eBPF Kernel Scheduling with Ghost

Barret Rhoden
brho@google.com

Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

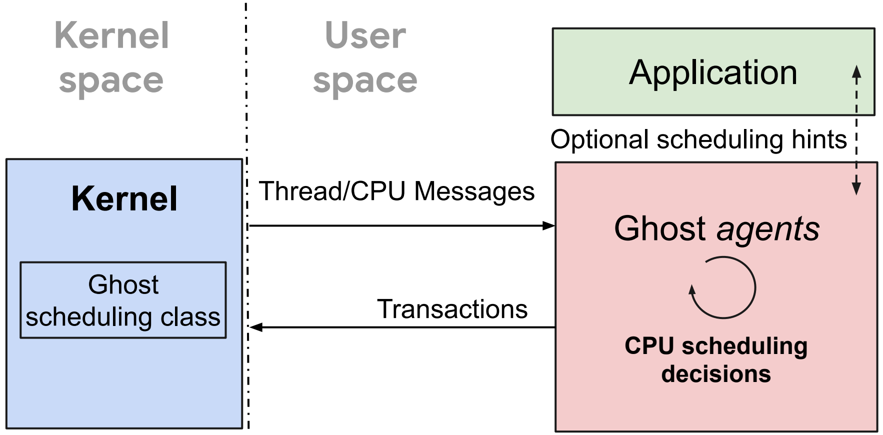
Agenda

- Ghost Primer
- How BPF works in Ghost
- “BPF-Only Scheduling”
- Biff: the world’s dumbest BPF-only scheduler
- Future work
- Discussion
- FAQ

Ghost Primer

What is Ghost?

- Kernel scheduler class, below CFS in priority
- Scheduling decisions made in userspace by an *agent* process
- Kernel sends *messages* to the agent: “task X blocked on cpu 6”
- Agent issues *transactions* to the kernel: “run task X on cpu 12”



Do No Harm

- Using Ghost should not hurt the OS: agent fault isolation
- Even **during operation**, ghost cannot hurt the rest of the system
  - Below CFS in priority: CFS preempts Ghost tasks
  - Including kernel threads: don’t want to stop those!
- If the agent *fails*, all tasks get moved back to CFS
- Failure is configurable, and also triggerable by userspace:
  - Kernel notices a runnable task doesn’t get on cpu for X msec
  - Userspace daemon (borglet, kubelet) notices errors or poor performance
  - Application notices errors or poor performance

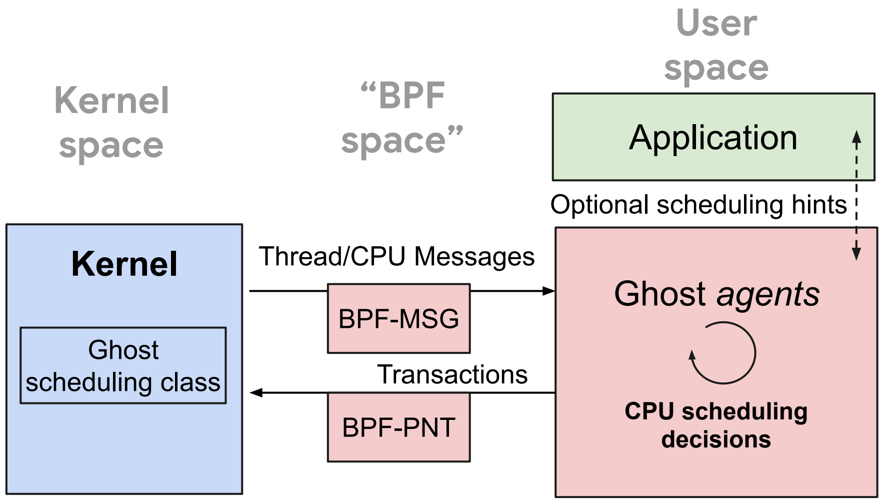
Multiple Agents per Machine

- Ghost sched class supports distinct, independent agents
- Enclave: a set of CPUs scheduled by a single Ghost agent
- Semi-hard partition: you can move CPUs between enclaves, but it requires the agent to yield the CPU
- Agent live-update mechanism to hand off control of an enclave
    - O(msec)
    - Have the new agent ready to go, kill the old one, etc.

