Understanding Your Hypervisor: Why Open Source Matters

Ryo Takakura, Koichiro Den
Akiko Yano, Hirokazu Takahashi

VA LINUX SYSTEMS JAPAN

VA Linux Systems Japan K.K.

5/13/2026

Introduction: Who am I?

- Ryo Takakura
- Linux Kernel Engineer/Hobbyist
- Working for VA Linux
  - Was taking part in Xen back in late 2000s
  - Blog: https://www.valinux.co.jp/blog/
- I got into Xen recently

Disclaimer

This case study shows how a better understanding of a system brings more than just better performance.

Note that the observed improvement is meaningful but modest, and was measured under a tightly controlled Context1 setup.

Also, the presentation only covers the rough outline, so refer to the Appendix for the full process and detailed results.

Agenda (Today's Goal)

Goal: Answering "Why Open Source?"

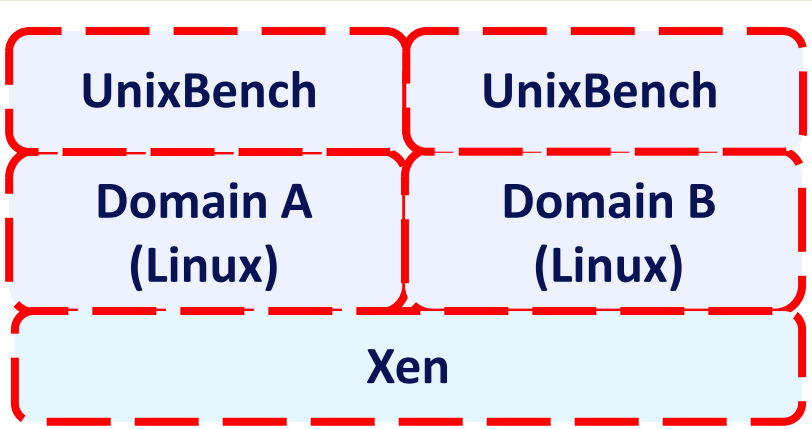
Roadmap

- How does Xen, Linux, and UnixBench work?
- How do they interact as a system?
  - ➢ How did it help to improve the system?
- Conclusion
  - ➢ Answering "Why Open Source?"
    - ✓ Better score
    - ✓ Predictable behavior
    - ✓ Reliable performance

Overview

Q: Xen, Linux, and UnixBench?

- **Hypervisor: Xen v4.20**
- **Domain: Linux v5.10**
- **Workload: UnixBench Context1**



UnixBench Context1 measures system's scheduler.

➢ To get the best out of it, we want to understand how they work together as a whole. But first, we need to understand how each works.

Q: How does each scheduler work?

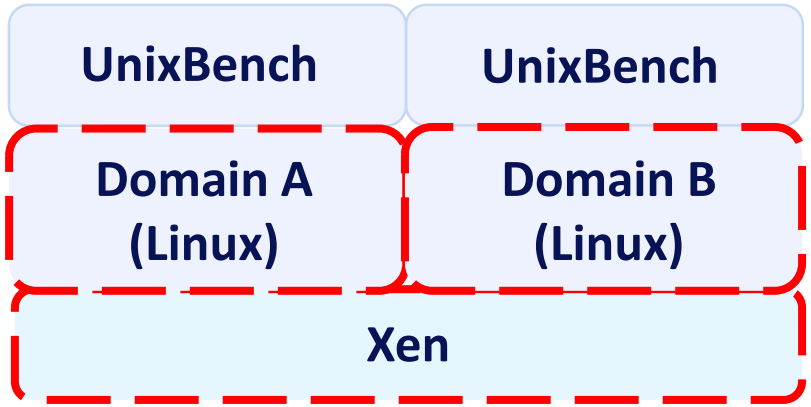
Xen Schedulers

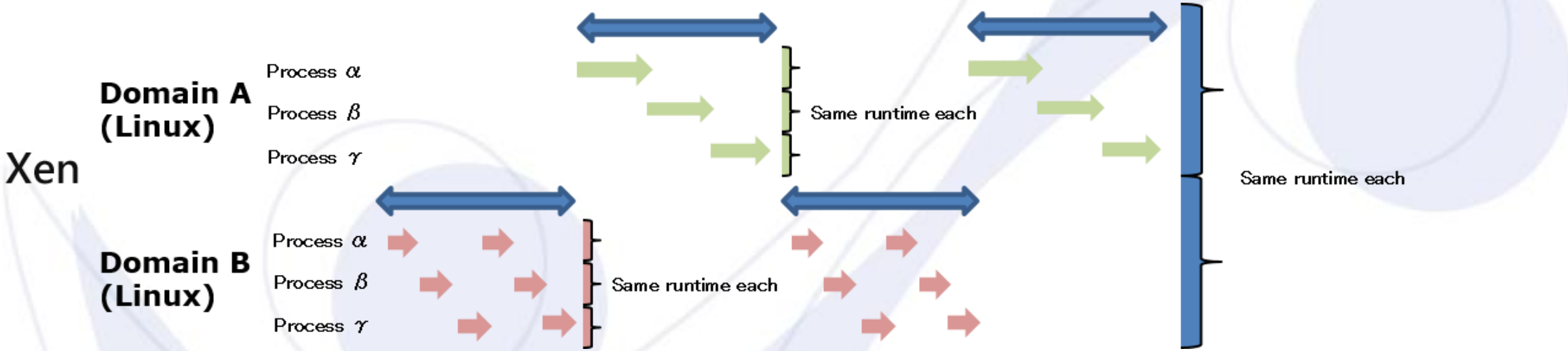
Scheduler	Suited for
Credit	General purpose, weighted fair-sharing
Credit2	General purpose, weighted fair-sharing + Better latency, more scalable… (Default scheduler)
RTDS	Real-time scheduling
ARINC653	Deterministic scheduling with fixed periodicity
Null	Static/dedicated CPU assignment

