LINUX PLUMBERS CONFERENCE September 20-24, 2021

eBPF in CPU Scheduler

Hao Luo <haoluo@google.com>
Barret Rhoden <brho@google.com>

Agenda

- Scheduling latency profiling
- Forced idle time accounting in core scheduling
- Using BPF to accelerate the ghOSt kernel scheduler

Scheduling Latency Profiling

Profile scheduling latencies

- Tracing programs attached to sched tracepoints
    - Approach similar to *runqslower*
        - Tool from Bpf Compiler Collection (BCC)
        - Trace long process scheduling delays
    - Attach points
        - sched_switch
        - sched_wakeup
- What’s profiled
    - Queueing delays: Time spent on waiting in run queues
    - Oncpu time: Time when using cpu
    - Offcpu time: Time scheduled off cpu

Cgroup-oriented profiling

- Queueing delay broken down into two parts
  - Wait time when a thread from the same cgroup is using the cpu
  - Wait time when a thread from another cgroup is using the cpu

- Identify starvations due to insufficient cpu shares

Report as distributions

- Profiled stats are organized in histograms
- Allow user configuration
  - Adjust bucket bounds
  - Reset values

- Service level indicator (SLI) for node management agent

Wins by using BPF for profiling

- Flexibility
  - Allow making changes easily and swiftly from userspace

- No kernel dependencies
  - Google kernel team adopts upstream-first approach.
  - Try to minimize the kernel patches carried internally.

Take away

- Cgroup-oriented profiling tool
  - Profile for jobs rather than threads
  - Differentiate types of starvations

- Reports distribution and allow customization
  - More insights
  - Better usability

Forced Idle Time Accounting

Core scheduling

- Cross-HT attack
  - Involves attacker and victim running on different Hyper Threads of the same core.
  - Example: L1TF and MDS

- Core scheduling
  - Mitigation for some cross-HT attacks
  - Ensure only tasks in a user-designated trusted group can share a core (example followed)
  - Expected better performance, compared to the option of disabling HT

Core scheduling

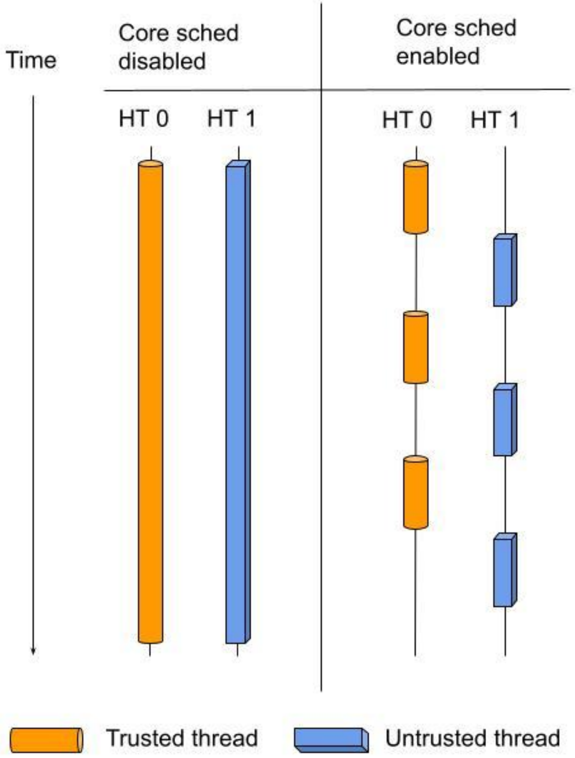
Core scheduling isolates trusted and untrusted tasks' execution.

When running untrusted task, the sibling cpu either

1. runs a task from the same untrusted group.

Or

2. forced idle.



Forced idle time

- Correct accounting of resource consumption requires attributing forced idle time to the untrusted group.
    - Before, reported cpu usage = real cpu usage
    - After, reported cpu usage = real cpu usage + forced idle time

- Why
    - Good indicator of core scheduling’s efficiency.
    - Opportunity cost of running untrusted tasks.

Measure forced idle time

- No upstream solution exists yet.
  - Challenging scenarios
    - How about >2 HT siblings?

- Using BPF
  - Provides a fast and flexible way to measure forced idle time.
  - Signal for tuning scheduling happening at userspace.

BPF Solution

- Tracing programs attached to sched tracepoints
    - Attach points
        - sched_switch
        - sched_wakeup
    - Attach points are within core scheduling’s critical section.
        - Not concerned about race between HT siblings.