How BPF works in Ghost

BPF in Ghost

- Agent process attaches a BPF program: BPF is an extension of the agent
- Messages -> BPF_GHOST_MSG_SEND
- Transactions -> BPF_GHOST_SCHED_PNT (pick_next_task)



Ghost BPF Program Types: called from the kernel

- BPF-MSG: BPF_PROG_TYPE_GHOST_MSG
  - Context is *struct bpf_ghost_msg*
  - Attached at produce_for_task(struct task_struct *p, struct bpf_ghost_msg *msg)
  - e.g. MSG_TASK_WAKEUP: “task 6 woke on cpu 15”

- BPF-PNT: BPF_PROG_TYPE_GHOST_SCHED
  - Context is *struct bpf_ghost_sched*
  - Attached in pick_next_task_ghost()
  - Essentially picks the next task to run on this cpu, via a helper

Ghost Messages: the functional API for BPF-MSG

Task Messages:

- MSG_TASK_NEW
- MSG_TASK_BLOCKED
- MSG_TASK_WAKEUP
- MSG_TASK_PREEMPT
- MSG_TASK_YIELD
- MSG_TASK_DEPARTED
- MSG_TASK_DEAD
- MSG_TASK_SWITCHTO
- MSG_TASK_AFFINITY_CHANGED
- MSG_TASK_LATCHED

CPU Messages:

- MSG_CPU_TICK
- MSG_CPU_TIMER_EXPIRED
- MSG_CPU_NOT_IDLE
- MSG_CPU_AVAILABLE
- MSG_CPU_BUSY
- MSG_CPU_AGENT_BLOCKED
- MSG_CPU_AGENT_WAKEUP

(so far…)

Ghost BPF Helpers: interface to the kernel

- bpf_ghost_wake_agent(cpu)
    - kick the userspace agent on a cpu
- bpf_ghost_run_gtid(task, …)
    - set task to run next on this cpu
    - called from BPF-PNT only
- bpf_ghost_resched_cpu(cpu)
    - force cpu to reschedule (sets need_resched)

BPF Programs are **part of the Agent**

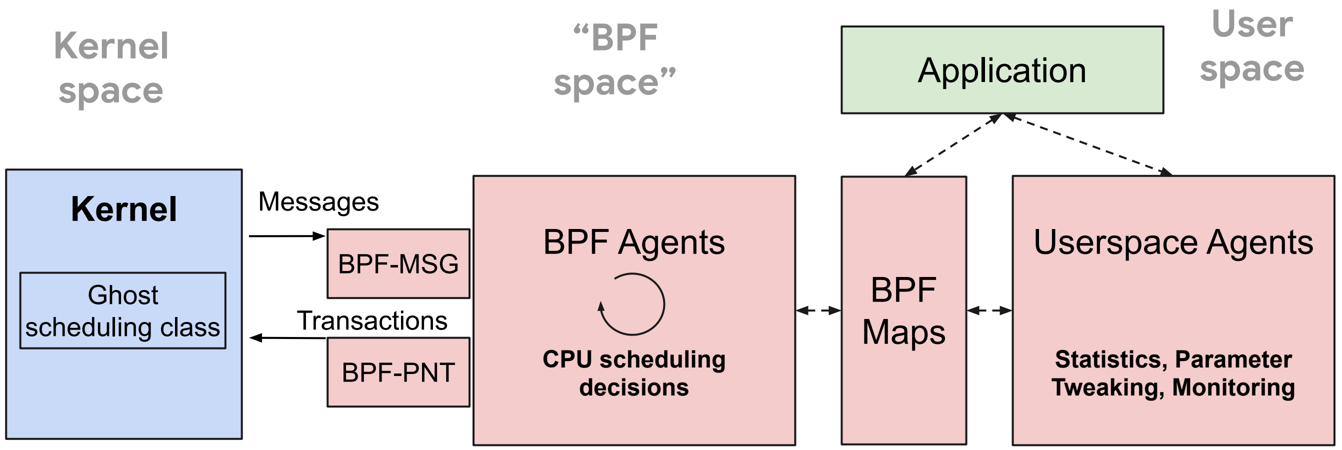
- Act as an agent 'thread', with similar privileges as userspace
- Closely coupled to the userspace agent 与用户空间代理紧密耦合
    - Embedded in the agent binary, libbpf-style 嵌入在代理二进制文件中，具有libbpf风格
    - Have the same lifetime as the agent 与agent具有相同生命周期
- Share memory with the userspace agent
    - e.g. BPF_MAP_TYPE_ARRAY: mmapped by userspace
- "BPF Space" or "Ring-B": analogous to x86 Ring-3:
    - Array maps are *windows* into the agent's address space
    - bpf helpers are the *entry points* to the kernel, like syscalls
    - BPF_PROG_RUN attach points are the *interrupt descriptor table* vectors.

