snd_sb_csp_load()` in `sb16_csp.c`:

```c
...  
spin_lock_irqsave(&p->chip->reg_lock, flags);
...  
unsigned char *kbuf, *_kbuf;
_kbuf = kbuf = kmalloc (size, GFP_KERNEL);
...  
(fixed in 2.6.11)
```
Bugs that you can find

In \texttt{midi\_outc()} of \texttt{sound/oss/sequencer.c}:

\begin{verbatim}
spin_lock_irqsave(&lock, flags);
while (n && !midi_devs[dev]->outputc(dev, data)) {
    interruptible_sleep_on_timeout(&seq_sleeper, ...);
    n--;
}
spin_unlock_irqrestore(&lock, flags);
\end{verbatim}
More on locksets: RacerX

A similar analysis is performed in


In RacerX they

- compute locksets (locks are abstracted by their type)
  - they compute all possible locksets as output of a function
- cache results for each function
  - (lockset before $\rightarrow$ locksets after)
- they compute possible ordering of locks
  - i.e. whether $b$ can be locked while $a$ is along with a small trace (to display error paths)
- and find cycles in dependencies
- they have a ranking of errors
ERROR: 2 thread global-global deadlock.
<rtc_lock>-><rtc_task_lock> occurred 1 time
<rtc_task_lock>-><rtc_lock> occurred 1 time

<rtc_lock>-><rtc_task_lock> =
depth = 1:
    linux-2.5.62/drivers/char/rtc.c:rtc_register:723
    ->rtc_register:728

int rtc_register(rtc_task_t *task) {
    if (task == NULL || task->func == NULL)
        return -EINVAL;
    spin_lock_irq(&rtc_lock);
    if (rtc_status & RTC_IS_OPEN) {
        spin_unlock_irq(&rtc_lock);
        return -EBUSY;
    }
    spin_lock(&rtc_task_lock);
    if (rtc_callback) {
        spin_unlock(&rtc_task_lock);
        spin_unlock_irq(&rtc_lock);
        return -EBUSY;
    }
    spin_lock(&rtc_task_lock);
    if (rtc_callback) {
        spin_unlock(&rtc_task_lock);
        spin_unlock_irq(&rtc_lock);
        return -EBUSY;
    }

<rtc_task_lock>-><rtc_lock> =
depth = 1:
    linux-2.5.62/drivers/char/rtc.c:rtc_unregister:749
    ->rtc_unregister:755
int rtc_unregister(rtc_task_t *task) {
    spin_lock_irq(&rtc_task_lock);
    if (rtc_callback != task) {
        spin_unlock_irq(&rtc_task_lock);
        return -ENXIO;
    }
    rtc_callback = NULL;
    spin_lock(&rtc_lock);
}
Signaling mutexes

Semaphores can have two uses:

▶ mutual exclusion
▶ wait for signals

In the second case, we generally have

▶ a producer
  up(s); // signal ready
▶ a consumer
  lock(l);
  down(s); // wait for result
  unlock(l);
  ...
  lock(l);

It looks like there is a possible deadlock between two consumers. We can use belief analysis to distinguish between the two cases.
Sleeping under spinlocks

We can also detect the “sleeping under spinlock” error:

```
//linux-2.5.62/net/atm/common.c:556:atm_ioctl:ERROR:BLOCK
// calling blocking function <put_user> w/ lock held!
spin_lock (&atm_dev_lock);
vcc = ATM_SD(sock);
switch (cmd) {
  case SIOCOUTQ:
    ...
    ret_val = put_user(...); // ERROR: can block.
```
It is interesting to notice that rarely executed code suffer from such problems...
Leap second bugs

The length of a day isn’t exactly 24h so we have to insert seconds at the end of the day from time to time:
Leap second bugs

From ntp_leap_second of kernel/time/ntp.c:

write_seqlock(&xtime_lock);
switch (time_state) {
case TIME_INS:
    timekeeping_leap_insert(-1);
    time_state = TIME_OOP;
    clock_was_set();
    printk(KERN_NOTICE "Clock: inserting leap second 23:59:60 UTC\n"
          break;
    
case TIME_DEL:
    timekeeping_leap_insert(1);
    time_state = TIME_WAIT;
    clock_was_set();
    printk(KERN_NOTICE "Clock: deleting leap second 23:59:59 UTC\n"
          break;
    // (more cases omitted ...)
}
write_sequnlock(&xtime_lock);
Leap second bugs