Sibling HT’s state

Detecting forced idle requires us to know whether the sibling HT is idle.

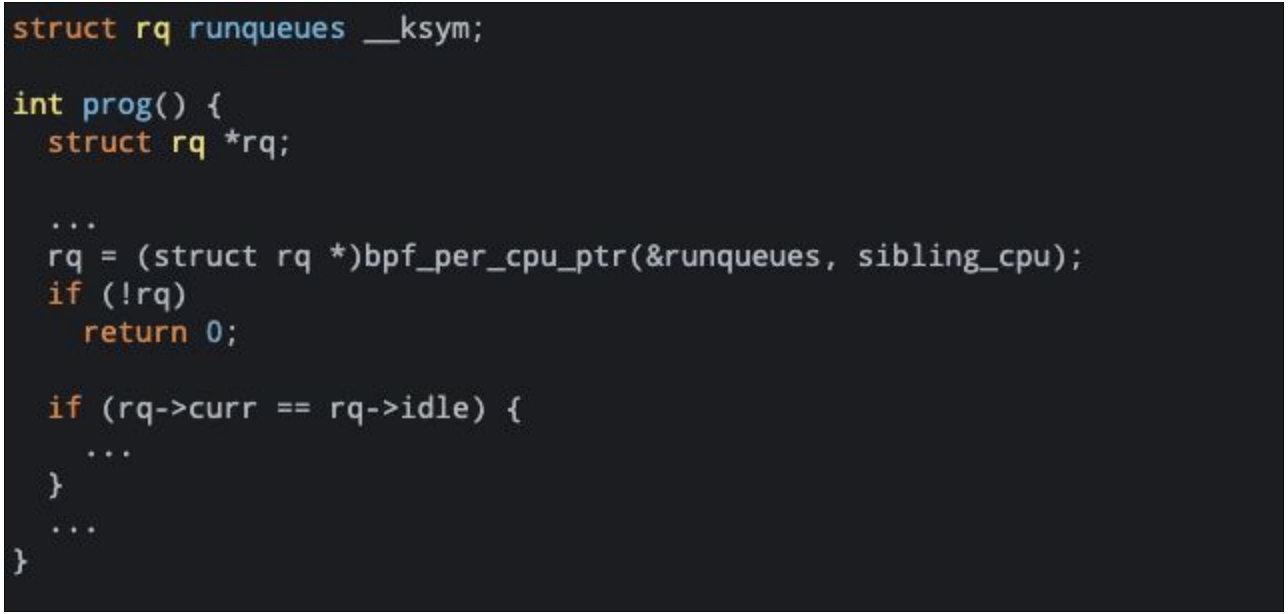
- Read from sibling_rq->curr
- Required to access sibling HT’s runqueue within BPF programs

Ksyms

In BPF program, one can declare a symbol as a *ksym*. If kernel has exported a global symbol of the same name, one can read the exported kernel symbol via the ksym (example next page).

- Libbpf reads the symbol’s kernel address from kallsyms.
- Kernel BTF is needed if wants to direct dereference the symbol.
- BPF verifier makes sure the access is safe.

Finding whether the sibling HT is idle



```
struct rq runqueues __ksym;

int prog() {
  struct rq *rq;

  ...
  rq = (struct rq *)bpf_per_cpu_ptr(&runqueues, sibling_cpu);
  if (!rq)
    return 0;

  if (rq->curr == rq->idle) {
    ...
  }
  ...
}
```

Algorithm

At context switch, perform the following operations (SMT=2 only),

1. Take timestamp for entering forced idle, if
    - (1) sibling_rq->curr is idle and (2) context switch to untrusted task
    - (1) self is running untrusted task and (2) sibling switches to idle

2. Take timestamp for exiting forced idle, if
    - Case I
        - (1) sibling_rq->curr is untrusted task and (2) context switch from idle
    - Case II
        - (1) sibling_rq->curr is idle and (2) context switch from untrusted task

3. Charge forced idle time
    - If case I, charge the time to sibling_rq->curr
    - If case II, charge the time to current

Take away

- Implementing sched stats using BPF is a promising idea.

