eBPF Shenanigans with Flux

Crazy kernel schedulers implemented in BPF

Barret Rhoden brho@google.com
Linux Plumbers Conference (LPC ‘23)
https://lpc.events/event/17/contributions/1601/

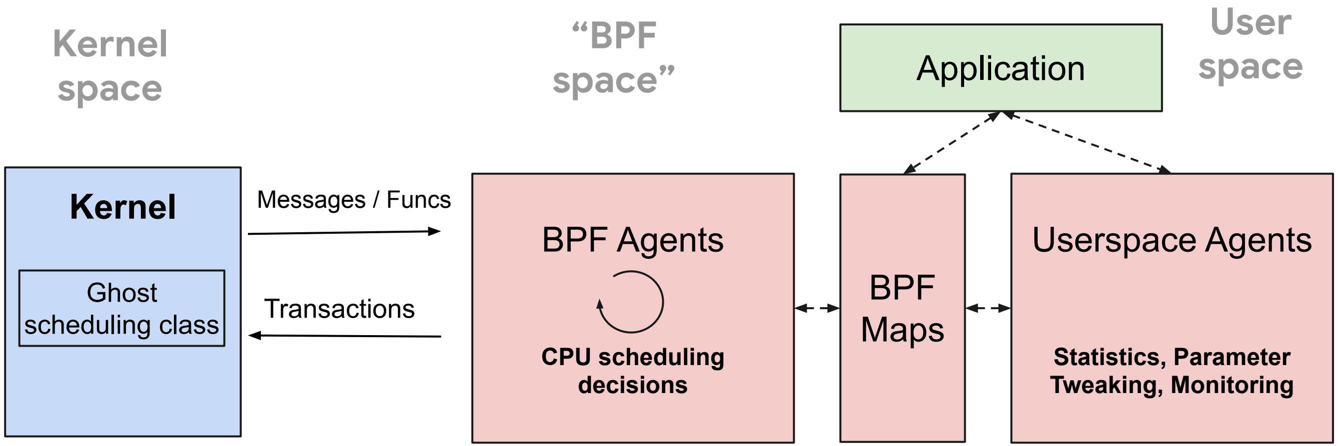
Agenda

- Brief intro to Flux
  - Framework for designing schedulers
  - Written in BPF for Ghost (LPC '22)
- Building data structures from Array Maps
- Simulating object-oriented programming without function pointers
- Future plans and open sourcing

Flux in 5 minutes

Ghost-BPF Scheduling

- All scheduling decisions are made in BPF
- Userspace has a role, but it is not in the critical path



Problem

Design a scheduler, given:

- A large, multicore machine, possibly with a fun cache topology
- For applications with different classes of threads or workloads
  - e.g. A set of threads handling RPCs and a set doing Housekeeping
- And your available set of cpus may change at runtime
  - Yielding to CFS kworkers
  - You’re a paravirtualized guest
  - Shared tenancy machine

Decomposing the scheduler

- We have multiple cpus: make them a central component of the scheduler
- What if I dedicated certain cpus to certain **classes** of threads in an app?
  - Partition the cpus, such that threads of the same type run in the same partition
- We can write *subschedulers* for each thread type
  - RPC threads get EDF, Housekeeping gets FIFO, etc.
  - Don't need to develop one magic policy that works for all thread types
- Overall partitioning policy, e.g. “Housekeeping gets 5 cpus, RPC gets the rest”
- Wait… where does that partitioning policy come from?

CPU Partitioning is Scheduling

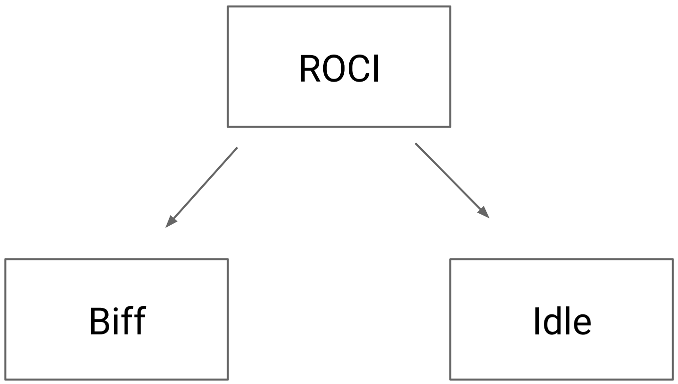
- The allocation of cpus to *subschedulers* is itself a scheduling decision
  - We need schedulers of schedulers!