There were (at least) four bugs\(^1\) related to the `xtime_lock` spinlock:

1. `clock_was_set` calls `smp_call_function` (to retrigger CPU local events), which can sleep
   \[\Rightarrow\text{remove } \text{clock\_was\_set}()\]

\[^1\]http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html
\[^2\]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d
Leap second bugs

There were (at least) four bugs\(^1\) related to the `xtime_lock` spinlock:

1. `clock_was_set` calls `smp_call_function` (to retrigger CPU local events), which can sleep
   \(\Rightarrow\) remove `clock_was_set()`

2. `printk` needs to schedule logging, which can check the timer under heavy load, and thus lock `xtime_lock` again

\(^1\) [http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html](http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html)

\(^2\) [https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d)
Leap second bugs

There were (at least) four bugs\(^1\) related to the `xtime_lock` spinlock:

1. `clock_was_set` calls `smp_call_function` (to retrigger CPU local events), which can sleep
   \[\Rightarrow\] remove `clock_was_set()`

2. `printk` needs to schedule logging, which can check the timer under heavy load, and thus lock `xtime_lock` again

3. `ntp_lock` was split from `xtime_lock`, with a deadlock\(^2\)

\(^1\) [http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html](http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html)

\(^2\) [https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d)
Leap second bugs

There were (at least) four bugs\(^1\) related to the \texttt{xtime_lock} spinlock:

1. \texttt{clock_was_set} calls \texttt{smp_call_function} (to retrigger CPU local events), which can sleep
   \[\Rightarrow\text{remove } \texttt{clock_was_set}()\]

2. \texttt{printk} needs to schedule logging, which can check the timer under heavy load, and thus lock \texttt{xtime_lock} again

3. \texttt{ntp_lock} was split from \texttt{xtime_lock}, with a deadlock\(^2\)

4. actually removing \texttt{clock_was_set}() was not a good idea because it made sub-second high-resolution timers to immediately return, which causes userspace applications that use them in loops to instead run in tight loops eating up CPU

\(^1\)\url{http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html}
\(^2\)\url{https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d}
Leap second bugs

There where (at least) four bugs\(^1\) related to the \texttt{xtime_lock} spinlock:

1. \texttt{clock\_was\_set} calls \texttt{smp\_call\_function} (to retrigger CPU local events), which can sleep
   \[\Rightarrow\] remove \texttt{clock\_was\_set}()

2. \texttt{printk} needs to schedule logging, which can check the timer under heavy load, and thus lock \texttt{xtime\_lock} again

3. \texttt{ntp\_lock} was split from \texttt{xtime\_lock}, with a deadlock\(^2\)

4. actually removing \texttt{clock\_was\_set}() was not a good idea because it made sub-second high-resolution timers to immediately return, which causes userspace applications that use them in loops to instead run in tight loops eating up CPU

5. ...

\(^1\)\url{http://winningraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html}

\(^2\)\url{https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6b43ae8a619d17c4935c3320d2ef9e92bdeed05d}
Note that previous analysis are not safe because they compute over-approximations of locksets:

```c
lock((struct*)->mutex);
...
unlock((struct*)->mutex);
```

We should keep track of mutexes that *must* be held instead.
Locksmith
Detecting race conditions: Locksmith

An interesting safe functional programming language approach:


Basic idea of **correlation analysis**: ensure that for every shared memory location there is a lock protecting it.

