# Towards a BPF Memory Model

Paul E. McKenney
Software Engineer

2021 Linux Plumbers Conference: Networking & BPF Summit

FACEBOOK

# Agenda

1. What is a memory model?
2. Why is a BPF memory model necessary?
3. Which memory model for BPF?
4. Overhead of Atomic Operations
5. Should all kernel atomics be added to BPF?
6. How would a BPF memory model work?

# What is a Memory Model?

A memory model defines outcomes of concurrent accesses.

What is a “data race”?  Certain types of concurrent accesses:

- C standard: At least one write and at least one unmarked access
- Linux kernel: It is complicated!
  - The kernel relies on compiler implementations, not the C standard

Linux-kernel memory model:
https://lwn.net/Articles/718628/
https://lwn.net/Articles/720550/

Compiler optimizations:
https://lwn.net/Articles/793253/
https://lwn.net/Articles/799218/

# The Compiler Might Not Always Be Your Friend

Many compiler optimizations assume sequential code:

Load tearing*, store tearing*, load fusing*, code reordering*, invented loads*, invented stores*, store-to-load transformations, dead-code elimination*

* Seen in the wild

All of this is in addition to what the hardware can do to your concurrent code!

Linux-kernel memory model:
https://lwn.net/Articles/718628/
https://lwn.net/Articles/720550/

Compiler optimizations:
https://lwn.net/Articles/793253/
https://lwn.net/Articles/799218/

# Why is a BPF Memory Model Necessary?

Increasing numbers of BPF programs feature concurrency, not only with each other and with the rest of the kernel, but also via shared memory with userspace BPF-program components.

In addition, ordering is an issue even on x86.

Example BPF kernel control: Networking TCP congestion control

https://nakryiko.com/posts/libbpf-bootstrap/
https://lwn.net/Articles/811631/

# Which Memory Model for BPF?

# Which Memory Model for BPF?

## The Linux Kernel Memory Model

# Overhead of Atomic Operations

| Linux-kernel operation | x86 | powerpc |
|---|---|---|
| `READ_ONCE() & WRITE_ONCE()` | Volatile load/store | Volatile load/store |
| `smp_load_acquire()` | Load then `barrier()` | Load then `lwsync` |
| `smp_store_release()` | `barrier()` then store | `lwsync` then store |
| `smp_rmb()` and `smp_wmb()` | `barrier()` | `lwsync` |
| `smp_mb()` | `lock;addl` to stack | `sync` |
| Atomic RMW operations | `lock;` RMW instruction | `sync;larx-stcx;sync` |

Note: barrier() emits no instructions

# Overhead of Atomic Operations

| Linux-kernel operation | x86 | powerpc |
|---|---|---|
| `READ_ONCE() & WRITE_ONCE()` | Volatile load/store | Volatile load/store |
| `smp_load_acquire()` | Load then `barrier()` | Load then `lwsync` |
| `smp_store_release()` | `barrier()` then store | `lwsync` then store |
| `smp_rmb()` and `smp_wmb()` | `barrier()` | `lwsync` |
| `smp_mb()` | `lock;addl` to stack | `sync` |
| Atomic RMW operations | `lock;` RMW instruction | `sync;larx-stcx;sync` |

Note: barrier() emits no instructions, and atomics often incur cache misses

# Should All Kernel Atomics Be Added to BPF?

# Should All Kernel Atomics Be Added to BPF?

READ_ONCE(), WRITE_ONCE(), smp_store_release(), smp_load_acquire(), rcu_assign_pointer(), rcu_dereference(), smp_store_mb(), smp_mb(), smp_rmb(), smp_wmb(), smp_mb__before_atomic(), smp_mb__after_atomic(), smp_mb__after_spinlock(), smp_mb__after_unlock_lock(), barrier(), xchg(), xchg_relaxed(), xchg_release(), xchg_acquire(), cmpxchg(), cmpxchg_relaxed(), cmpxchg_acquire(), cmpxchg_release(), atomic_read(), atomic_set(), atomic_read_acquire(), atomic_set_release(), atomic_add(), atomic_sub(), atomic_inc(), atomic_dec(), atomic_add_return(), atomic_add_return_relaxed(), atomic_add_return_acquire(), atomic_add_return_release(), atomic_fetch_add(), atomic_fetch_add_relaxed(), atomic_fetch_add_acquire(), atomic_add_return_release(), atomic_fetch_add(), atomic_fetch_add_relaxed(), atomic_fetch_add_acquire(), atomic_fetch_add_release(), atomic_inc_return(), atomic_inc_return_relaxed(), ...

{,raw_}spin_lock{,_bh,_irq,_irqsave}(), {,raw_}spin_unlock{,_bh,_irq,_irqrestore}(), {,raw_}spin_trylock{,_bh,_irq,_irqsave}(), spin_is_locked(), mutex_lock(), mutex_unlock(), mutex_trylock(), mutex_is_locked(), mutex_is_locked(), test_and_set_bit_lock(), ...

