Kernel Scalability

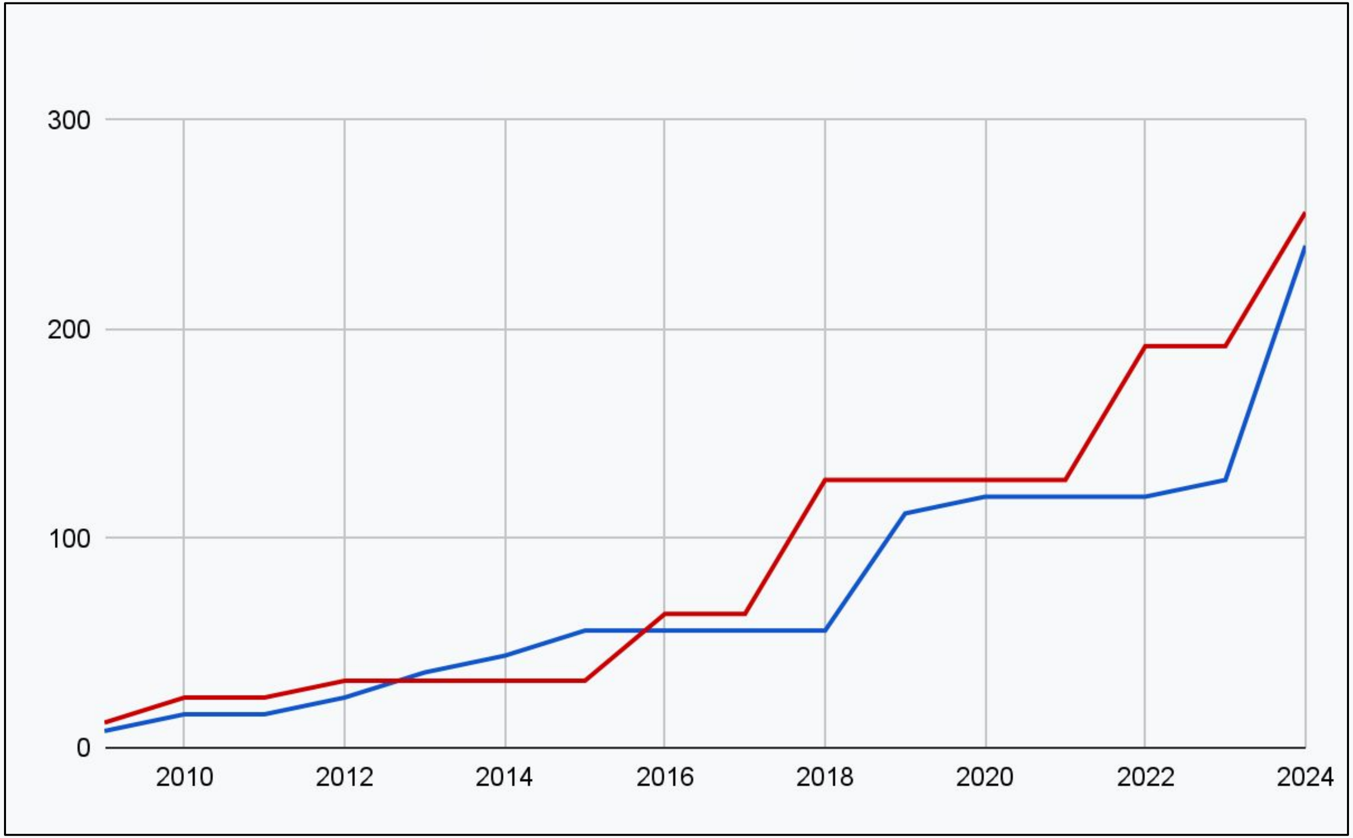
LINUX PLUMBERS CONFERENCE | Vienna, Austria
Sept. 18-20, 2024