- they use a polymorphic λ-calculus for this (with C backend)
- Locksmith is implemented in OCaml
- open-source³

³http://www.cs.umd.edu/projects/PL/locksmith/
Correlation between locks and memory locations

Typical example:

```c
pthread_mutex_t L1 = ..., L2 = ...;
int x, y, z;

void munge(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
... 

munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```

The **correlation** is:

- \(x \triangleright L1\)
- \(y \triangleright L2\)
- \(z \triangleright L2\)
Typing system

They have a typing system (with subtyping) with rules of the form

\[ C; \Gamma \vdash e : \tau; \varepsilon \]

where

- \( C \) is a set of constraints
- \( \Gamma \) is a list of type assumptions
- \( e \) is an expression
- \( \tau \) a type
- \( \varepsilon \) an effect

An algorithm propagates the constraints (which takes care of aliasing) and ensure that they are satisfiable.
Finding bugs

They run on medium-sized C programs, e.g. the Aget ftp client:

Possible data race on
\&bwritten(aget_comb.c:943)
References:
dereference at aget_comb.c:1079
locks acquired at dereference:
\&bwritten_mutex(aget_comb.c:996)
in: FORK at aget_comb.c:468 ->
http_get aget_comb.c:468

dereference at aget_comb.c:984
locks acquired at dereference:
(none)
in: FORK at aget_comb.c:193 ->
signal_waiter(aget_comb.c:193) ->
sigalrm_handler(aget_comb.c:957)
Goblint
Another example of a safe tool is Goblint:


It is:

- programmed in OCaml
- open-source
- Eclipse compatible

---

4 [http://goblint.in.tum.de/](http://goblint.in.tum.de/)
5 [https://github.com/goblint/analyzer](https://github.com/goblint/analyzer)
6 [https://github.com/goblint/bench](https://github.com/goblint/bench)
int global;

void race() { global++; }
void nice() { printf("mu"); }
void (*f)() = nice;

void *tfun(void *arg) {
    f();
    return NULL;
}

int main() {
    pthread_create(tfun);
    f = race;
    global++;
    return 0;
}
Idea

- We analyze each thread in separation, identifying the effect it has on the rest of the program (through modification of variables).
- When updating variables, trigger re-evaluation of impacted portions of code.

In our example,

- \texttt{tfun} is claimed to be safe at the first analysis, but we note it depends on \texttt{f}
- when the \texttt{main} updates \texttt{f} we re-analyze \texttt{tfun} and join the result of this analysis with the previous one
In practice

- They have some (simple) abstract interpretation (e.g. cofinite sets of \( \mathbb{N} \)).
- They use a general-purpose constraint solver in order to compute the fixpoint.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (kloc)</th>
<th>Goblint</th>
<th></th>
<th>Locksmith</th>
<th></th>
<th>Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>aget</td>
<td>1.2</td>
<td>0.3</td>
<td>5</td>
<td>1.0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>knot</td>
<td>1.3</td>
<td>0.3</td>
<td>7</td>
<td>9.1</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>pfscan</td>
<td>1.3</td>
<td>0.1</td>
<td>2</td>
<td>0.6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>1.4</td>
<td>0.3</td>
<td>2</td>
<td>3.0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>smtprc</td>
<td>5.7</td>
<td>12</td>
<td>2</td>
<td>8.2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Fast and Accurate Static Data-Race Detection for Concurrent Programs, Kahlion & Yang & Sankaranarayanan & Gupta, CAV, 2007 (77¢).

Conditional Must Not Aliasing for Static Race Detection, Naik & Aiken, POPL, 2007 (186¢).
SYSTEMATIC EXPLORATION
Race freedom is not enough

Consider the bank program:

```java
int balance;

synchronized void deposit(int n) { balance += n; }

synchronized int read() { return balance; }

void withdraw(int n) {
    int r = read();
    synchronized(this) { balance -= n; }
}
```

Consider the possible executions of

```java
deposit(10) || withdraw(10)
```

Note that it is race-free!
A bug in java.lang.StringBuffer (jdk 1.4)

The methods of StringBuffer are synchronized but\textsuperscript{7}...

```java
public final class StringBuffer {
    private int count;
    private char[] value;

    public synchronized StringBuffer append (StringBuffer sb) {
        int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length) expandCapacity(newcount);
        sb.getChars(0, len, value, count); // bad len!!??
        count = newcount;
        return this;
    }

    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
}

\textsuperscript{7}http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4810210
Exploring all the schedules

The basic idea is to explore all the schedules (or a representative set of schedules) in order to ensure that things cannot go wrong.