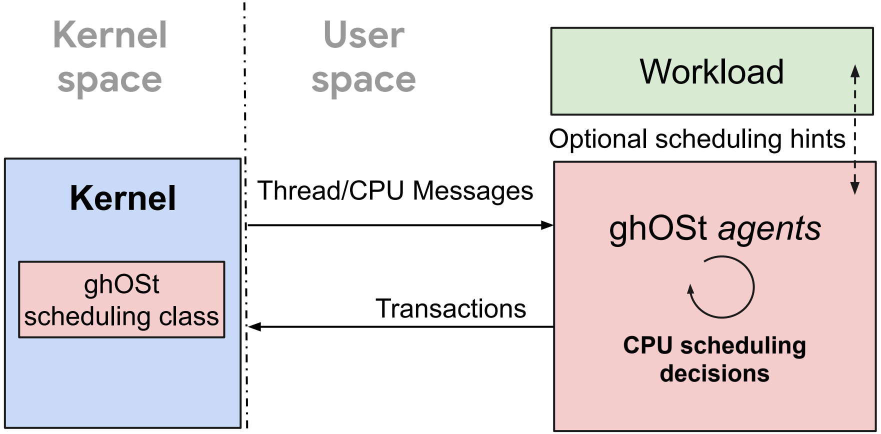
- The ability to read per-cpu variables within BPF programs enables many sched BPF applications.
    - Sched uses per-cpu variable extensively.

ghOSt + BPF

Using BPF to accelerate ghOSt

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”



Why ghOSt?

- Workload-specific scheduling policies
    - Different policies for hosting virtual machines versus running search engines
    - Agent-to-application interface is independent of the kernel ABI
- Update the scheduling policy independently from a kernel rollout
- More details: ghOSt: Fast & Flexible User Space Delegation of Linux Scheduling (Netdev 0x15 (2021))

Messages and Transactions

- Both are through shared memory, plus a “poke”
- *Messages*: from the kernel to the agent:
  - Ring buffer for the payload
  - Wake an agent on a particular cpu (not necessarily where the event occurred)
- *Transactions*: from the agent to the kernel
  - Per-cpu array of *struct ghost_txn*
    - GTID (PID), cpu, txn_state, task_barrier, agent_barrier, run_flags, commit_flags, commit_time, cpu_seqnum, sync_group_owner
  - Syscall to ask the kernel to look at specific transaction requests
  - Instructs pick_next_task_ghost() to run a particular task next: called the *latched task*

Various Multicore Scheduler Styles

- *Per-cpu* scheduling: an agent task on **each** cpu schedules **its** cpu
- *Global* scheduling: an agent task on **one** cpu schedules **all** cpus
- Hybrid: switch between per-cpu and global models

There's an agent task on every cpu; userspace determines which do what.

Global Scheduling Woes

- Typical global agent loop (spinning):
  - Handle messages
  - Schedule runnable tasks on available cpus
  - Fancy policy stuff: preempt low priority tasks with higher priority tasks
- On a large machine (112 cpus), the loop can take a while
  - Workload dependent: how many wakeups per second
  - Scheduling policy dependent: complex policy may take a while to compute
- On average, 30-60us…
  - … is the average amount of time until the agent responds to a message
  - … is the average amount of time a cpu sits idle before the agent schedules it
- That’s way too slow: every time a task blocks, we waste 30us?!?!

Global Scheduling Woes (from schedghostidle)

```
Latency of a CPU going Idle until a task is Latched:
----------------------------------------------------
     usec               : count     distribution
         0 -> 1         : 0        |                                        |
         2 -> 3         : 3        |                                        |
         4 -> 7         : 98       |                                        |
         8 -> 15        : 266      |                                        |
        16 -> 31        : 2784     |                                        |
        32 -> 63        : 283485   |************                            |
        64 -> 127       : 904240   |****************************************|
       128 -> 255       : 150271   |******                                  |
       256 -> 511       : 4852     |                                        |
       512 -> 1023      : 481      |                                        |
      1024 -> 2047      : 47       |                                        |
      2048 -> 4095      : 1        |                                        |
```

This is the global agent’s loop time

Use BPF to respond quickly to events

- When pick_next_task_ghost() has no *latched task*, we could:
  - Idle. And then wait for the global agent to notice and issue a transaction… no thanks!
  - Wake that cpu's agent, which can issue a transaction… extra context switches
  - Run a bpf program, which can also issue a transaction!
- *BPF-PNT*
  - BPF_PROG_TYPE_GHOST_SCHED
  - Attached in pick_next_task_ghost()
- BPF Helpers:
  - bpf_ghost_wake_agent(cpu): kick the agent on a cpu
  - bpf_ghost_run_gtid(task, …): essentially the same as a transaction

BPF Programs are **part of the Agent**

- Closely coupled to the userspace agent
  - Embedded in the agent binary, libbpf-style, with a bpf skeleton
  - Has the same lifetime as the agent: agent holds the FD from BPF_LINK_CREATE
  - Coded side-by-side: e.g. edf_scheduler.cc and edf.bpf.c
- Share memory with the userspace agent
  - BPF_MAP_TYPE_ARRAY: mmapped by userspace
- Act as an agent ‘thread’, with similar privileges as userspace
- “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.