How Do Xen and Linux work?

Xen: Credit2 Scheduler
Linux: Completely Fair Scheduler(CFS)

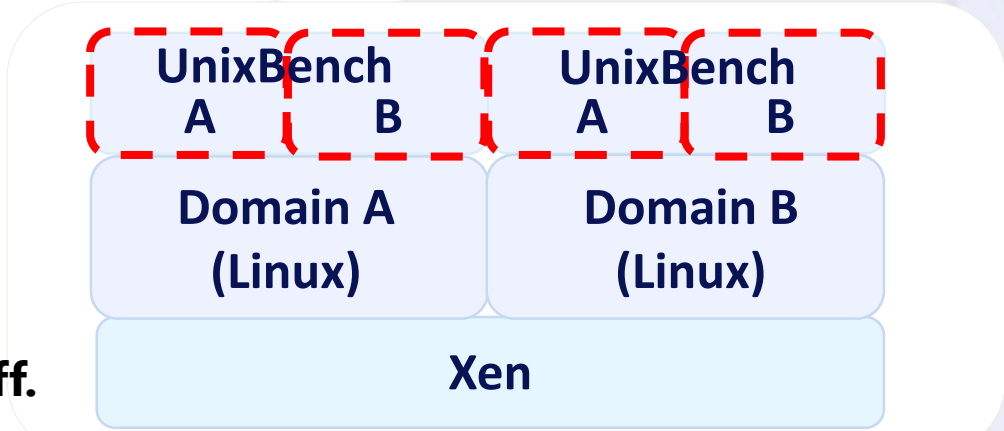
- They are both fair schedulers, trying to balance its domain/processes' runtime





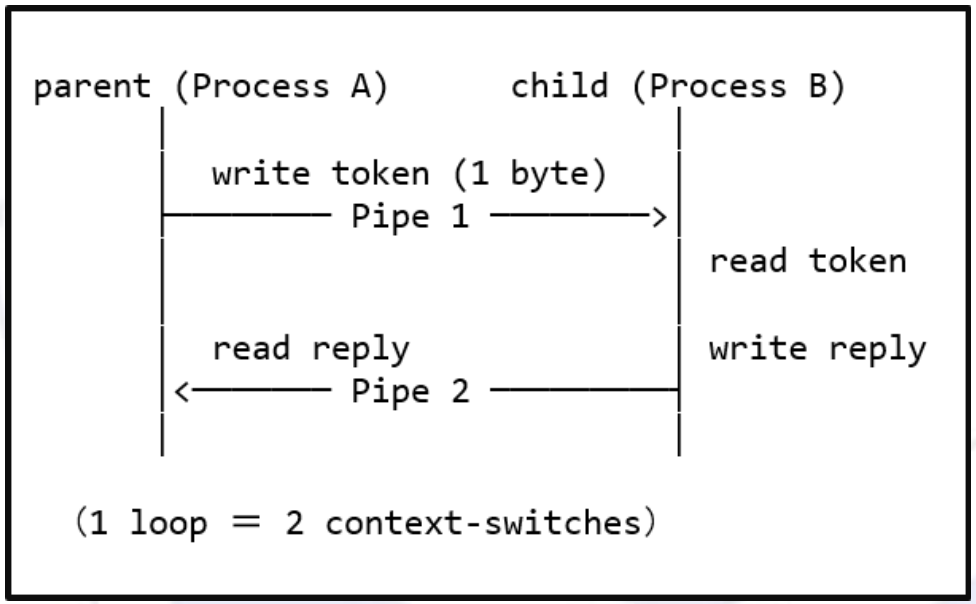
Why UnixBench Context1?

UnixBench is a widely used benchmark suit for evaluating different aspects of system performance.