- The interface between coordinating schedulers is cpus
  - When schedulers talk to each other: make requests, make allocations, etc., they talk about cpus
  - This is a universal concept in scheduling: applies to both M:N scheduling and paravirt scheduling

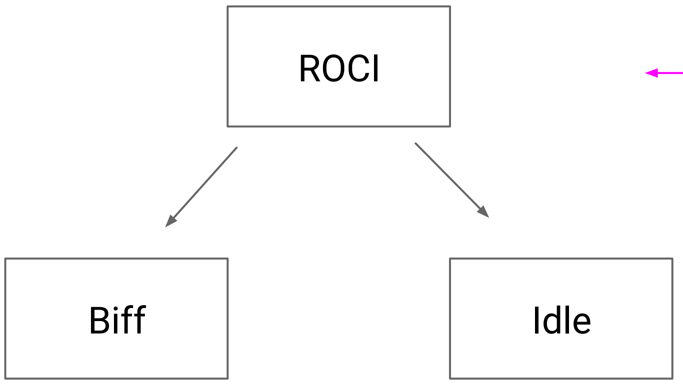
Flux:

- Compose an overall scheduler from a hierarchy of smaller *subschedulers*
- A thread belongs to a single subscheduler at a time.
- Cpus are allocated to subschedulers.
- Subschedulers:
  - Are just blobs of code and data
  - Exist in a parent-child relationship
  - Schedule either a thread or another subscheduler

Hello world Flux scheduler: Global FIFO policy



Hello world Flux scheduler: Global FIFO policy



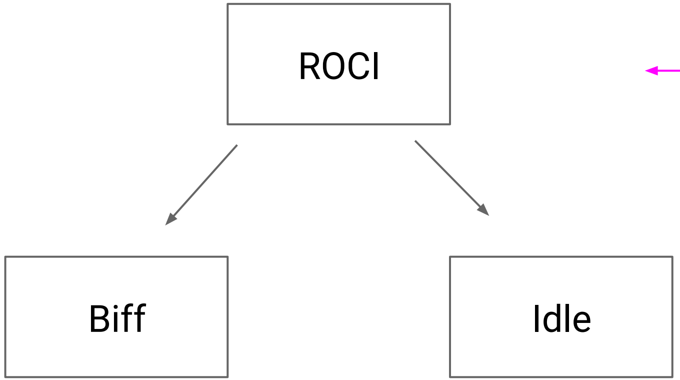
ROCI:

a cpu scheduler (Root One Child and Idle)

Policy:

Give Biff whatever it wants, give Idle the rest

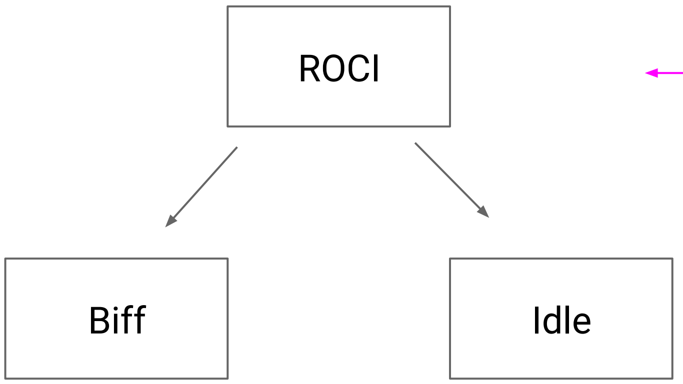
Hello world Flux scheduler: Global FIFO policy



ROCI:
a cpu scheduler (Root One Child and Idle)
Policy:
Give Biff whatever it wants, give Idle the rest

Biff:
a thread scheduler
Policy:
Global FIFO

Hello world Flux scheduler: Global FIFO policy

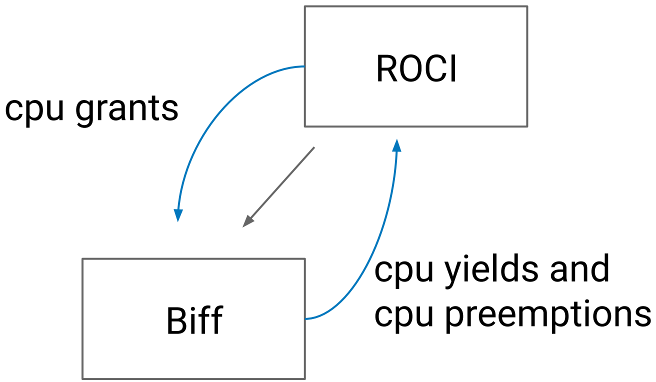


ROCI:
a cpu scheduler (Root One Child and Idle)
Policy:
Give Biff whatever it wants, give Idle the rest

