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

LOCKSET ANALYSIS

Race conditions

We want to detect **race conditions**: unprotected concurrent access to memory (one of which is a write).

Eraser

Locksets: Eraser

Lockset analysis is based on the idea that every accessed shared variable should have a lock associated to it. It was introduced in

- *Eraser: A dynamic data race detector for multithreaded programs*, Savage & Burrows & Nelson & Sobalvarro, ACM Transactions on Computer Systems, 1997 (1443¢).

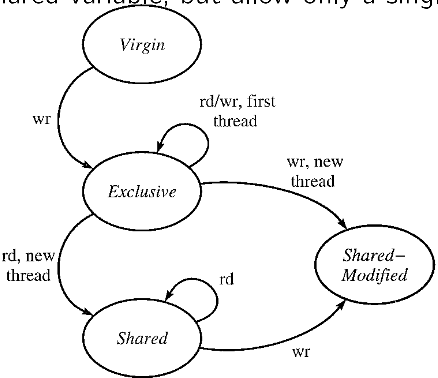
They use a dynamic analysis:

> Let $locks\_held(t)$ 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,
>   set $C(v) := C(v) \cap locks\_held(t)$;
>   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:

```
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:

```
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*
  `void mutex_init(struct mutex *lock);`

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

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

- *check the state of a mutex*
  `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 <u>never sleep</u>.

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 <u>never sleep</u>.

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 dm365mma
Backtrace:
[<c002a5a0>] (dump_backtrace+0x0/0x110) from [<c028e56c>] (dump_stack+0
 r6:c1099460 r5:c04ea000 r4:00000000 r3:20000013
[<c028e554>] (dump_stack+0x0/0x1c) from [<c00337b8>] (__schedule_bug+0x
[<c0033760>] (__schedule_bug+0x0/0x64) from [<c028e864>] (schedule+0x84
 r4:c10992c0 r3:00000000
[<c028e7e0>] (schedule+0x0/0x378) from [<c0033a80>] (__cond_resched+0x2
[<c0033a58>] (__cond_resched+0x0/0x38) from [<c028ec6c>] (_cond_resched
 r4:00013000 r3:00000001
[<c028ec38>] (_cond_resched+0x0/0x44) from [<c0082f64>] (unmap_vmas+0x5
[<c00829f4>] (unmap_vmas+0x0/0x620) from [<c0085c10>] (exit_mmap+0xc0/0
[<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*
  `void spin_lock_init(spinlock_t *lock);`
- *locking*
  `void spin_lock(spinlock_t *lock);`
- *releasing*
  `void spin_unlock(spinlock_t *lock);`

Spinlocks implementation

```
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!

- *Static deadlock detection in the Linux kernel*, Breuer & MG Valls, International Conference on Reliable Software, 2004 (25¢).
- *Detecting deadlock, double-free and other abuses in a million lines of linux kernel source*, Breuer & Pickin & Petrie, 30th Annual IEEE/NASA Software Engineering Workshop, 2006 (13¢).

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)!

Detecting scheduling while atomic

A portion of code is **spinlocked** if it is of the form

```
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:

$$N(c) \quad : \quad \mathbb{N} \quad \rightarrow \quad \mathbb{N}$$

defined by

- $N(a; b)(n) = N(b)(N(a)(n))$
- $N(\texttt{spin\_lock(...)})(n) = n + 1$
- $N(\texttt{spin\_unlock(...)})(n) = n - 1$
- $N(\texttt{f(...)})(n) = n$
- $N(\texttt{if}\, a \texttt{ then } b \texttt{ else } c)(n) = \max(N(b)(n), N(c)(n))$
- $N(\texttt{while } a\ b)(n) = \max(n, N(b)(n) \times \infty)$

Detecting scheduling while atomic

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

$$N(c) \quad : \quad \mathbb{N} \quad \to \quad \mathbb{N}$$

defined by

- $N(a; b)(n) = N(b)(N(a)(n))$
- $N(\texttt{spin\_lock(...)})(n) = n + 1$
- $N(\texttt{spin\_unlock(...)})(n) = n - 1$
- $N(\texttt{f(...)})(n) = n$
- $N(\texttt{if} a \texttt{ then } b \texttt{ else } c)(n) = \max(N(b)(n), N(c)(n))$
- $N(\texttt{while } a\ b)(n) = \max(n, N(b)(n) \times \infty)$