# Should All Kernel Atomics Be Added to BPF?

It is true that increasing numbers of BPF programs feature concurrency, both with each other and with the rest of the kernel, however ...

# Should All Kernel Atomics Be Added to BPF?

It is true that increasing numbers of BPF programs feature concurrency, both with each other and with the rest of the kernel, however ...

... much of this concurrency is handled by BPF helpers, reducing (but by no means eliminating) the need for concurrency in BPF programs.

# Should All Kernel Atomics Be Added to BPF?

It is true that increasing numbers of BPF programs feature concurrency, both with each other and with the rest of the kernel, however …

… much concurrency is in BPF helpers.

Therefore, add the most popular/useful first, for example:

- atomic_inc()*, cmpxchg()*, fetch_and_add()*, xchg()*, atomic_add()*, atomic_sub()*, atomic_and()*, atomic_or()*, and atomic_xor()*
- smp_load_acquire() & smp_store_release, not so much smp_rmb() & smp_wmb()

* Already in mainline.

# Which Memory Model for BPF?

***An Appropriate Subset of***
the Linux Kernel Memory Model

# How Would a BPF Memory Model Work?

"Use the right tool for the job!"

Exhaustive/exact analysis of small programs: Herd7

- Requires hand translation

Dynamic/approximate (but quite good) analysis of full kernel: KCSAN

- Requires integration with BPF (which looks doable)
- Handling of userspace components is still an open question

https://lwn.net/Articles/816850/
https://lwn.net/Articles/816854/
https://docs.google.com/document/d/1r4-ggu8RW2nzvFeT8AQxaCn6c-XkuXljzf7m0j5PypM/edit?usp=sharing

# How Would a BPF Memory Model Work With Herd7?

Hand translate usermode and BPF programs to Linux-kernel litmus tests

- Normal load/store remains as-is, but litmus test uses pointers
- volatile-casted load to READ_ONCE()
- volatile-casted store to WRITE_ONCE()
- __sync_fetch_and_add() to atomic_fetch_add()
- __sync_fetch_and_sub() to atomic_fetch_sub()
- __sync_val_compare_and_swap() to cmpxchg()
- __sync_lock_test_and_set() to xchg()
- BPF map access as that access while holding lock (e.g., bpftrace)

# Create Litmus Test From Simplified atomics.c Tests

```
/* In-kernel BPF program.  tools/testing/selftests/bpf/progs/atomics.c */
__u64 xchg64_value = 1;
__u64 xchg64_result = 0;

SEC("fentry/bpf_fentry_test1")
int BPF_PROG(xchg, int a)
{
        __u64 val64 = 2;

        xchg64_result = __sync_lock_test_and_set(&xchg64_value, val64);
}

/* User-mode BPF program.  tools/testing/selftests/bpf/prog_tests/atomics.c */
static void test_xchg(struct atomics *skel)
{
        err = bpf_prog_test_run(prog_fd, 1, NULL, 0, NULL, NULL, &retval, &duration);
        ASSERT_EQ(skel->data->xchg64_value, 2, "xchg64_value");
}
```

# Create Litmus Test From Simplified atomics.c Tests

```
C bpf-xchg

{}

P0(int *xchg64_result, int *xchg64_value) // test_xchg()
{
	int r0;
	int r1;

	r1 = smp_load_acquire(xchg64_result);
	if (r1) { // BPF program complete?
		r2 = *xchg64_value;
	}
}

P1(int *xchg64_result, int *xchg64_value) // BPF_PROG(xchg)
{

	r1 = atomic_xchg(xchg64_value, 2);
	smp_store_release(xchg64_result, 1); // Emulate BPF program completion
}

locations [xchg64_result; xchg64_value]
exists ((0:r1=1 /\ ~0:r2=2) \/ ~1:r1=0) (* Bad outcome. *)
```

# Run Litmus Test Using herd7 and LKMM

```
$ cd tools/memory-model
$ herd7 -conf linux-kernel.cfg /tmp/bpf-xchg.litmus
Test bpf-xchg Allowed
States 2
0:r1=0; 0:r2=0; 1:r1=0; xchg64_result=1; xchg64_value=2;
0:r1=1; 0:r2=2; 1:r1=0; xchg64_result=1; xchg64_value=2;
No
Witnesses
Positive: 0 Negative: 2
Condition exists (0:r1=1 /\ not (0:r2=2) \/ not (1:r1=0))
Observation bpf-xchg Never 0 2
Time bpf-xchg 0.00
Hash=d0b286381f9e93048632ae9c9d25e363
$ # Bad condition never happens
```

# How to Adapt KCSAN to BPF Programs? (1/2)

Kernel Concurrency Sanitizer (KCSAN) provides SW watchpoints, which are used to detect data races and to enforce concurrency design rules.