Example: BPF scheduler with a Global Agent

- The agent pushes runnable tasks into (yet another) shared memory ring buffer
  - BPF-PNT consumes tasks as cpus idle; latches them in pick_next_task_ghost()
  - This is not an ABI: it's between the agent Ring-3 and the agent Ring-B code
- Can have a hierarchy of ring buffers, based on the cache hierarchy
  - BPF-PNT looks in per-cpu, then per-numa rings, etc.
- Global agent monitors the tasks in the rings
  - Moves tasks from cpu to numa, based on an SLO or between cpus for load balancing
  - If a high priority task doesn't run in X usec, issue a transaction to preempt some other task

- You (the agent) can come up with whatever you want, independent of the kernel
  - Just like with userspace-only ghOSt, now you have BPF too.
  - e.g. maybe implement a BPF_MAP_TYPE_PRIORITY_QUEUE and have per-cpu runqueues.

Global Scheduling with BPF-PNT

```
Latency of a CPU going Idle until a task is Latched:
----------------------------------------------------
     usec               : count     distribution
         0 -> 1          : 322273   |****************************************|
         2 -> 3          : 621      |                                        |
         4 -> 7          : 7775     |                                        |
         8 -> 15         : 23681    |**                                      |
        16 -> 31         : 39125    |****                                    |
        32 -> 63         : 33892    |****                                    |
        64 -> 127        : 19430    |**                                      |
       128 -> 255        : 8127     |*                                       |
       256 -> 511        : 3297     |                                        |
       512 -> 1023       : 675      |                                        |
      1024 -> 2047       : 60       |                                        |
      2048 -> 4095       : 1        |                                        |
```

This is when BPF-PNT found a task to run

This is the global agent loop still. If the agent falls behind on handling messages, BPF-PNT has no tasks to run

What about wakeups?

- It’s not enough to have BPF only at pick_next_task()
  - Respond quickly to wakeups and other runnability edges (yields, preemptions from CFS)
  - Keep BPF-PNT busy with tasks to run; e.g. push tasks into those shared memory rings
- Remember *messages*?
  - Messages are the primary mechanism for the kernel to inform the agent of a ghost event
  - BPF is part of the agent; let’s interpose on message delivery!
- *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)
- Can we replace ghost’s messaging backend with BPF_MAP_TYPE_RINGBUF?
  - Conceptually, yes. Both are shared-memory ring buffers.
  - It’d require all ghost agents to use BPF.
  - It’d allow agent-specific customizations to message payloads.

Do you need a userspace agent?

- Maybe not! But it’s all the same agent program
    - Messages are the interface to the agent, whether the agent is in Ring-3 or Ring-B
- Set of desired policy operations:
    - “Run task X on cpu 3 now”
    - “Set need_resched on cpu 5”
    - “Let cpu 6 go into a deep C state”
- Ghost’s kernel code solves the hard problems of delegating scheduling to an untrusted agent
    - Which messages to send, their semantics and parameters, etc.
    - e.g. from how many places in the kernel do we need to send MSG_TASK_NEW? 5!
- Some code is easier in userspace
    - Easily communicate with applications and system daemons (RPCs, etc.)
    - Can spin in a loop, monitoring system progress (global agent style), issuing preemptions
    - Monitor devices, e.g. flash or NIC, to adjust task priorities
    - Use complicated data structures
    - No battles with the verifier! =)
- For an agent that ran primarily in BPF, I’d still want a userspace component

ghOSt + BPF

- Main points:
    - Ghost: delegate kernel scheduling to an agent process
    - Agent composed of userspace and BPF programs
    - Use BPF as an accelerator to recover the overheads of going out and back to userspace
- I glossed over everything unrelated to BPF:
    - Netdev 0x15 talk
    - Upcoming SOSP21 paper (no link yet)
- Code
    - https://github.com/google/ghost-kernel
    - https://github.com/google/ghost-userspace
    - Sorry, this doesn't have the latest bpf stuff yet, but it does have BPF-PNT