We have to find a way to limit the number of interleavings.

Also, we have to assume deterministic inputs, a reproducible set of tests, etc.
SKI
Finding bugs by exploration

A possible approach to find bugs is to explore schedulings. In


This tool

- is a VM to run various schedulings of a (unmodified) kernel
- detects liveness of threads and randomizes schedulings
- monitors kernel’s error messages
- monitors fs corruption (through *fsck*)
- detects racing memory accesses: pauses a threads at a read and see whether other read at the same location
## Finding bugs by exploration

<table>
<thead>
<tr>
<th>Bug</th>
<th>Kernel</th>
<th>FS</th>
<th>Function</th>
<th>Detector / Failure</th>
<th>E</th>
<th>FS</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.11.1</td>
<td>Btrfs</td>
<td>btrfs_find_all_root()</td>
<td>Crash: Null-pointer</td>
<td>41</td>
<td>0.030</td>
<td>Fixed</td>
</tr>
<tr>
<td>2</td>
<td>3.11.1</td>
<td>Btrfs</td>
<td>run_clustered.refs()</td>
<td>Crash: Null-pointer + Warning</td>
<td>26</td>
<td>0.020</td>
<td>Fixed</td>
</tr>
<tr>
<td>3</td>
<td>3.11.1</td>
<td>Btrfs</td>
<td>record_one_backref()</td>
<td>Warning</td>
<td>74</td>
<td>0.030</td>
<td>Fixed</td>
</tr>
<tr>
<td>4</td>
<td>3.11.1</td>
<td>Btrfs</td>
<td>NA</td>
<td>Fsck: Refs. not found</td>
<td>11</td>
<td>0.200</td>
<td>Reported</td>
</tr>
<tr>
<td>5</td>
<td>3.12.2+p</td>
<td>Btrfs</td>
<td>btrfs_find_all_root()</td>
<td>Crash: Null pointer</td>
<td>61</td>
<td>0.060</td>
<td>Fixed</td>
</tr>
<tr>
<td>6</td>
<td>3.12.2</td>
<td>Btrfs</td>
<td>inode_tree_add()</td>
<td>Warning</td>
<td>53</td>
<td>0.010</td>
<td>Fixed</td>
</tr>
<tr>
<td>7</td>
<td>3.13.5</td>
<td>Logfs</td>
<td>indirect_write_alias()</td>
<td>Crash: Null pointer</td>
<td>31</td>
<td>0.065</td>
<td>Reported</td>
</tr>
<tr>
<td>8</td>
<td>3.13.5</td>
<td>Logfs</td>
<td>btree_write_alias()</td>
<td>Crash: Invalid paging</td>
<td>142</td>
<td>0.020</td>
<td>Reported</td>
</tr>
<tr>
<td>9</td>
<td>3.13.5</td>
<td>Jfs</td>
<td>lbmIODone()</td>
<td>Crash: Assertion</td>
<td>74</td>
<td>0.005</td>
<td>Reported</td>
</tr>
<tr>
<td>10</td>
<td>3.13.5</td>
<td>Ext4</td>
<td>ext4_do_update_inode()</td>
<td>Data race</td>
<td>32</td>
<td>0.005</td>
<td>Fixed</td>
</tr>
<tr>
<td>11</td>
<td>3.13.5</td>
<td>VFS</td>
<td>generic_fillattr()</td>
<td>Data race</td>
<td>125</td>
<td>0.005</td>
<td>Reported</td>
</tr>
</tbody>
</table>

where

- **E**: number of schedules to expose the bug
- **FS**: fraction of schedules exposing the bug
CHESS
The purpose of CHESS is to explore a representative number of scheduling (but not in the safe sense)


It is apparently extensively used at Microsoft.
Preemptive context switches

There are two kinds of context (=thread) switches:

- **non-preemptive**: at a point where the thread yields explicitly (a yield, locking a locked mutex, waiting for an input, etc.)
- **preemptive**: at any time (kernel is the king)
Preemptive context switches

There are two kinds of context (=thread) switches:

- **non-preemptive**: at a point where the thread yields explicitly (a yield, locking a locked mutex, waiting for an input, etc.)
- **preemptive**: at any time (kernel is the king)

Idea of **iterative context-bounding**: explore all the traces by increasing number of preemptive context-switches (which are placed at accesses to variables).