Josh Don

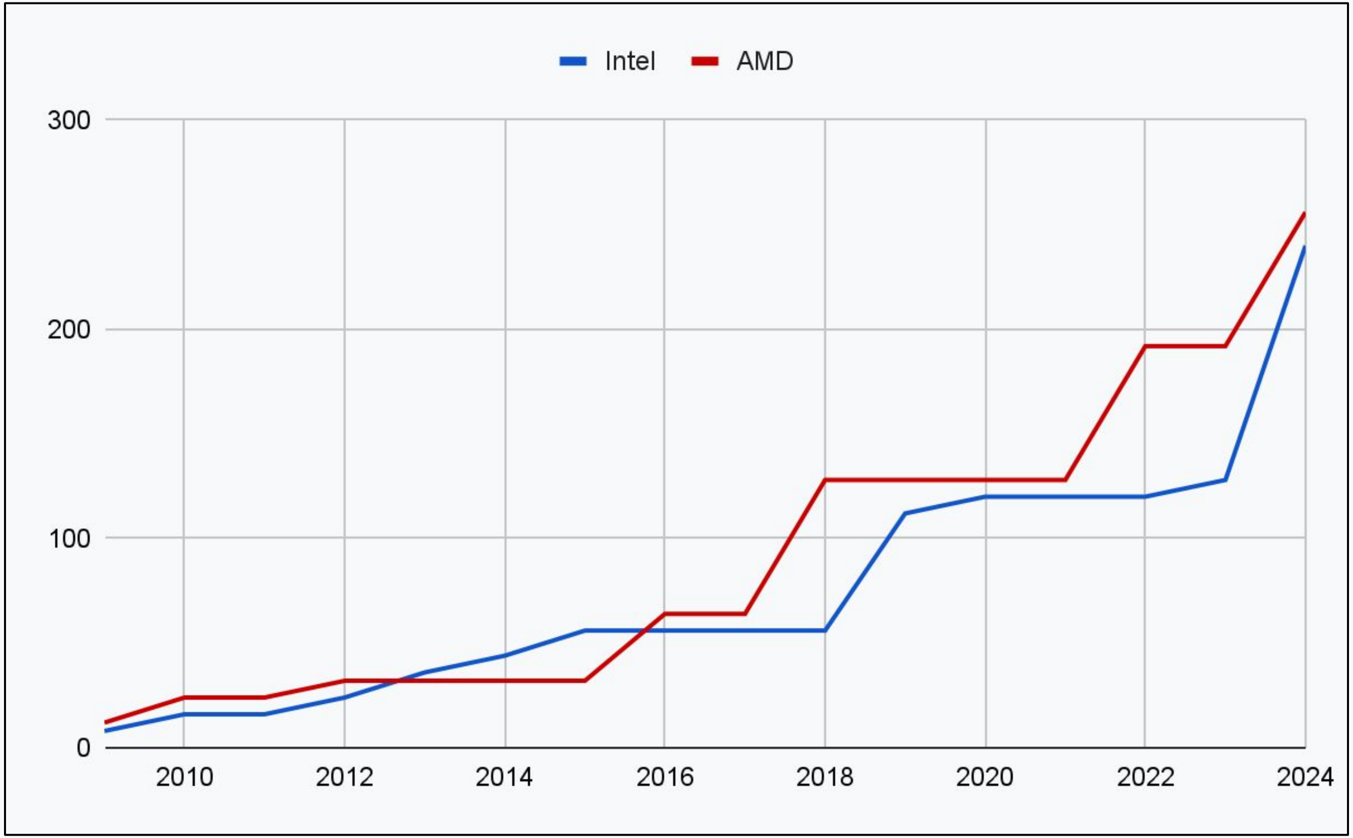
01

Introduction

What does this graph represent?