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 adapated 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`:

```
...
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 `midi_outc()` of `sound/oss/sequencer.c`:

```
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);
```

RacerX

More on locksets: RacerX

A similar analysis is performed in

- *RacerX: effective, static detection of race conditions and deadlocks*, Engler & Ashcraft, ACM SIGOPS Operating Systems Review, 2003 (674¢).

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

Boum

```
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;

<rtc_task_lock>-><rtc_lock> =
  depth = 1:
    linux-2.5.62/drivers/char/rtc.c:rtc_unregister:749
      ->rtc.c: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

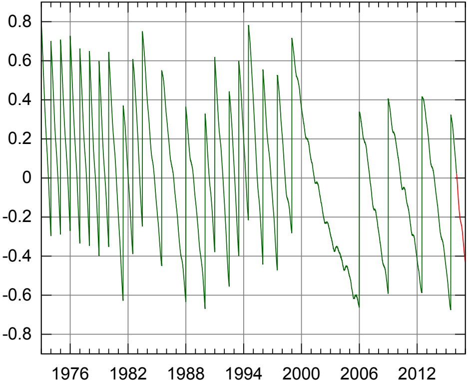
We can also detect the "sleeping under spinlock" error:

```
//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\
  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 where (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()`

[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 where (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

[2] 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 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

[2] 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 `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]
4. actually removing `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] 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 where (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
   ⇒ 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]
4. actually removing 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] http://winninggraceconditions.blogspot.fr/2012/07/linuxs-leap-second-deadlocks.html

[2] 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:

```
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:

- Locksmith*: context-sensitive correlation analysis for race detection*, Pratikakis & Foster & Hicks, ACM SIGPLAN Notices, 2006 (219¢).

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-source[3]

[3] `http://www.cs.umd.edu/projects/PL/locksmith/`

Correlation between locks and memory locations

Typical example:

```
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

$$\mathrm{x} \triangleright \mathrm{L1} \qquad \mathrm{y} \triangleright \mathrm{L2} \qquad \mathrm{z} \triangleright \mathrm{L2}$$

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
- Γ is a list of type assumptions
- e is an expression
- τ a type
- ε 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

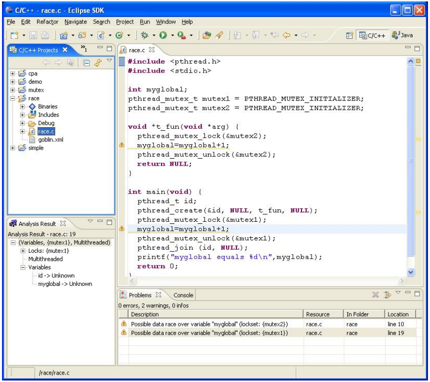
Goblint

Another example of safe tool is Goblint:

- *Goblint: Path-sensitive data race analysis*, Vojdani & Vene, Annales Univ. Sci. Budapest., 2009 (24¢).

it is

- programmed in OCaml
- open-source[4][5][6]
- Eclipse compatible



---

[4] http://goblint.in.tum.de/

[5] https://github.com/goblint/analyzer

[6] https://github.com/goblint/bench

Typical example

```
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,

- `tfun` is claimed to be safe at the first analysis, but we note it depends on `f`
- when the `main` updates `f` we re-analyze `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.

Benchmark	Size (kloc)	Goblint		LOCKSMITH		Races
		Time	Warn.	Time	Warn	
aget	1.2	0.3	5	1.0	4	4
knot	1.3	0.3	7	9.1	8	7
pfscan	1.3	0.1	2	0.6	2	0
ctrace	1.4	0.3	2	3.0	2	0
smtprc	5.7	12	2	8.2	0	0

TODO

TODO:

- *Fast and Accurate Static Data-Race Detection for Concurrent Programs*, Kahlon & 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:

```
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

$$\texttt{deposit(10)} \quad \| \quad \texttt{withdraw(10)}$$

Note that it is race-free!

A bug in java.lang.StringBuffer (jdk 1.4)

The methods of StringBuffer are synchronized but[7]...