Observation: bugs usually manifest with few (less than 3) preemptions.
### Results

<table>
<thead>
<tr>
<th>Programs</th>
<th>LOC</th>
<th>Max Num Threads</th>
<th>Max $K$</th>
<th>Max $B$</th>
<th>Max $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth</td>
<td>400</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>File System Model</td>
<td>84</td>
<td>4</td>
<td>20</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>Work Stealing Q.</td>
<td>1266</td>
<td>3</td>
<td>99</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>APE</td>
<td>18947</td>
<td>4</td>
<td>247</td>
<td>2</td>
<td>75</td>
</tr>
<tr>
<td>Dryad Channels</td>
<td>16036</td>
<td>5</td>
<td>273</td>
<td>4</td>
<td>167</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programs</th>
<th>Total Bugs</th>
<th>Bugs with Context Bound</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Work Stealing Queue</td>
<td>3</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Transaction Manager</td>
<td>3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>APE</td>
<td>4</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dryad Channels</td>
<td>5</td>
<td></td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2.** For a total of 14 bugs that our model checker found, this table shows the number of bugs exposed in executions with exactly $c$ preemptions, for $c$ ranging from 0 to 3. The 7 bugs in the first three programs were previously known. Iterative context-bounding algorithm found the 9 previously *unknown* bugs in Dryad and APE.
Variants

There are variants such as **delay bounding**: we take a deterministic scheduler and count the number of time we can switch to next available thread. Both are compared in


who used some benchmarks which are available on the web

http://sites.google.com/site/sctbenchmarks

https://github.com/sctbenchmarks/sctbenchmarks
Results

Bugs found:

- IPB: preemption bounding
- IDB: delay bounding
- DFS: depth-first search
Remark

Some other interesting benchmarks can be found here

https://github.com/sosy-lab/sv-benchmarks

(from the SV-COMP competition on software verification)
RaceFuzzer
Another idea is to orient the scheduler in order to favor bugs:


what they call **race-directed random testing**.
Algorithm

- find pairs of read / write which could potentially occur together (a rough approximation is enough, you can use happens-before relation in order to remove those which can trivially never occur at the same time)
- for each of those pairs, randomly schedule until one of the two occurs:
  - if one occurs, block the thread and hope that the other thread will perform the other action.
Example

Initially: \( x = y = z = 0 \);

```c
thread1 {
1: \ x = 1;
2: \ lock(L);
3: \ y = 1;
4: \ unlock(L);
5: \ if \ (z==1) \{
6: \ \quad \text{ERROR1;}
\}
}
thread2 {
7: \ z = 1;
8: \ lock(L);
9: \ if \ (y==1) \{
10: \quad \text{if} \ (x \neq 1)\{
11: \quad \quad \text{ERROR2;}
12: \quad \}
13: \}
14: \ unlock(L);
}
```

Figure 1. A program with a real race

- \( x/x \): there is no race on \( x \)
- \( z/z \): there is a race on \( z \)
# Results