Biff:
a thread scheduler
Policy:
Global FIFO

Idle:
some sort of scheduler
Policy:
halt the cpu

Hello world Flux scheduler: Global FIFO policy



CPU Lifecycle

1: Biff calls `flux_request_for_cpus(nr_cpus)`

2: ROCI callback:

`roci_request_for_cpus(biff, nr_cpus)`

3: ROCI picks a cpu for Biff, possibly sends an IPI

On that cpu:

4: ROCI calls `flux_cpu_grant(biff)`

5: Biff picks a task, calls `flux_run_thread()`

6: Or Biff calls `flux_cpu_yield()`

Okay... How are we doing this in BPF?

- Data structures of different types
  - Different types of threads
  - Different types of subschedulers
  - Cpus are important too - need structs for those
- That exist in some hierarchy
  - Pointers?
  - And we're making decisions. Linked lists? RB Trees?
  - Lists of threads, lists of cpus
- And I saw callbacks in there...

Data structures and whatnot

Memory management with ARRAY_MAPs

- Just about every allocation we make is from an ARRAY_MAP
  - Subschedulers, threads, per-cpu data, etc.
- These are mmapable (at least those without spinlocks)
  - Userspace **agent** can adjust policy bits with atomics
  - Userspace **application** can tell us thread-specific info, e.g. an RPC deadline
- Pointers are replaced with **dense** integers and an (implicit) array
  - `struct flux_sched *roci` is known as “sched_id 1”
  - `struct flux_thread *foo` is known as “thread_id 42”
- Thread IDs are discoverable via another map (e.g. pid_t -> dense index)
- And we can build our own data structures

Linked Lists: BSD-style “sys/queue.h” list / tailq

```
struct arr_list {
    unsigned int first;
    unsigned int last;
};

struct arr_list_entry {
    unsigned int next;
    unsigned int prev;
};

struct some_element {
    struct arr_list_entry link;
    int foo;
};
```

Pointers are replaced with integers

Embed the link, like usual

Basic Structures: BSD-style “sys/queue.h” list / tailq

- The usual operations:
  - First, next, prev
  - Insert head, Insert tail
  - Remove
  - `arr_list_insert_tail(arr, arr_sz, head, elem, field)`
- for_each iteration
  - It’s BPF, so we can’t loop forever
  - “for each up to N times” (for debugging)
  - `arr_list_foreach(var, arr, arr_sz, head, field, _i, max)`

Pass the array and the array size…

Why pass the array size?

- `arr_list_insert_tail(arr, arr_sz, head, elem, field)`
- Gotta convince the verifier any time we convert from index to pointer
- Treat idx == 0 as “no element” and idx == 1 is the 0th element of the array
- `bpf_array_elem_sz(arr, arr_sz, id - 1);`
  - That’s some inline asm to force the bounds check on arr_sz
  - Essentially `&arr[idx]`

Subtle point about locking and arrays

- Picture the ARRAY_MAP of struct flux_thread
  - Is it N elements of type struct flux_thread?
  - That would mean each lookup is a `bpf_map_lookup_elem()` call
  - Which you can't do while holding a bpf spinlock!
- Instead, it's an ARRAY_MAP of one item, which is an array of N threads
- Same memory layout, but lets you do one Map Lookup for all threads
  - Get the array outside the lock, etc.
  - Similarly, could put the array in BSS
- This trick doesn't work for our `struct flux_sched` arrays
  - Each sched has a **spinlock**, and you can't put spinlocks in interior structs
  - Can't put spinlocks in BSS either (or at least I couldn't…)

AVL Trees! (Self-balancing, binary trees)

- AVL are denser than RB and easier to implement
- Replace `while` loops with `for (i =0; i < MAX_AVL_HEIGHT; i++)`
- That means we might not be able to stuff all nodes into the tree
- Solution: *overflow* linked list
  - e.g. “Get Min” might not always be the **real** minimum
  - Check the front of the overflow list for any Get Min or Get Max

Half-baked Object-Oriented programming with Unions

- We’ve got threads and subschedulers of different types
- But a BPF Map can only have a single type.
- Two classic styles of hooking specific objects to generic ones:
  - Have a `void *private` blob in the generic struct. e.g. VFS
    - Don’t want to use more pointers
  - Embed the generic object in the specific object. e.g. container_of() stuff.
    - Need the objects to all be the same size
- Add a union to the overall object
  - Each possible thread type gets a union member
  - e.g. One size for every thread struct, regardless of type

Example Thread Struct

```
struct flux_thread {
    struct __flux_thread f;
    union {
        struct biff_flux_thread biff;
        struct doc_flux_thread doc;
    };
};
```

the generic part, including f.type

the specific part, based on the thread's type

Different memory management than the kptrs style

Kptrs managed memory style:

- bpf_obj_new(), bpf_obj_drop(), bpf_list_head, bpf_rbtree_add, bpf_rb_node, etc.
- The verifier knows what you're doing

versus

Blob of RAM, build what you want!