Per-**socket Logical** Processor Count



Scalability Bottlenecks

per-cpu iteration

- More cpus => iteration takes longer
- Includes sources of implicit iteration (ie. wait for IPI broadcast)

per-thread/process iteration

- Larger machine means more work will be piled on

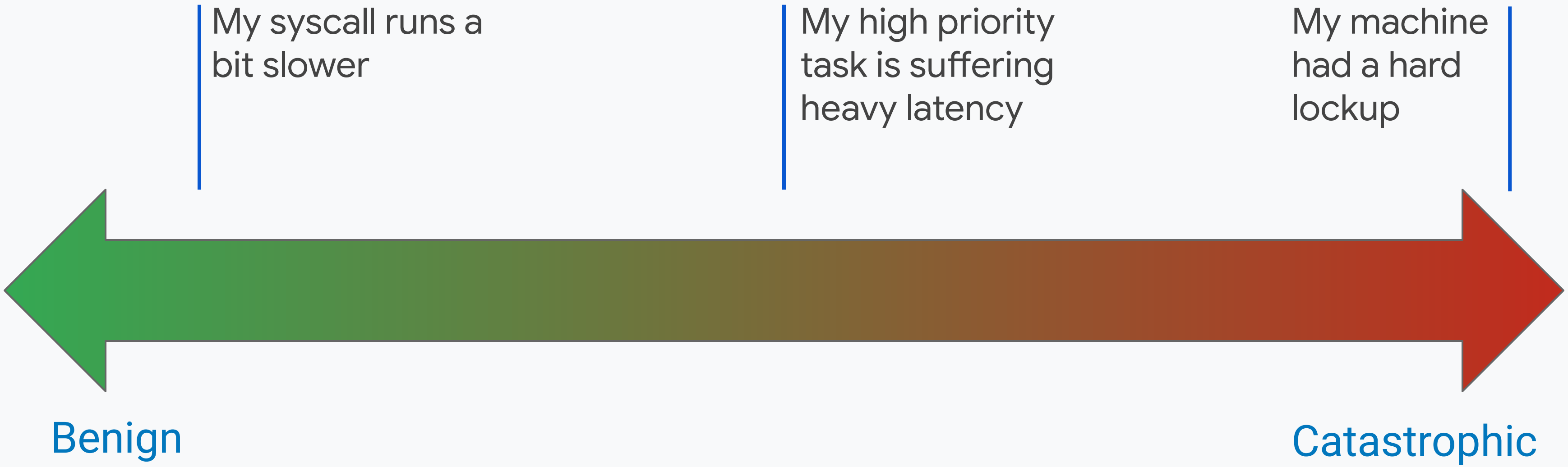
per-cgroup iteration

- Larger machine can land more users in a cotenant environment

lock contention

- Lock granularity doesn’t scale with core count
- Global locks are particularly bad here

What Can Happen?



Benefits of Scalability

Efficiency

Fully utilize machine resources by packing as many tasks as possible. Bigger chip = better perf / TCO.

Performance

Remove bottlenecks that slow down execution.

Reliability

Avoid lockups and other sources of instability.

Scalability Theorem

Scalability

Can it scale to large systems?

Velocity

Can we develop and integrate it quickly?

Upstream

Can we get it merged upstream?