<table>
<thead>
<tr>
<th>Program Name</th>
<th>SLOC</th>
<th>Average Runtime in sec.</th>
<th># of Races</th>
<th># of Exceptions</th>
<th>Probability of hitting a race</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal</td>
<td>Hybrid</td>
<td>RF</td>
<td>Hybrid</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,352</td>
<td>2.07</td>
<td>&gt; 3600</td>
<td>42.37</td>
<td>59</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,924</td>
<td>3.25</td>
<td>&gt; 3600</td>
<td>3.81</td>
<td>2</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,619</td>
<td>3.48</td>
<td>&gt; 3600</td>
<td>6.44</td>
<td>5</td>
</tr>
<tr>
<td>cache4j</td>
<td>3,897</td>
<td>2.19</td>
<td>4.26</td>
<td>2.61</td>
<td>18</td>
</tr>
<tr>
<td>sor</td>
<td>17,689</td>
<td>0.16</td>
<td>0.35</td>
<td>0.23</td>
<td>8</td>
</tr>
<tr>
<td>hede</td>
<td>29,948</td>
<td>1.10</td>
<td>1.35</td>
<td>1.11</td>
<td>9</td>
</tr>
<tr>
<td>weblech</td>
<td>35,175</td>
<td>0.91</td>
<td>1.92</td>
<td>1.36</td>
<td>27</td>
</tr>
<tr>
<td>jspider</td>
<td>64,933</td>
<td>4.79</td>
<td>4.88</td>
<td>4.81</td>
<td>29</td>
</tr>
<tr>
<td>jigsaw</td>
<td>381,348</td>
<td>-</td>
<td>-</td>
<td>0.81</td>
<td>547</td>
</tr>
<tr>
<td>vector 1.1</td>
<td>709</td>
<td>0.11</td>
<td>0.25</td>
<td>0.2</td>
<td>9</td>
</tr>
<tr>
<td>LinkedList</td>
<td>5979</td>
<td>0.16</td>
<td>0.26</td>
<td>0.22</td>
<td>12</td>
</tr>
<tr>
<td>ArrayList</td>
<td>5866</td>
<td>0.16</td>
<td>0.26</td>
<td>0.24</td>
<td>14</td>
</tr>
<tr>
<td>HashSet</td>
<td>7086</td>
<td>0.16</td>
<td>0.26</td>
<td>0.25</td>
<td>11</td>
</tr>
<tr>
<td>TreeSet</td>
<td>7532</td>
<td>0.17</td>
<td>0.26</td>
<td>0.24</td>
<td>13</td>
</tr>
</tbody>
</table>
PARTIAL ORDER REDUCTION
Happens-before

An important point tool is **happens-before** relation defined in


Key property: $a \leq b$ implies $T(a) \leq T(b)$
Race freedom

In order to check that a program is race-free, we can check that any two write and reads at same location are ordered, for any happens before relation (sequentializing blocking sections, or send/receives).
DPOR
Partial order reduction

An example

With $n$ threads tid inserting message $w$ in hash table:

```c
while (true) {
    w := getmsg();
    h := hash(w);
    while (cas(table[h], 0, w) == false) {h := (h+1) % size;}
}
```

```c
int getmsg() {
    if (m < max ) {return (++m) * 11 + tid;}
    else {exit();}
}
```

```c
int hash(int w) {
    return (w * 7) % size;
}
```

*Alias* analysis is impossible because is depends on messages.
Notations

Given a trace $S$ (a sequence of transitions)

- $S_i$: $i$-th transition
- $dom(S)$: number of transitions in $S$
- $pre(S, i)$: source state of $S_i$
- $last(S)$: target state of $S$

It induces a *happens-before* relation $\rightarrow_S$ which is the smallest partial order relation such that, for $i \leq j$, $S_i$ and $S_j$ are dependent (do not commute) implies $i \rightarrow_S j$.

We also write $i \rightarrow_S p$ when there exists $j$ with $i \rightarrow_S j$ and $proc(S_j) = p$. 
The algorithm

0 Initially: Explore(∅);

1 Explore(S) {
    2     let s = last(S);
    3     for all processes p {
    4         if ∃ i = max({i ∈ dom(S) | S_i is dependent and may be co-enabled with next(s, p) and i ∉ S p}) {
    5             let E = {q ∈ enabled(pre(S, i)) | q = p or ∃ j ∈ dom(S) : j > i and q = proc(S_j) and j → S p};
    6             if (E ≠ ∅) then add any q ∈ E to backtrack(pre(S, i));
    7                 else add all q ∈ enabled(pre(S, i)) to backtrack(pre(S, i));
    8         }
    9     }
  10     if (∃ p ∈ enabled(s)) {
  11         backtrack(s) := {p};
  12         let done = ∅;
  13         while (∃ p ∈ (backtrack(s) \ done)) {
  14             add p to done;
  15             Explore(S.next(s, p));
  16         }
  17     }
  18 }

72 / 92
A simple example
Consider two processes

\[(x = 1; x = 2) \parallel (y = 1; x = 3)\]