```
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(...) { ... }
}
```

[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

- *SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration*, Fonseca & Rodrigues & Brandenburg, 11th USENIX Symposium on Operating Systems Design and Implementation, 2014 (10¢).

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

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

where

- E: number of schedules to expose the bug
- FS: fraction of schedules exposing the bug

CHESS

CHESS

The purpose of CHESS is to explore a representative number of scheduling (but not in the *safe* sense)

- *Iterative Context Bounding for Systematic Testing of Multithreaded Programs*, Musuvathi & Qadeer, PLDI, 2007 (370¢).
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi & Qadeer & Ball & Basler & Nainar Arumuga & Neamtiu, OSDI, 2008 (443¢).

It is apparently extensively used at Microsoft.

Preemptive context switches

There are two kind 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 kind 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

Programs	LOC	Max Num Threads	Max K	Max B	Max c
Bluetooth	400	3	15	2	8
File System Model	84	4	20	8	13
Work Stealing Q.	1266	3	99	2	35
APE	18947	4	247	2	75
Dryad Channels	16036	5	273	4	167

Programs	Total Bugs	Bugs with Context Bound			
		0	1	2	3
Bluetooth	1	0	1	0	0
Work Stealing Queue	3	0	1	2	0
Transaction Manager	3	0	0	2	1
APE	4	2	1	1	0
Dryad Channels	5	1	4	0	0

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

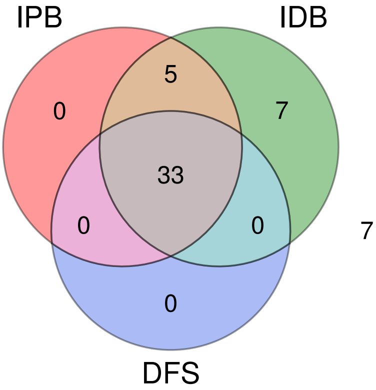
- *Concurrency Testing Using Schedule Bounding: an Empirical Study*, Thomson & Donaldson & Betts, PPoPP, 2014 (29¢).

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

RaceFuzzer

Another idea is to orient the scheduler in order to favor bugs:

- *Race Directed Random Testing of Concurrent Programs*, Sen, PLDI, 2008 (270¢).

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 tread and hope that the other tread will perform the other action.

Example

```
     Initially: x = y = z = 0;

thread1 {              thread2 {
1:  x = 1;             7:   z = 1;
2:  lock(L);           8:   lock(L);
3:  y = 1;             9:   if (y==1) {
4:  unlock(L);         10:     if (x != 1){
                       11:        ERROR2;
5:  if (z==1)          12:     }
6:    ERROR1;          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

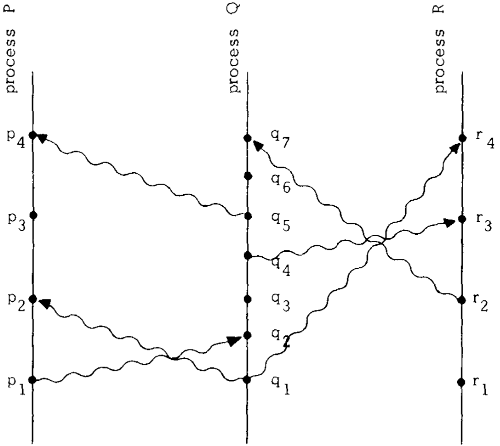
Program Name	SLOC	Average Runtime in sec.			# of Races			# of Exceptions		Probability of hitting a race
		Normal	Hybrid	RF	Hybrid	RF (real)	known	RF	Simple	
moldyn	1,352	2.07	> 3600	42.37	59	2	0	0	0	1.00
raytracer	1,924	3.25	> 3600	3.81	2	2	2	0	0	1.00
montecarlo	3,619	3.48	> 3600	6.44	5	1	1	0	0	1.00
cache4j	3,897	2.19	4.26	2.61	18	2	-	1	0	1.00
sor	17,689	0.16	0.35	0.23	8	0	0	0	0	-
hedc	29,948	1.10	1.35	1.11	9	1	1	1	0	0.86
weblech	35,175	0.91	1.92	1.36	27	2	1	1	1	0.83
jspider	64,933	4.79	4.88	4.81	29	0	-	0	0	-
jigsaw	381,348	-	-	0.81	547	36	-	0	0	0.90
vector 1.1	709	0.11	0.25	0.2	9	9	9	0	0	0.94
LinkedList	5979	0.16	0.26	0.22	12	12	-	5	0	0.85
ArrayList	5866	0.16	0.26	0.24	14	7	-	7	0	0.55
HashSet	7086	0.16	0.26	0.25	11	11	-	8	1	0.54
TreeSet	7532	0.17	0.26	0.24	13	8	-	8	1	0.41