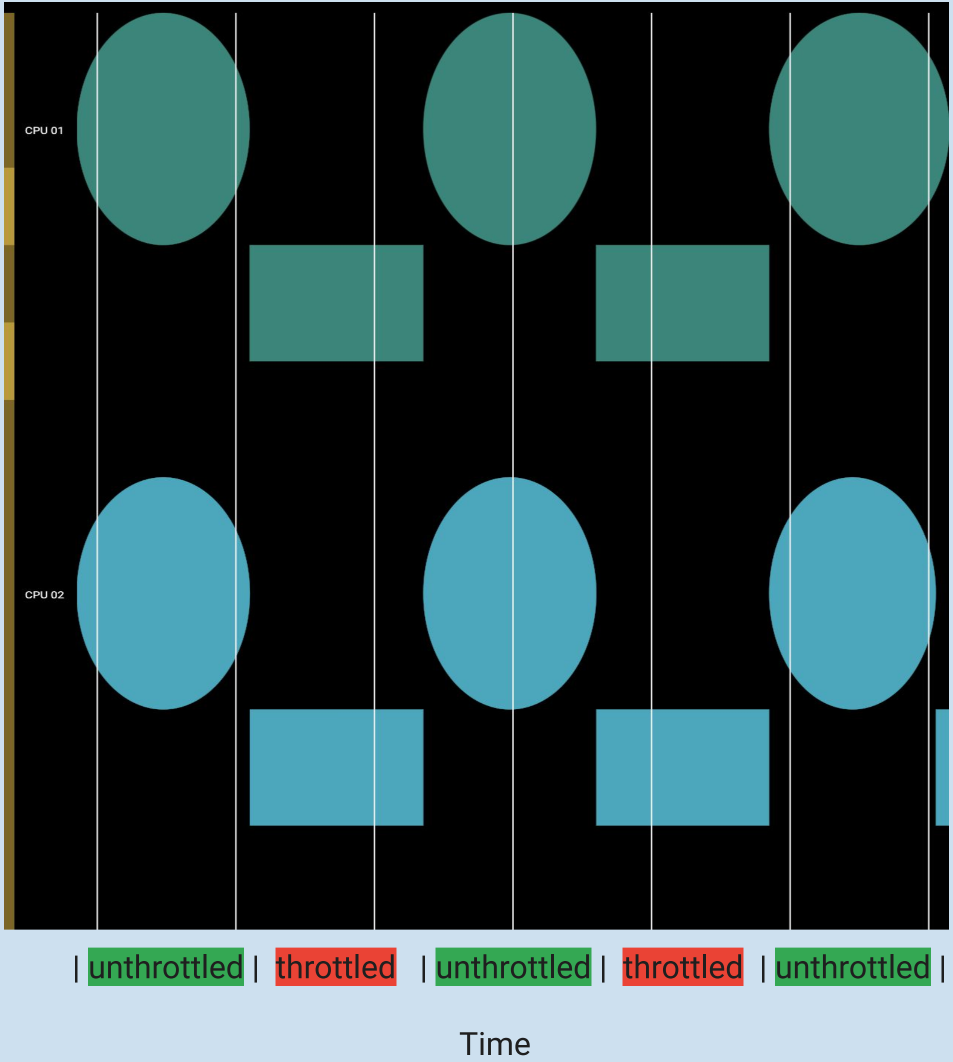
(Generally) pick any two

02

Case Studies of Resolved Issues

CFS Bandwidth Throttling

- Bandwidth control limits cgroups to a fixed cpu quota per period
- An hrtimer goes off every period to refresh quota and unthrottle all throttled cpus



A Closer Look at Quota Refresh

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

A Closer Look at Quota Refresh

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

O(cpus)

O(cgroups)

A Closer Look at Quota Refresh

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

O(cpus)

O(cgroups)

- We're in hrtimer context (ie. hardirq); IRQ are disabled for the duration
- Result was a hard lockup (10+ seconds stuck without running IRQ)
    - 256 cpus and O(1000) cgroups in the throttled cgroup hierarchy

A Look at The Solution

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq_async(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

O(cpus)

O(1)

- Dispatch the unthrottle to the remote cpu, rather than doing it inline, thus sharding the O(cgroup) work to the entire system

8ad075c2eb1f6: sched: Async unthrottling for cfs bandwidth

A Look at The Solution

8ad075c2eb1f6: sched: Async unthrottling for cfs bandwidth

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,      <- O(cpus)
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq_async(cfs_rq);      <- O(1)

                rq_unlock_irqrestore(rq);
        }
}
```

Still suffers from O(cpus) in hrtimer context, but more on that later…

getrusage syscall

- Returns resource usage information for the current process

