

Improving Linux Development with better tools

Andi Kleen

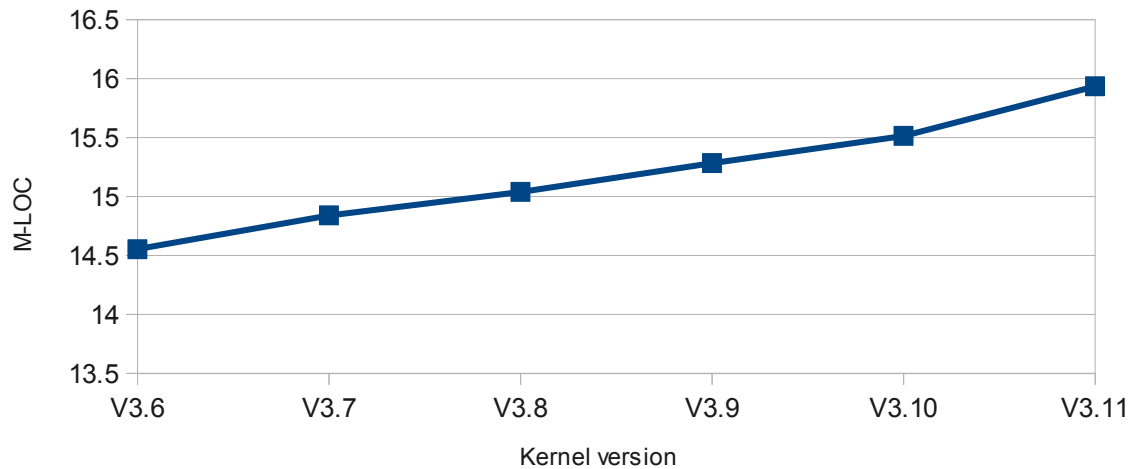
Oct 2013

Intel Corporation
ak@linux.intel.com

Linux complexity growing

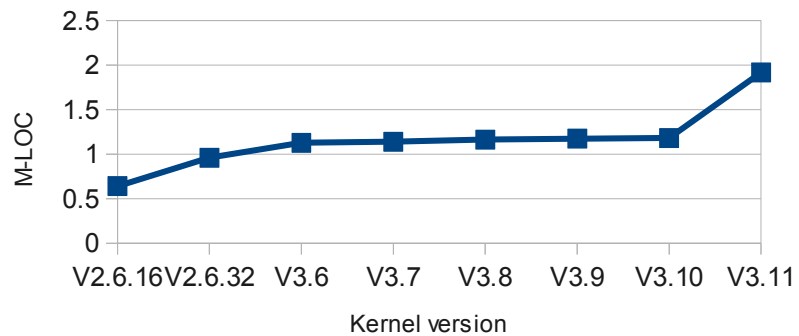
Source lines in Linux kernel

All source code



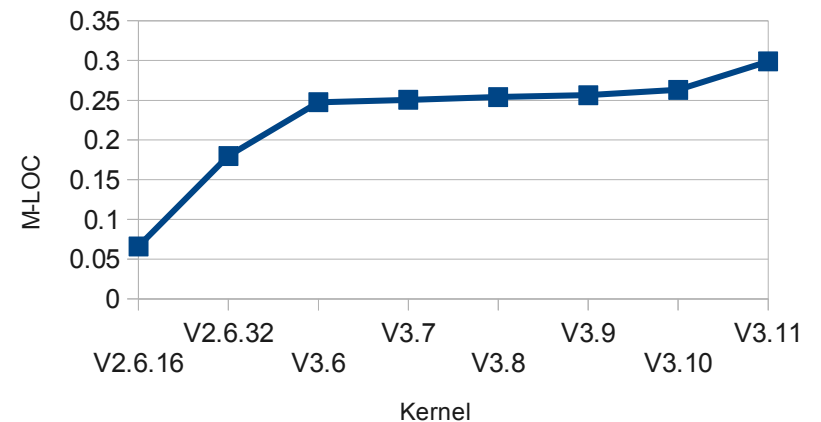
Linux kernel source lines IO

net/ fs/ block/



Source lines Linux Kernel core

kernel/ lib



Do we have a problem?

- If we assume number of bugs stays constant per line there would be more and more bugs
- If we assume programmers don't get cleverer some code may become too complex to change/debug
- Of course modularity saves us to some degree

Or we can use better tools to find bugs

- Static code checker tools
- Dynamic runtime checkers
- Fuzzers/test suites
- Debuggers/Tracers to understand code
- Tools to read/understand source

Static checkers

- sparse, smatch, coccinelle, clang checker, checkpatch, gcc -W/LTO, stanse
- Can check a lot of things, simple mistakes, complex problems
- Generic C and kernel specific rules

Static checker challenges

- Some are very slow
- False positives
 - Often only can do new warnings
 - Otherwise too many false positives
- May need concentrated effort to get false positives down
 - Only done for gcc/sparse so far
 - Needs both changes to Linux and to checkers

Study bug fixes

- “At least 14.8%~24.4% of the sampled bug fixes are incorrect. Moreover, 43% of the incorrect fixes resulted in severe bugs that caused crash, hang, data corruption or security problems.”
 - “How do fixes become bugs” Yin/Yuan et.al.
 - <http://opera.ucsd.edu/~zyin2/fse11.pdf>
 - Great paper, every kernel programmer should read it
- Can new rules for static checkers help?