PARTIAL ORDER REDUCTION

Happens-before

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

- *Time, clocks, and the ordering of events in a distributed system*, Lamport, Communications of the ACM, 1978 (9914¢).



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

- *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, Godefroid, Springer, 1996 (1098¢).
- *Dynamic Partial-Order Reduction for Model Checking Software*, Flanagan & Godefroid, POPL, 2005 (506¢).

An example

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

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

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

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 $\to_S$ which is the smallest partial order relation such that, for $i \leq j$, S_i and S_j are dependent (do not commute) implies $i \to_S j$.

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

The algorithm

Initially: Explore($\emptyset$);

Explore(S) {
let $s = last(S)$;
for all processes p {
if $\exists i = max(\{i \in dom(S) \mid S_i$ is dependent and may be co-enabled with $next(s,p)$ and $i \not\to_S p\})$ {
let $E = \{q \in enabled(pre(S,i)) \mid q = p$ or $\exists j \in dom(S) : j > i$ and $q = proc(S_j)$ and $j \to_S p\}$;
if $(E \neq \emptyset)$ then add any $q \in E$ to $backtrack(pre(S,i))$;
else add all $q \in enabled(pre(S,i))$ to $backtrack(pre(S,i))$;
}
}
if $(\exists p \in enabled(s))$ {
$backtrack(s) := \{p\}$;
let $done = \emptyset$;
while $(\exists\ p \in (backtrack(s) \setminus done))$ {
add p to $done$;
Explore($S.next(s,p)$);
}
}
}

A simple example

Consider two processes

$$(x = 1; x = 2) \qquad \| \qquad (y = 1; x = 3)$$

- first exploration:

$$x = 1; x = 2; y = 1; x = 3$$

A simple example

Consider two processes

$$(x = 1; x = 2) \qquad \| \qquad (y = 1; x = 3)$$

- 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

$$(x = 1; x = 2) \qquad \| \qquad (y = 1; x = 3)$$

- 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:

$$x = 1; x = 3; x = 2; y = 1$$

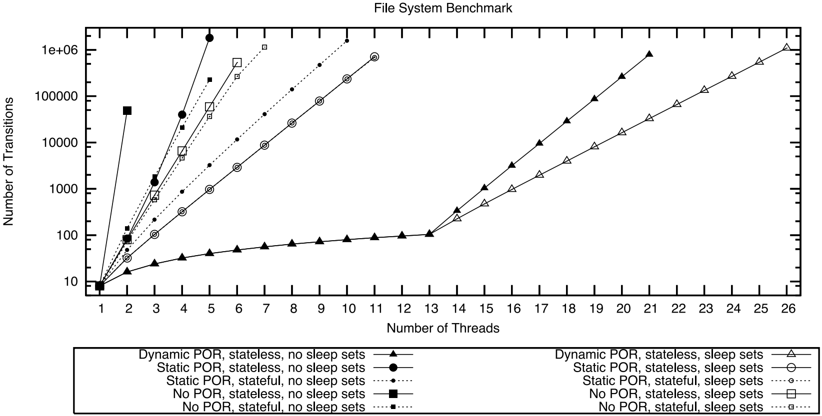
- again $x = 3$ depends with $x = 1$ and we backtrack:

$$y = 1; x = 3; x = 1; x = 2$$

An example: a file system

```
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(locki[i]);
```

An example: a file system



ReEx

Maximal causality: ReEx

It is noticed in

- *Systematic Concurrency Testing with Maximal Causality*, Luo & Huang & Rosu, 2015.

that DPOR is sometimes “too local”.