```c
int main() {
    struct rusage usage;

    if (getrusage(RUSAGE_SELF, &usage) == -1) {
        perror("getrusage");
        return 1;
    }

    printf("User time: %ld.%06lds\n"
           "System time: %ld.%06lds\n"
           "Max RSS: %ld bytes\n"
           "Voluntary context switches: %ld\n"
           "Involuntary context switches: %ld\n",
           usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
           usage.ru_stime.tv_sec, usage.ru_stime.tv_usec,
           usage.ru_maxrss,
           usage.ru_nvcsw,
           usage.ru_nivcsw);


    return 0;
}
```

A Closer Look at getrusage

```
void getrusage(struct task_struct *p, int who, struct rusage *r)
{
        struct task_struct *t;

        lock_task_sighand(p)

        switch (who) {
        case RUSAGE_SELF:
                t = p;
                do {
                        accumulate_thread_rusage(t, r);
                } while_each_thread(p, t);
                break;
        }

        unlock_task_sighand(p);
}
```

A Closer Look at getrusage

```
void getrusage(struct task_struct *p, int who, struct rusage *r)
{
        struct task_struct *t;

        lock_task_sighand(p)

        switch (who) {
        case RUSAGE_SELF:
                t = p;
                do {
                        accumulate_thread_rusage(t, r);
                } while_each_thread(p, t);
                break;
        }

        unlock_task_sighand(p);
}
```

Per process spinlock

Iterate all threads of the process

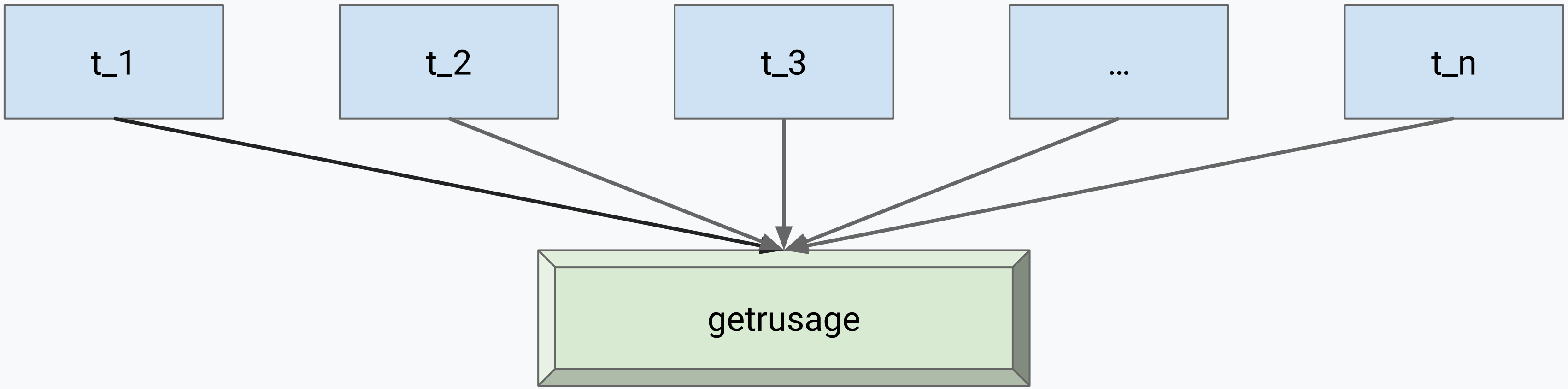
A Closer Look at getrusage

```
void getrusage(struct task_struct *p, int who, struct rusage *r)
{
        struct task_struct *t;

        lock_task_sighand(p)

        switch (who) {
        case RUSAGE_SELF:
                t = p;
                do {
                        accumulate_thread_rusage(t, r);
                } while_each_thread(p, t);
                break;
        }