▶ first exploration:

\[x = 1; x = 2; y = 1; x = 3\]
A simple example

Consider two processes

\[(x = 1; x = 2) \parallel (y = 1; x = 3)\]

- first exploration:

  \[x = 1; x = 2; y = 1; x = 3\]

- before executing \(x = 3\), we see that it depends with \(x = 2\),
  we thus backtrack:

  \[x = 1; x = 3; x = 2; y = 1\]
A simple example

Consider two processes

\[(x = 1; x = 2) \parallel (y = 1; x = 3)\]

▸ first exploration:

\[x = 1; x = 2; y = 1; x = 3\]

▸ before executing \(x = 3\), we see that it depends with \(x = 2\), we thus backtrack:

\[x = 1; x = 3; x = 2; y = 1\]

▸ again \(x = 3\) depends with \(x = 1\) and we backtrack:

\[y = 1; x = 3; x = 1; x = 2\]
An example: a file system

```c
i := tid % NUMINODE;
acquire(locki[i]);
if (inode[i] == 0) {
    b := (i*2) % NUMBLOCKS;
    while (true) {
        acquire(lockb[b]);
        if (!busy[b]) {
            busy[b] := true;
            inode[i] := b+1;
            release(lockb[b]);
            break;
        }
        release(lockb[b]);
        b := (b+1)%NUMBLOCKS;
    }
    release(lockb[b]);
    b := (b+1)%NUMBLOCKS;
}
release(locki[i]);
```
An example: a file system
ReEx
Maximal causality: ReEx

It is noticed in


that DPOR is sometimes “too local”.
An example

initially x=y=0

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop twice:</td>
<td>loop twice:</td>
<td>loop twice:</td>
</tr>
<tr>
<td>1: lock(l)</td>
<td>5: lock(l)</td>
<td>11: if(x&gt;1)</td>
</tr>
<tr>
<td>2: x=1</td>
<td>6: x=0</td>
<td>12: if(y==3)</td>
</tr>
<tr>
<td>3: y=1</td>
<td>7: unlock(l);</td>
<td>13: Error</td>
</tr>
<tr>
<td>4: unlock(l);</td>
<td>8: if(x&gt;0)</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>9: y++</td>
<td>14: y=2</td>
</tr>
<tr>
<td></td>
<td>10: x=2</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{event at line } i \text{ in loop } j \]

\[ i=1, 2, \ldots, 14 \quad j=1, 2 \]

\[ L_i^j / U_i^j : \text{lock/unlock} \]

\[ R_i^j / W_i^j : \text{read/write} \]

error-triggering schedule

\[ T2 - T2 - T2 - T1 - T1 - T1 - T2 - T2 - T2 - T2 - T2 - T3 - T3 - T3 - T3 - T2 - T2 - T2 - T2 - T2 - T2 - T2 - T3 - T3 \\
L_5^1 - W_6^1 - U_7^1 - L_4^1 - W_2^1 - W_3^1 - U_4^1 - R_8^1 - R_9^1 - W_9^1 - W_{10}^1 - R_{11}^1 - R_{12}^1 - W_{14}^1 - L_5^2 - W_6^2 - U_7^2 - L_4^2 - W_2^2 - R_8^2 - R_9^2 - W_9^2 - W_{10}^2 - R_{11}^2 - R_{12}^2 \]

For **Error** to happen:

- L2: L9 must be executed before L14
- L1: L2 must be executed between L7 and L8

With CHESS, we need 58478 schedules to hit it...
The algorithm

1. record the trace from one execution, recalling information to construct the “maximal causal model”
2. generate causally different schedules: a read event reads new data
3. execute the generated schedules
The maximal causal schedule

To any trace $\tau$ (with fork/join) one can associate the maximal set of causally consistent traces $\sigma$, with same events, which can be encoded by a first-order formula:

- **must happen-before**: the events in a given thread of $\sigma$ should be in the same order as in $\tau$

- **locking constraints**: two sequences of instructions protected by the same lock cannot be interleaved

- **read-write constraints**: any read event in $\sigma$ should read the same value as in $\tau$ (read-write dependency)
Schedule generation

- once a trace has been explored, it is not necessary to explore another trace in the same maximally consistent set
- we thus explore a different trace, i.e. one in which a read gets a different value
- they thus change a read-write pair and check whether it is feasible (using the Z3 constraint solver)
- they prune before in order to exclude “obviously” unfeasible schedules
Remarks

- This is claimed to be better than DPOR because in

\[(x = 1; x = 2) \parallel (y = 1; x = 3)\]

only one trace has to be explored (since nobody reads...)