这个将bpf虚拟机和真os做对比的这一段写得还挺好的

BPF-Only Scheduling

“BPF-only” Scheduling

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



Why Schedule in BPF instead of Userspace?

- Alternative: context switch to that cpu’s agent task and let it handle messages and pick_next_task.
- Three reasons BPF is better:
  - No context switches! (Depends on your app if this matters)
  - Don’t have to preempt a running task to run that cpu’s agent.
    - e.g. Task 6 wakes up. Don’t have to preempt another task to tell the agent about it.
  - BPF is synchronous! Solves a lot of heartache.
    - Hold the rq lock during bpf-msg, but not in bpf-pnt
    - In schedule()->pick_next_task() for bpf-pnt
- Downsides
  - Harder programming environment: limited loops, etc. 确实，这一点在论文中有提及
  - Event driven: harder to “spawn a background thread”
  - Data structures are limited to BPF Map types

Biff: a simple BPF-only scheduler

Biff Scheduler: world's simplest BPF agent

- Global FIFO scheduling policy! global_rq: BPF_MAP_TYPE_QUEUE

```
int biff_pnt(struct bpf_ghost_sched *ctx)
{
       bpf_map_pop_elem(&global_rq, next);
       bpf_ghost_run_gtid(next, ...);
}

int biff_msg_send(struct bpf_ghost_msg *msg)
{
       switch (msg->type) {
       case MSG_TASK_WAKEUP:
       case MSG_TASK_PREEMPT:
       case MSG_TASK_YIELD:
                bpf_map_push_elem(&global_rq, msg->gtid, 0);
                break;
      }
}
```

Biff

- The ‘real’ Biff scheduler is a little more complicated
- Error handling, accounting helpers, etc.
- Any non-trivial scheduler will need to track **per-cpu** and **per-thread** data
- Biff is a policy-less tutorial for how you can track data and share it with userspace or an application

Biff Maps

您甚至可以通过 unix socket将此 FD 传递给应用程序，让他们告诉我们每个workload的提示

You can even pass this FD over a unix socket to the application to let them tell us per-workload hints!

- cpu_data: per-cpu data
  - struct biff_bpf_cpu_data { current_task; etc; }
  - BPF_MAP_TYPE_ARRAY, **mmappable by userspace**
  - indexed by cpu id
- sw_data: per-task data
  - struct biff_bpf_sw_data { runnable_at; last_ran_at; etc; }
  - BPF_MAP_TYPE_ARRAY, **mmappable by userspace**
  - indexed by a task's status_word_index (densely allocated integer per task)
- sw_lookup:
  - BPF_MAP_TYPE_HASH
  - From task id (gtid) to status_word_index

Biff Helper Examples

```
static void task_stopped(int cpu)
{
        struct biff_bpf_cpu_data *pcpu;

        pcpu = bpf_map_lookup_elem(&cpu_data, &cpu);
        if (!pcpu)
                return;
        pcpu->current = 0;
}

/* Forces the cpu to reschedule and eventually call bpf-pnt. */
static int resched_cpu(int cpu)
{
        struct biff_bpf_cpu_data *pcpu;

        pcpu = bpf_map_lookup_elem(&cpu_data, &cpu);
        if (!pcpu)
                return -1;
        return bpf_ghost_resched_cpu(cpu, pcpu->cpu_seqnum);
}
```

Biff Actual Message Handler

```
static void __attribute__((noinline)) handle_wakeup(struct bpf_ghost_msg *msg)
{
        struct ghost_msg_payload_task_wakeup *wakeup = &msg->wakeup;
        struct biff_bpf_sw_data *swd;
        u64 gtid = wakeup->gtid;
        u64 now = bpf_ktime_get_us();

        swd = gtid_to_swd(gtid);
        if (!swd)
                return;
        swd->runnable_at = now;

        enqueue_task(gtid, msg->seqnum);
}
```

noinline and casting games…

Get per-thread struct, do
your accounting

Enqueue: whatever policy you want.
Biff just sticks it in the global FIFO map

Gotcha! Why is handle_wakeup() noinline?