- The overall ARRAY_MAP is a blob of memory, up to us to allocate **within** it
- The verifier just need to make sure you stay inside the blob

Pros and Cons: Kptrs style

- Dynamic allocation
- Kernel can enforce invariants on your structures (e.g. safely traverse a tree)
- Verifier needs to know about your types
- Need to associate your spinlocks with your data structures
- Ownership model for memory. Can an object belong to multiple lists/trees yet?
- Need the helpers / kfuncs built into the kernel.
  - Want a new structure? Need a new kernel.
  - Want a new operation on an existing structure? Need a new kernel.
- Can’t touch the managed memory.
  - e.g. atomic_or a bit in a bpf_cpumask from userspace or whatever

Pros and Cons: Blob of RAM

- mmappable by userspace
- No guardrails. The verifier protects the kernel, not your code.
- Hard to convince the verifier your code terminates
    - e.g. `avl_tree_insert()` is very branchy
    - Had to limit the size of the AVL tree and have that overflow list
- Giant blob of RAM? That's wasted kernel memory.
    - TBD - we think we can fault in the ARRAY_MAP on demand, instead of populating it.

Function pointers?

There are no function pointers

- How do we get from `flux_request_for_cpus()` to `roci_request_for_cpus()`?
- You'd expect something like "roci->**ops**.request_for_cpus(nr_cpus)"
- We can't follow function pointers
- But every subscheduler and thread has an integer type
- Flux library code uses macros that generate switch statements, e.g.

```
#define __pick_next_task(sched, cpu, ctx) ({
        switch ((sched)->f.type) {
        __gen_cpu_op_cases(__cat_op, _pick_next_task, sched, cpu, ctx)
        };
})
```

Your agent must define this

Compose your agent.bpf.c from subschedulers

```
#define __gen_cpu_op_cases(op_type, op, sched, ...)
        case SCHED_TYPE_HOUSEKEEPING:
                op_type(biff, op)(sched, __VA_ARGS__);
                break;
        case SCHED_TYPE_RPC:
                op_type(doc, op)(sched, __VA_ARGS__);
                break;
        case SCHED_TYPE_IDLE:
                op_type(idle, op)(sched, __VA_ARGS__);
                break;
...


#include "third_party/ghost/bpf/bpf/biff_flux.bpf.c"
#include "third_party/ghost/bpf/bpf/doc_flux.bpf.c"
#include "third_party/ghost/bpf/bpf/idle_flux.bpf.c"
```

Similar to function pointers, tell Flux what code to use for which scheduler

Literally composing your agent from subscheduler C code

Future plans
and
Open sourcing

Code Stuff

- https://github.com/google/ghost-userspace
  - flux_header_bpf.h, flux_api.bpf.c, flux_dispatch.bpf.c
  - queue.bpf.h, avl.bpf.h
- Flux is built on top of Ghost.
- The linked list and AVL tree and whatnot can be used independently of Flux
- The model of “build your structures from a blob of memory” can be used in any BPF program

Speaking of open source

Although not related directly to Flux or BPF shenanigans:

- “Google is committed to upstreaming our changes”
- “Google's prodkernel cadence follows the LTS stable kernel and is on track to pickup the 6.x LTS kernel”

Ghost and Sched Ext (SCX)

- Overall vision: build Ghost on top of SCX
- Port Flux to use SCX's interfaces
- Open question of whether to stick with the "blob of memory" or use kptrs
  - The memory management of threads, cpus, etc. is all handled by the Flux code
- Ideally any scheduler written against Flux-on-Ghost would work on Flux-on-SCX

Thanks!

- Flux: a framework for building schedulers from a hierarchy of subschedulers
  - It’s crazy, and there’s a lot more to cover. Maybe some other time.
- You can build anything out of a blob of memory, even in BPF
  - Pointers -> Integers + ARRAY_MAPs
- You can even do object oriented programming in BPF
  - With some macros and some patience…