        unlock_task_sighand(p);
}
```

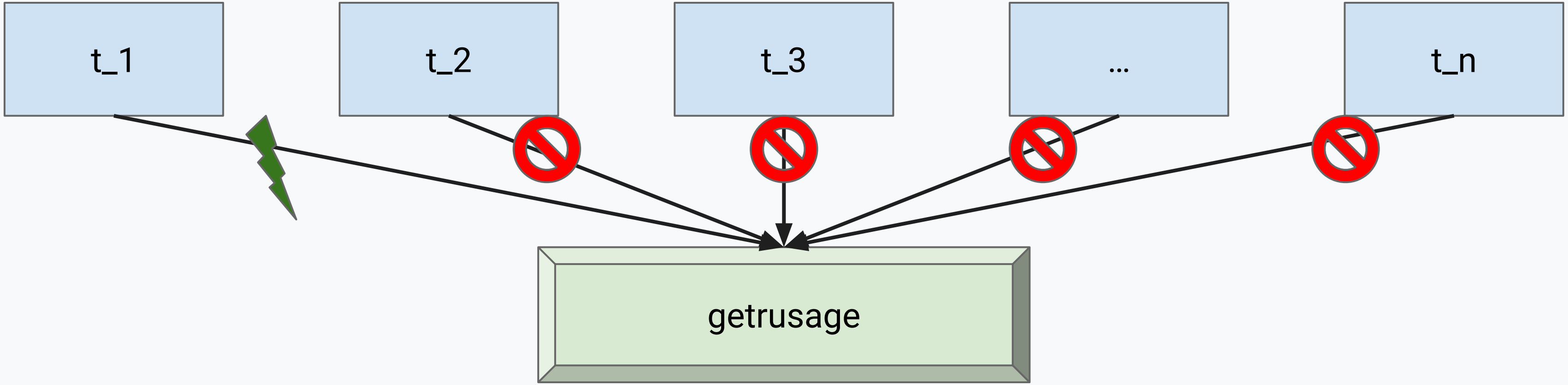
Actually… per process spinlock that spins with **IRQ disabled**

Iterate all threads of the process

What Was the Problem?



What Was the Problem?



- Threads in a process can call getrusage concurrently, only one can make progress at a time
- Each takes a long time in the critical section due to O(threads) iteration
- user process with O(250k) threads triggered a hard lockup by a userspace bug in which multiple threads called getrusage at the same time
  - Userspace bug, but… **this shouldn’t cause a kernel crash**

A Look at The Solution

```
void getrusage(struct task_struct *p, int who, struct rusage *r)
{
        struct task_struct *t;

retry:
        read_seqbegin_or_lock_irqsave(&sig->stats_lock, &seq);

        switch (who) {
        case RUSAGE_SELF:
                t = p;
                do {
                        accumulate_thread_rusage(t, r);
                } while_each_thread(p, t);
                break;
        }
        if (need_seqretry(&sig->stats_lock, seq)) {
                seq = 1;
                goto retry;
        }
        done_seqretry_irqrestore(&sig->stats_lock, seq, flags)
}
```

Run locklessly in common case of readers only

03

CFS Bandwidth Control

Recall the Quota Distribution Handler

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq_async(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

O(cpus)

O(1)

Recall the Quota Distribution Handler

```
static void distribute_cfs_runtime(struct cfs_bandwidth *cfs_b)
{
        list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                                throttled_list) {
                struct rq *rq = rq_of(cfs_rq);

                rq_lock_irqsave(rq);

                cfs_rq->runtime_remaining += refresh;
                if (cfs_rq->runtime_remaining > 0)
                        unthrottle_cfs_rq_async(cfs_rq);