- “dereference of modified ctx ptr R6 off=3 disallowed”
- The context is:

```
struct bpf_ghost_msg {
        union {
                struct ghost_msg_payload_task_dead      dead;
                struct ghost_msg_payload_task_blocked   blocked;
                struct ghost_msg_payload_task_wakeup    wakeup;
                ...
```

- Need to trick the compiler to not modify the register holding the ctx pointer?
- The verifier should think the context is fully modifiable…
  - ghost_msg_is_valid_access() returns true
- I’m probably messing up something…

Future Work

Implement the CFS algorithm in BPF

- Is it possible to implement complex scheduling policies purely in BPF?
  - e.g. loop limitations.
  - New MAP_TYPES needed?
- What changes are needed to Ghost? Are BPF-PNT and BPF-MSG sufficient?
- What is the “Ghost Tax”, the performance overhead of our mechanisms?
  - By having the same policy as kernel-CFS, we can do an apples-to-apples comparison
  - Also would like to try CFS in ghost-userspace
- Can tweak CFS-on-Ghost beyond the existing sysfs settings
  - And can do so for a subset of cpus instead of the entire machine

New MAP_TYPE for a Priority Queue / Heap?

- Would like a Map that's an O(log n) tree, e.g. rb tree
- bpf_rbtree map (RFC from davemarchevsky@fb.com)
- Probably can't just use existing bpf_map_helpers
- update, delete, pop, etc. probably aren't expressive enough for an rb tree.

New MAP_TYPE “preexisting memory blob”?

- All RAM for bpf maps is allocated by kernel/bpf/ code
- What if I want to look at a blob that came from somewhere else?
  - e.g. a device
  - e.g. I’m paravirtualized, and it is a host memory blob
- Want to treat it like an array map
- Instead of kmalloc (or vmalloc), it’s pinned memory (GUP, etc.)

Discussion

Can you implement Ghost’s ABI purely in BPF?

- *status_word_table*: (dense map of thread data, updated by the kernel)
  - Make it a BPF array map, managed by BPF-MSG handlers
- Ghost’s message infrastructure (channels, power-of-two rings, etc.)
  - BPF ring buffers + bpf_ghost_wake_agent() helper
- Agent Tasks (one per cpu) are special…
  - Run **above** CFS, and are also a token marking the CPU in use by an enclave
  - Not sure that is doable with BPF as easily…
- Userspace agents are **asynchronous**: Ghost-BPF can handle that
  - Messages have sequence numbers, which are passed back to the kernel for transactions
  - Makes sure the agent is acting on the current state of a task.
  - Any “implement ghost userspace on BPF” scheme would need something like that

Is Ghost right for other BPF-only scheduling frameworks?

- Important distinction between SCHED_CLASS_GHOST and user agents/ABI
- BPF-MSG isn't just "messages": it's the functional API from kernel to BPF
  - It's a switch statement, like a dispatcher syscall, e.g. fcntl()
  - You could have a separate PROG_TYPE for every message
- Even if you wanted only BPF schedulers, I'd still want the BPF-MSG interface
  - e.g. MSG_TASK_NEW: it's generated in 7 places in ghost.c! Lots of nuances about when threads change classes: were they on_cpu, were they about to block, did they join and leave before blocking, etc…
- Ghost solves the issue of safely delegating scheduling to some other agent
  - BPF or user space
  - Synchronous or asynchronous
  - Or at least tries to solve this issue. =)

Fin

- Main points:
  - Ghost: safe, extensible, kernel scheduling in both userspace and BPF-space
  - You can make a purely-BPF scheduler with Ghost
  - Biff: basic policy, example code for making your own scheduler
  - TBD: CFS, more advanced schedulers, MAP_TYPES, etc.
- Rough code
  - https://github.com/google/ghost-kernel
  - https://github.com/google/ghost-userspace
  - Tends to lag our in-house changes. Sorry.
  - Have to use “basel” to build the userspace libraries, for now. Sorry.

FAQ

FAQ: what about BPF task local storage?

- Per-task storage:
  - void *bpf_task_storage_get(struct bpf_map *map, struct task_struct *task, void *value, u64 flags)
- Can we use it? Not really.
  - ghost-bpf doesn't have visibility into the kernel's data structures
  - the contexts are ABI structs, e.g. struct bpf_ghost_msg
  - Tasks are referred to by ID, not by struct task_struct *.
- Even if you did use task_storage, it's not accessible to userspace (agent or application)

FAQ: can you do hybrid BPF and Userspace Agents?