Cocinelle example

```
/// Find &&|| operations that include the same argument more than once
///  
// A common source of false positives is when the argument performs a side
// effect.  
@r expression@  
expression E;  
position p;  
@@  
(  
  * E@p  
    || ... || E  
  |  
  * E@p  
    && ... && E  
)  
@script:python depends on org@  
p << r.p;  
@@  
cocci.print_main("duplicated argument to && or ||",p)
```


Challenge: global checks

- No static checker I found can follow indirect calls (“OO in C”, common in kernel)

```
struct foo_ops {  
    int (*do_foo)(struct foo *obj);  
}  
  
foo->do_foo(foo);
```

- Can be done by using type information
- Misses a lot of potential bugs

Lock ordering: lockdep

- Deadlock from lock ordering (“ABBA” bugs) used to be common
- Lockdep basically eliminated this problem
- Checks lock ordering, interrupt

Kmemcheck / AddressSanitizer

- Check uninitialized/freed/out of bounds data
- Kmemcheck based on page faults
 - Quite slow
- AddressSanitizer seems to be a better alternative
 - Compiler instrumentation, much faster
 - Still need port to kernel (some reports already)

Thread checkers

- Find data races:
 - Shared data accesses not protected by locks
- User space: helgrind, ThreadSanitizer, ..
- Problem: kernel does not mark lock less accesses.
Solvable?

User:

```
__atomic_write(&foo, 1, __ATOMIC...);
```

– Kernel:

```
Foo = 1;
```

```
mb();
```

•

–

Undefined behavior checker

- UBSan. New gcc/LLVM feature
- Checks undefined C behavior at runtime
 - e.g. $x \ll 100$, signed integer overflows, ...
- Needs special runtime library
- Would need to be ported to kernel

Fuzzers

- Trinity is a great tool
 - Finds many bugs
- Needs manual model for each syscall

How do we cover all the ioctls/sys/proc files?
- Modern fuzzers around using automatic feedback
 - But not for kernel yet
 - http://taviso.decsystem.org/making_software_dumber.pdf
-

The biggest challenge

- How to run all these tools on every new patch:
 - Cannot ask every developer to use all of them
- Static checkers are relatively easy
 - But can we get beyond just deltas for new code?
- But how to run the dynamic tools?

Test suites

- Ideally all kernel code would come with a test suite
 - Then someone could run all the dynamic checkers
- Difficult for hardware drivers
- LKP, kernel unit tests, tools/* limited
- Need a real unit testing framework

Coverage

- Kernel gcov can be used to test coverage of test suites
- Should be used much more widely

Tracers

- Long beyond “real men don't use debuggers”
 - Linux has good debuggers these days (kgdb etc.)
- But how to debug hard to reproduce bugs
 - Ideal enough information to debug on first trigger
- Tracing:
 - Low overhead instrumentation
 - When problem triggers dump data

ftrace: function tracer

- Trace all functions in the kernel for PID

```
# trace-cmd record -p function -e sched_switch -P $(pidof firefox-bin)
  plugin function
disable all
enable sched_switch
path = /sys/kernel/debug/tracing/events/sched_switch/enable
path = /sys/kernel/debug/tracing/events/*/sched_switch/enable
path = /sys/kernel/debug/tracing/events/sched_switch/enable
path = /sys/kernel/debug/tracing/events/*/sched_switch/enable
Hit Ctrl^C to stop recording
```

...

```
# trace-cmd report
```

...

```
firefox-bin-13822 [002] 36628.537061: function:
firefox-bin-13822 [002] 36628.537062: function:
firefox-bin-13822 [002] 36628.537062: function:
firefox-bin-13822 [002] 36628.537062: function:
firefox-bin-13822 [002] 36628.537063: function:
firefox-bin-13822 [002] 36628.537063: function:
firefox-bin-13822 [002] 36628.537063: function:
firefox-bin-13822 [002] 36628.537064: function:
firefox-bin-13822 [002] 36628.537064: function:
firefox-bin-13822 [002] 36628.537065: function:
firefox-bin-13822 [002] 36628.537065: function:
firefox-bin-13822 [002] 36628.537065: function:
firefox-bin-13822 [002] 36628.537065: function:
firefox-bin-13822 [002] 36628.537066: function:
```

...