                rq_unlock_irqrestore(rq);
        }
}
```

Could take non-trivial time; O(cgroup) throttling holds rq lock

We’re not yet safe from bandwidth distribution

- O(cpus) iteration could be slow
  - Worst case, we’re back to our hard lockup (unlikely)
    - **Idea:** Shard the timer callback to multiple cpus (complex and unlikely unnecessary at this point)
  - A `cfs_rq` we unthrottle could get re-throttled in the same iteration
    - **Idea:** Don’t revisit the same cpu more than once in a given iteration (we could unthrottle cpu X, then cpu X could throttled again before we’re finished with the iteration)

- Wait… what about the O(cgroup) throttling operation?

Throttling Scalability

- Throttle/unthrottle still has an O(cgroup) scalability factor
    - `walk_tg_tree_from(cfs_rq->tg, tg_throttle_down, tg_nop, (void *)rq);`
- Done with rq lock held!
- Why do we do this tree walk?
    - Some statistics updates
    - Increment throttle count of all child cgroups
        - Allows O(1) detection of throttled hierarchy on task enqueue, migration, etc.
        - Maybe worthwhile to compute throttled hierarchy state lazily? Common case of enqueue already does an ancestor walk (**`h_nr_running`** updates, etc.)

Throttling Scalability

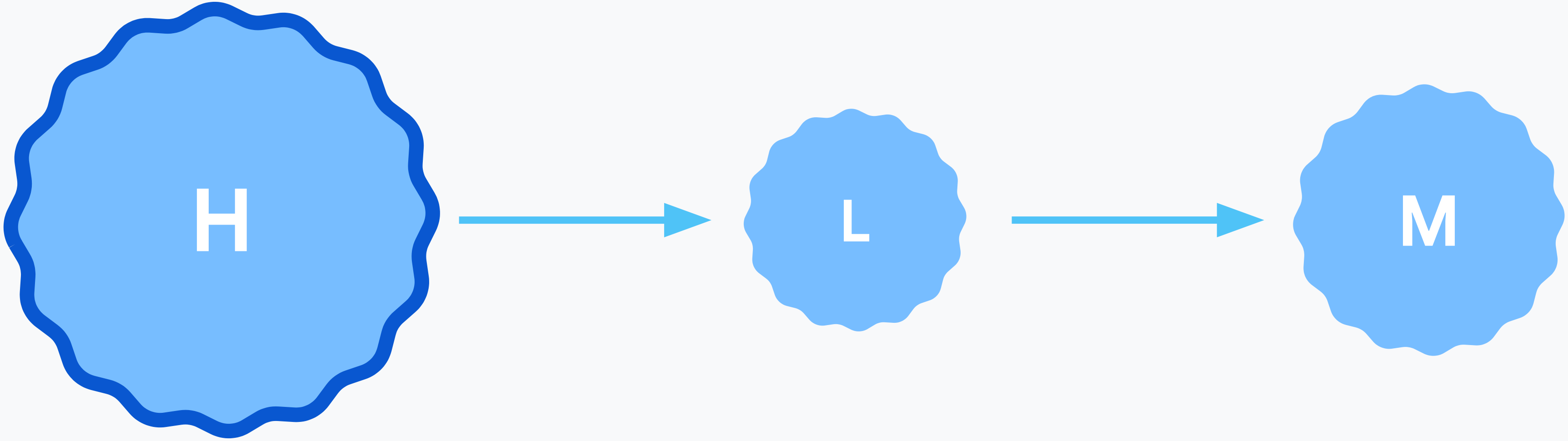
- So far, not causing extreme pain, but this is a consistent bottleneck
- Being **proactive** vs reactive
- Maybe no one else cares that much about scaling CFS bandwidth to this number of cpus and cgroups?
- Making this scalable will shift the overhead to be more distributed on the time axis, but that might negatively impact some users
- Increased code complexity

- **Should the kernel keep the simplicity and runtime benefits of the current model, or sacrifice these somewhat to be more scalable?**

04

Priority Inversion

Classic Priority Inversion



Priority Inversion vs Scalability

- More threads => more contention over shared resources, longer tails of wait queues
  - Particularly with coarse locks like `cgroup_mutex` and `mmap_lock`
  - Including more abstract resources like memory bandwidth

- Things are starting to look a little better here…
  - Proxy execution to mitigate prio inversion due to locking (mutex only)
  - Internal experiments to prioritize execution of threads in kernel context (see Xi Wang's LPC talk)
  - Per VMA locking

Internal Experiments

kernel mutex wait time P99

~50% reduction in kernel lock max wait time

~67% reduction in cgroup_mutex max wait time

~40x reduction in watchdog panic rate

05

Perf

Uncore management

- Larger CPU = more uncore PMUs
  - L3 cache uncore PMU count grows linearly with pcore count
  - eg. Granite Rapids with 120 pcores has ~150 uncore PMUs
- Problem: A single CPU per socket is designated to manage uncore PMUs
  - Events could also be multiplexed, which requires uncore management rotate events every millisecond from hrtimer context
- **Solvable:** Can fix this by sharding uncore management
  - On our backlog; on paper doesn’t appear infeasible

Perf tool

- Creating event counters on multiple cpus is bottlenecked by a per-cpu iteration
  - Kernel API only installs on a single cpu per call
  - Profiling on hundreds of cpus requires iterative sched set_affinity or IPI to create all events
  - **Solvable:** Kernel can expose an API to install on multiple cpus via broadcast
- Ian Rogers: adding parallelism is a theme of things to do in the perf tool

Perf tool

- When perf tool starts in profiling mode it has to first look at all mmap entries under /proc in order to symbolize samples
    - Bigger machine = more processes
    - **Solvable**: Ian Rogers working on alternative to avoid mmap scan and instead include build ID + text offset for each sample.
        - Trade-offs; for example, increase each record size by 24 bytes to support build ID => better when sampling for an infrequent event

06

Memory Management

Lock Contention

- Lock granularity continues to be a scalability concern, not just with size of memory, but **number of cpus per node**
- Examples
  - **mmap_lock**: protects VMA lookup
    - per-VMA locking is helping, but still observe multi-second page fault tails waiting on mmap_lock (possibly due to reader/writer contention)
  - **LRU lock**: protects LRU list for working set
    - A single LRU lock can protect a lot of memory, depending on length of the list
    - List operations are frequent
  - **Zone lock**: protects free pages of each mm zone for page allocation
    - Each NUMA node has multiple zones, but still one big lock per zone
  - **Swap lock**: protects swap device files
    - Mitigated somewhat by using multiple swap files per machine

Per-cpu structures

- Many structs are allocated per-cpu
  - **Benefit:** lockless access to cpu local struct
  - **Downside:** increased memory overhead

- More nuanced downside: aggregation takes longer on larger machines
  - e.g. rstat
    - stats tracked per-cpu
    - reads from userspace trigger an aggregation that follows a **`for_each_possible_cpu()`** iteration to do the flushing
    - userspace doing frequent observations suffers, especially when observing multiple cgroups, as each must do a separate iteration

07

Other Quick Examples

NOHZ

- Timer migration scans O(cpus)