- Original use of BPF was to accelerate and supplement userspace agents
  - I sketched this out at LPC 21 (slide 29)
- BPF-MSG's return value of 1 means "don't send this message to userspace"
  - BPF-MSG can filter messages
  - e.g. MSG_CPU_TICK (timer tick fired) - don't need to hear about that all the time!
- Ghost's message API was originally designed for slower, userspace agents
  - e.g. there was no MSG_CPU_UNAVAILABLE / AVAILABLE, since CPUs would come and go too quickly (whenever a CFS thread landed on_rq).
  - When tasks "SwitchTo" (Google's fast context switch syscall, Turner LPC 13), we don't send messages. Only send a message when a task starts a "switchto chain"
  - Too many messages for userspace, but not for BPF!

FAQ: what other BPF limitations have you run into?

- Limited loops, no floating point, communicate through Maps only, etc.
- Atomic compare and swap on 64 bit only
- Hand-written smp_store_release()?
    - Tried __atomic_store_n(&some_bool, false, __ATOMIC_RELEASE)
    - Had to do asm volatile ("" ::: "memory"); WRITE_ONCE(some_bool, false);

FAQ: what is the *status word*?

- The ghost kernel exports an mmapable file called the *status word table*
  - Every task in ghost has an entry in here
  - Contains info like “are you on_cpu” or “are you runnable”
  - Read-only to userspace
  - It’s a dense mapping: every task has an index into the table. O(65k) entries.
  - Made for fast info sharing to userspace agents, predates ghost-bpf.
- Biff uses a task’s *status word index* for its equivalent table: *Status Word Data*
  - We really just need an index allocator
  - Technically, we could have a QUEUE map of ints, loaded with 65k entries by userspace
  - The kernel gives us the status word index, so let’s use it
  - Though we could implement the status_word in BPF!

FAQ: what is an *enclave*?

- Enclave: a set of CPUs scheduled by a single Ghost agent
- Semi-hard partition: you can move CPUs between enclaves, but it requires the agent to yield the CPU
- One ghost-bpf program per attachpoint (e.g. BPF-MSG) per enclave
- BPF programs may run on CPUs outside an enclave
    - Consider a task woken up by an unrelated task on a cpu outside the enclave

FAQ: what about the global scheduling model?

- This is having a single CPU (in userspace) spin and schedule all of the cpus
  - Outlined at LPC 21 (slide 24-26)
  - Without BPF on every cpu, particularly BPF-PNT, you're just too slow for certain applications
- You can have a thread spin in userspace, monitoring and updating bpf maps
- You can pursue a hybrid approach, where that userspace thread occasionally overrides BPF. But synchronization is a pain. I've tried, and it's tricky.

FAQ: why not hook select_task_rq()?

- Determines which cpu’s struct rq (runqueue) to enqueue a waking task on
- The in-kernel RQ doesn’t really matter: the “real” RQ is in the agent
- When Ghost runs a task (bpf_ghost_run_gtid() or a transaction) it will migrate the task_struct from whichever struct rq it was on to the target struct rq
- If you knew where a task was likely to run, then putting it there when it wakes could be a slight performance win
- But not nearly as important as it is for in-kernel CFS
  - select_task_rq() is part of the scheduling policy for the kernel. But not for ghost.
- Have a per-enclave tunable for whether to wake on waker’s or wakee’s cpu
- Maybe we’ll add a hook for select_task_rq() if it’s important

FAQ: what are the RQ locking rules with ghost-bpf?

- An RQ lock is held during BPF-MSG
  - If the message is for task X, we hold the RQ lock for that task’s RQ
- No RQ is locked during BPF-PNT
  - This is so we can call bpf_ghost_run_gtid(task), which needs to grab both the task’s RQ lock and the current cpu’s RQ lock.

FAQ: any other Ghost improvements on the horizon?

- Maybe more BPF helpers:
    - “kill my agent / enclave”: things went poorly and we want to tear down the system
    - <Insert Your Helper Here>
- Remove userspace support stuff from kernel/sched/ghost.c: truly BPF-only! Perhaps that will make Ghost more upstreamable?
- Agents in other languages: since we aren’t scheduling with the agent tasks, we don’t need to write in low-level code (C or Rust). Just interact with Maps (Go, Python, whatever)

Fin

- Main points:
  - Ghost: safe, extensible, kernel scheduling in both userspace and BPF-space
  - You can make a purely-BPF scheduler with Ghost
  - Biff: basic policy, example code for making your own scheduler
  - TBD: CFS, more advanced schedulers, MAP_TYPES, etc.
- Rough code
  - https://github.com/google/ghost-kernel
  - https://github.com/google/ghost-userspace
  - Tends to lag our in-house changes. Sorry.
  - Have to use “basel” to build the userspace libraries, for now. Sorry.