All kernel functions
executed

```
sys_poll
  poll_select_set_timeout
    ktime_get_ts
      timekeeping_get_ns
        set_normalized_timespec
          timespec_add_safe
            set_normalized_timespec
do_sys_poll
  copy_from_user
    might_fault
      _cond_resched
        should_resched
          need_resched
            test_ti_thread_flag
```

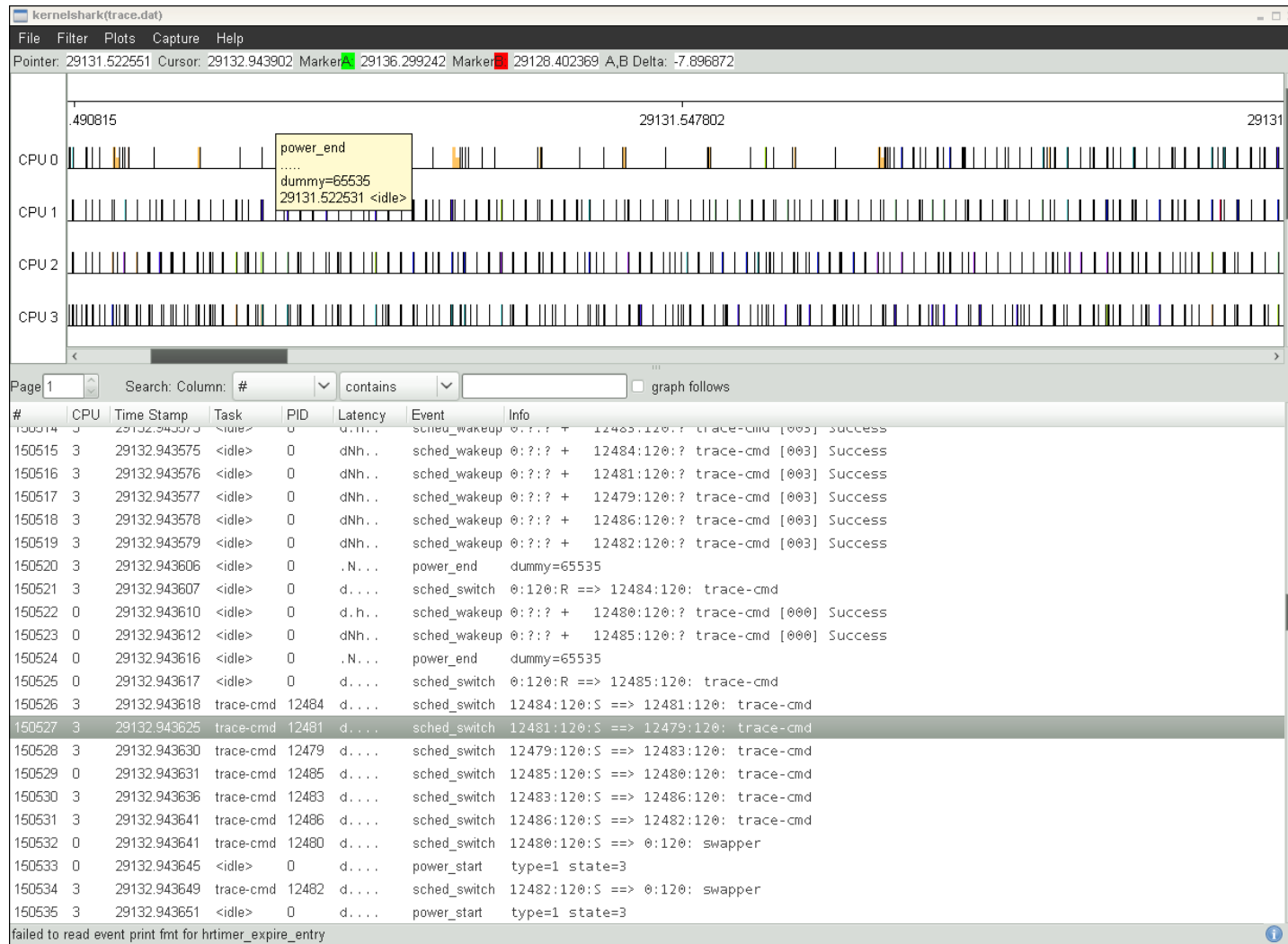
ftrace

- Can dump on events / oops / custom triggers
- But still too much overhead in many cases to run always during debug

Intel PT

- Upcoming Intel CPU feature
- Traces all branches with low overhead
- Will be supported in perf and gdb and with “FlightRecorder”

Biggest challenge is better tools to understand traces (too much data)



Understanding source code

- Often biggest problem finding code
- grep/cscope work great for many cases
- But do not understand indirect pointers (OO in C model used in kernel): Give me all “do_foo” instances

```
struct foo_ops {  
    int (*do_foo)(struct foo *obj);  
} = { .do_foo = my_foo };  
foo->do_foo(foo)
```

- Would be great to have a cscope like tool that understands this based on types/initializers

Conclusion

- Linux has a lot of great tools for making kernel development easier
- We need them to control complexity
- But still many improvements possible
- Questions?