In-kernel C code such as BPF helpers are already handled by KCSAN.
BPF JIT code can use KCSAN public API:

- __tsan_{read,write}{1,2,4,8}() preferred.
- __kcsan_check_access() also works, but not as good performance.
- This should also cover bpftrace
- Userspace code TBD

https://lwn.net/Articles/816850/
https://lwn.net/Articles/816854/
https://docs.google.com/document/d/1r4-ggu8RW2nzvFeT8AQxaCn6c-XkuXljzf7m0j5PypM/edit?usp=sharing

# How to Adapt KCSAN to BPF Programs? (2/2)

Kernel Concurrency Sanitizer (KCSAN) permits considerable control of the types of races reported (showing defaults):

- CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC=y
- CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY=y
- CONFIG_KCSAN_INTERRUPT_WATCHER=n
- CONFIG_KCSAN_STRICT=n (Used by RCU to override the above.)
- CONFIG_DEBUG_INFO=y # Translate stack addresses

https://lwn.net/Articles/816850/
https://lwn.net/Articles/816854/
https://docs.google.com/document/d/1r4-ggu8RW2nzvFeT8AQxaCn6c-XkuXljzf7m0j5PypM/edit?usp=sharing

# What Does KCSAN Tell You? (1/2)

```
==================================================================
BUG: KCSAN: data-race in tick_nohz_idle_stop_tick / tick_nohz_idle_stop_tick

write to 0xffffffffabc1c940 of 4 bytes by task 0 on cpu 7:
 tick_nohz_idle_stop_tick+0x146/0x3c0
 do_idle+0x103/0x290
 cpu_startup_entry+0x15/0x20
 secondary_startup_64_no_verify+0xc3/0xcb

no locks held by swapper/7/0.
irq event stamp: 1390256
hardirqs last  enabled at (1390255): [<ffffffffa99d0b50>] tick_nohz_idle_enter+0x110/0x140
hardirqs last disabled at (1390256): [<ffffffffa990879c>] do_idle+0x9c/0x290
softirqs last  enabled at (1390246): [<ffffffffa98b5574>] __irq_exit_rcu+0x64/0xc0
softirqs last disabled at (1390223): [<ffffffffa98b5574>] __irq_exit_rcu+0x64/0xc0
```

# What Does KCSAN Tell You? (2/2)

```
read to 0xffffffffabc1c940 of 4 bytes by task 0 on cpu 10:
 tick_nohz_idle_stop_tick+0x12c/0x3c0
 do_idle+0x103/0x290
 cpu_startup_entry+0x15/0x20
 secondary_startup_64_no_verify+0xc3/0xcb

no locks held by swapper/10/0.
irq event stamp: 3677807
hardirqs last  enabled at (3677806): [<ffffffffa99d0b50>] tick_nohz_idle_enter+0x110/0x140
hardirqs last disabled at (3677807): [<ffffffffa990879c>] do_idle+0x9c/0x290
softirqs last  enabled at (3677797): [<ffffffffa98b5574>] __irq_exit_rcu+0x64/0xc0
softirqs last disabled at (3677788): [<ffffffffa98b5574>] __irq_exit_rcu+0x64/0xc0

Reported by Kernel Concurrency Sanitizer on:
CPU: 10 PID: 0 Comm: swapper/10 Not tainted 5.14.0-next-20210902+ #2916
Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS 1.13.0-1ubuntu1.1 04/01/2014
==================================================================
```

# How to Adapt BPF Programs to KCSAN?

Kernel Concurrency Sanitizer (KCSAN) provides SW watchpoints, which are used to detect data races and to enforce concurrency design rules.

- ASSERT_EXCLUSIVE_ACCESS() complains if racing access.
- ASSERT_EXCLUSIVE_WRITER() complains if racing write.
- Use these to detect violations of your concurrency design.

https://lwn.net/Articles/816850/
https://lwn.net/Articles/816854/
https://docs.google.com/document/d/1r4-ggu8RW2nzvFeT8AQxaCn6c-XkuXljzf7m0j5PypM/edit?usp=sharing

# LKMM/herd7 or KCSAN?

It depends…

- LKMM does moral equivalent of full state-space search, while KCSAN only detects problems that actually occur in testing.
- LKMM requires hand translating to tiny restricted litmus tests, while KCSAN can operate across the entire kernel.  For example, LKMM does not handle unbounded loops, function calls, interrupts, and so on, though many of these can be emulated.
- LKMM can check for complex conditions in the “exists” clause, while KCSAN gets a similar effect using ASSERT_EXCLUSIVE_ACCESS() and ASSERT_EXCLUSIVE_WRITER(), along with WARN_ON_ONCE() &c.

https://lwn.net/Articles/816850/
https://lwn.net/Articles/816854/
https://docs.google.com/document/d/1r4-ggu8RW2nzvFeT8AQxaCn6c-XkuXljzf7m0j5PypM/edit?usp=sharing

# Questions & Discussion

# Thank you!