```
int get_nohz_timer_target(void)
{
        for_each_domain(cpu, sd) {
                for_each_cpu_and(i, sched_domain_span(sd), hk_mask) {
                        if (cpu == i)
                                continue;

                        if (!idle_cpu(i))
                                return i;
                }
        }

        return this_cpu;
}
```

NOHZ

- Timer migration scans O(cpus)

```
int get_nohz_timer_target(void)
{
        for_each_domain(cpu, sd) {
                for_each_cpu_and(i, sched_domain_span(sd), hk_mask) {
                        if (cpu == i)
                                continue;

                        if (!idle_cpu(i))
                                return i;
                }
        }

        return this_cpu;
}
```

CPU iteration

Lots of remote accesses;
poor cache locality

NOHZ

- Timer migration scans O(cpus)

```
int get_nohz_timer_target(void)
{
        for_each_domain(cpu, sd) {
                for_each_cpu_and(i, sched_domain_span(sd), hk_mask) {
                        if (cpu == i)
                                continue;

                        if (!idle_cpu(i))
                                return i;
                }
        }

        return this_cpu;
}
```

CPU iteration

Lots of remote accesses;
poor cache locality

Solutions:

- disable `sysctl.timer_migration`
- place a search limit on the loop (something we should probably do in general…)

Slow task death

- KILL'ing a large process can be slow
  - We walk all its threads and trigger a wakeup
  - Wakeup on a large system might be non-trivial, due to wakeup (`select_task_rq`) heuristics

- **Simple workaround:** short-circuit wakeup selection for dying tasks to pick the last used cpu

Chiplet Architecture

- Larger CPUs tend to have sub-NUMA nodes
    - Chiplets are a way to improve scalability in the hardware

- Chiplets create asymmetric architecture
    - Split L3 cache

- Increased hardware complexity to support scalability means we also need to make the software more complex
    - Chiplet scheduling is active area of open research
        - Soft affinity to a particular chiplet
        - When to queue on local chiplet vs spill to a remote chiplet

08

Q&A