NB: but, one does usually write for nothing...

- This is claimed to be safe

- There is no sharing between schedulings

- Reminds me of Uli’s partial orders
Lipton
Lipton reduction

Another kind of reduction was proposed in


whose main idea is that some sequences of instructions can sometimes be merged into one.
Going back to the bank / StringBuffer example, it is claimed in


that the property we want to ensure is **atomicity**, i.e. every execution is equivalent to one where the synchronized methods are executed atomically.

This is the kind of properties Lipton can help to show.
Movers

An action $a$ is a **right-mover** when if we execute $a \cdot b$ where $b$ is an action of a different thread, then executing $b \cdot a$ results in the same state. Dually, a **left-mover**...

<table>
<thead>
<tr>
<th>operation</th>
<th>mover</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>right-mover</td>
</tr>
<tr>
<td>release</td>
<td>left-mover</td>
</tr>
<tr>
<td>protected read/write</td>
<td>both-mover</td>
</tr>
<tr>
<td>unprotected read/write</td>
<td>non-mover</td>
</tr>
</tbody>
</table>

**Theorem**

*An sequence of actions of the form*

\[
\text{right-mover}^* \text{non-mover}^? \text{left-mover}^*
\]

*can be considered as atomic.*
Reduction through symmetry

Another way of reducing the state-space is presented in


by considering its symmetries, i.e. quotienting it under automorphisms.

Typically, an algorithm might not depend on the exact pid of a process.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Algorithm</th>
<th>size</th>
<th>time</th>
<th>% reduction</th>
<th>max possible reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Unreduced</td>
<td>1,694</td>
<td>12s</td>
<td>0%</td>
<td>$1 - \frac{1}{2! \times 2!} = 75%$</td>
</tr>
<tr>
<td></td>
<td>Canonicalized</td>
<td>425</td>
<td>48s</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normalized</td>
<td>429</td>
<td>7s</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Unreduced</td>
<td>91,254</td>
<td>23min</td>
<td>0%</td>
<td>$1 - \frac{1}{3! \times 2!} = 92%$</td>
</tr>
<tr>
<td></td>
<td>Canonicalized</td>
<td>7,741</td>
<td>4.5hr</td>
<td>92%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normalized</td>
<td>9,002</td>
<td>13min</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Unreduced</td>
<td>exceeded 80Mbytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Canonicalized</td>
<td>exceeded 36hr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normalized</td>
<td>206,169</td>
<td>36hr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TODO:

How to publish

► have a (vaguely) new idea
► insist on the fact it’s new
► begin with the usual state-space explosion introduction
► end with a summary of new points
► you should have a big table of benchmarks
► if you go for POPL, have some semantics / inference rules
► remember that you are doing something new and better than others:
  ► support this affirmation with well-chosen (or even crafted) benchmarks
  ► be partial on bibliography
  ► don’t try to understand too deeply what you are doing or other’s papers
► in the future you could be better than the best

But anyway, most of them find real bugs in real programs.
Criteria

- errors checked: data races / deadlocks / exception (*NULL*)
- size of analyzed programs: toy examples / small programs / the kernel
- static or dynamic?
- if static, is it safe?
- techniques: lockset / happens-before / Lipton / abstract interpretation / POR / bug-directed / etc.
- do we have to write a test set?
- language analyzed: C / Java / some esoteric DSL
- open-source?
- language it is programmed in?
What can we do

- which criteria do we want to meet?
- we could think of other kind of verifications (cyber-physical systems?)
- homotopy depends on observations \(\Rightarrow\) a general framework?
- many explorer don’t share between traces or consider only local properties (DPOR)