An example

initially x=y=0

T1	**T2**	**T3**
loop twice:	*loop twice:*	*loop twice:*
1: lock(l)	5: lock(l)	11: if(x>1){
2: x=1	6: x=0	12: if(y==3){
3: y=1	7: unlock(l);	13: ***Error***
4: unlock(l);	8: if(x>0){	else
	9: y++	14: y=2
	10: x=2	}
	}	}

event at line i in loop j

$i=1,2,\ldots,14 \quad j=1,2$

L_i^j / U_i^j : *lock/unlock*

R_i^j / W_i^j : *read/write*

error-triggering schedule

$T2 - T2 - T2 - T1 - T1 - T1 - T1 - T2 - T2 - T2 - T2 - T3 - T3 - T3 - T2 - T2 - T2 - T2 - T1 - T1 - T2 - T2 - T2 - T2 - T3 - T3$

$L_5^1 - W_6^1 - U_7^1 - L_1^1 - W_2^1 - W_3^1 - U_4^1 - R_8^1 - R_9^1 - W_9^1 - W_{10}^1 - R_{11}^1 - R_{12}^1 - W_{14}^1 - L_5^2 - W_6^2 - U_7^2 - L_1^2 - W_2^2 - R_8^2 - R_9^2 - W_9^2 - W_{10}^2 - R_{11}^2 - R_{12}^2$

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 τ (with fork/join) one can associate the *maximal set of causally consistent* traces σ, with same events, which can be encoded by a first-order formula:

- *must happen-before*: the events in a given thread of σ should be in the same order as in τ
- *locking constraints*: two sequences of instructions protected by the same lock cannot be interleaved
- *read-write constraints*: any read event in σ should read the same value as in τ (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

$$(x = 1; x = 2) \qquad \| \qquad (y = 1; x = 3)$$

  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

- *Reduction: A Method of Proving Properties of Parallel Programs*, Lipton, Communications of the ACM, 1975 (404¢).

whose main idea is that some sequences of instructions can sometimes be merged into one.

Back to the bank

Going back to the bank / `StringBuffer` example, it is claimed in

- *Atomizer: a dynamic atomicity checker for multithreaded programs*, Flanagan & Freund, POPL, 2004 (404¢).

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**...

operation	mover
lock	right-mover
release	left-mover
protected read/write	both-mover
unprotected read/write	non-mover

Theorem

An sequence of actions of the form

$$\textit{right-mover}^{*}\ \textit{non-mover}^{?}\ \textit{left-mover}^{*}$$

can be considered as atomic.

Reduction through symmetry

Another way of reducing the state-space is presented in

- *Better verification through symmetry*, Ip & Dill, Formal Methods in System Design, 1996 (421¢).

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

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

#Nodes	Algorithm	size	time	% reduction	max possible reduction
2	Unreduced	1,694	12s	0%	$1 - \frac{1}{2! \times 2!} = 75\%$
	Canonicalized	425	48s	75%	
	Normalized	429	7s	75%	
3	Unreduced	91,254	23min	0%	$1 - \frac{1}{3! \times 2!} = 92\%$
	Canonicalized	7,741	4.5hr	92%	
	Normalized	9,002	13min	90%	
4	Unreduced	exceeded 80Mbytes			$1 - \frac{1}{4! \times 2!} = 98\%$
	Canonicalized	exceeded 36hr			
	Normalized	206,169	36hr	—	

TODO:

- *Partial Orders for Efficient Bounded Model Checking of Concurrent Software*, Alglave & Kroening & Tautschnig.

CONCLUSION

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 bester 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)