Today: Context1

- **Measures how efficiently the system handles process handoff. In other words: it measures the performance of the system's scheduler.**



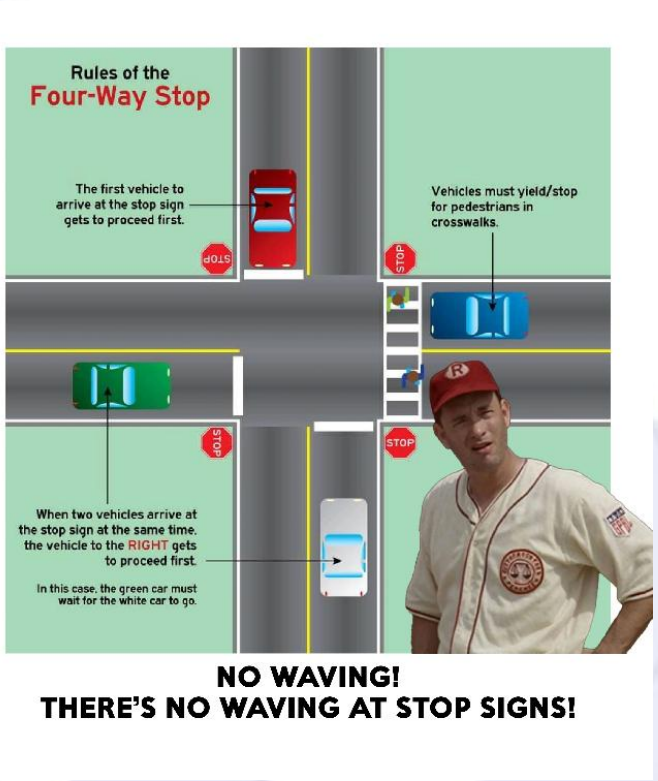
Scheduler performance measured by number of loops.

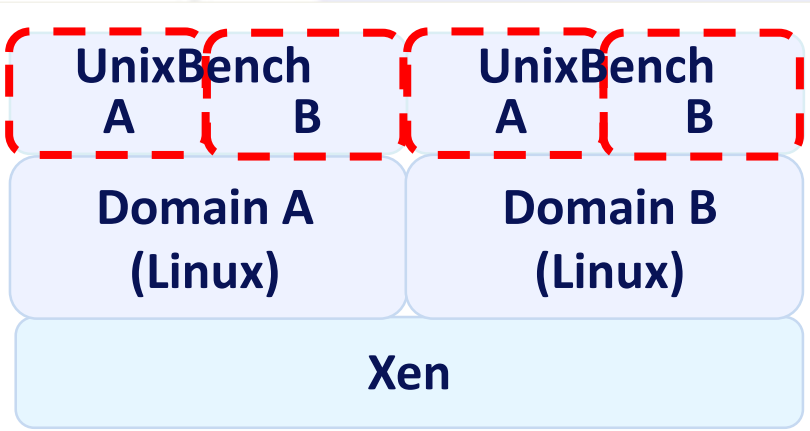
- **More loops = better score.**

Why UnixBench Context1?

Q: How can a token be exchanged between the processes more quickly?

- **Depends on UnixBench/Linux scheduler decision when to switch to the other**



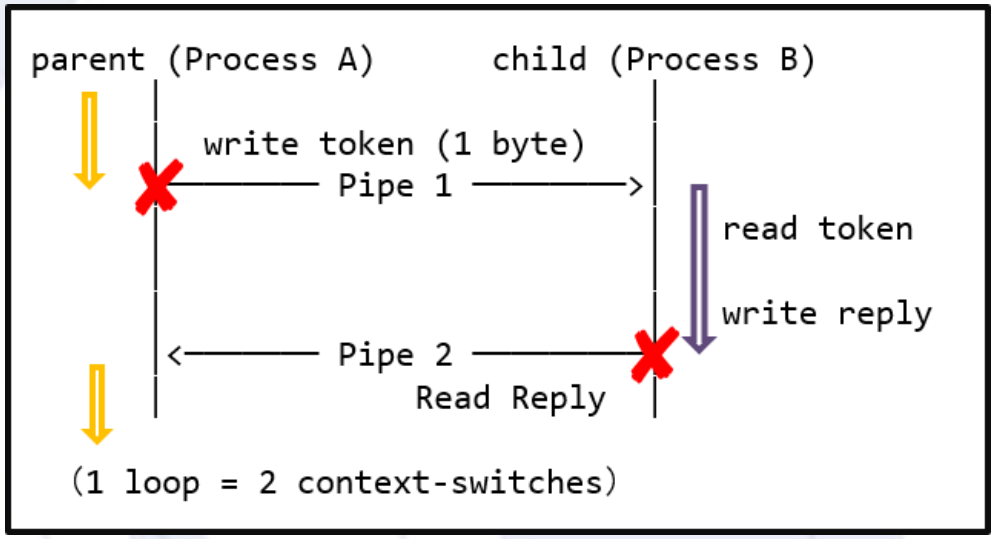


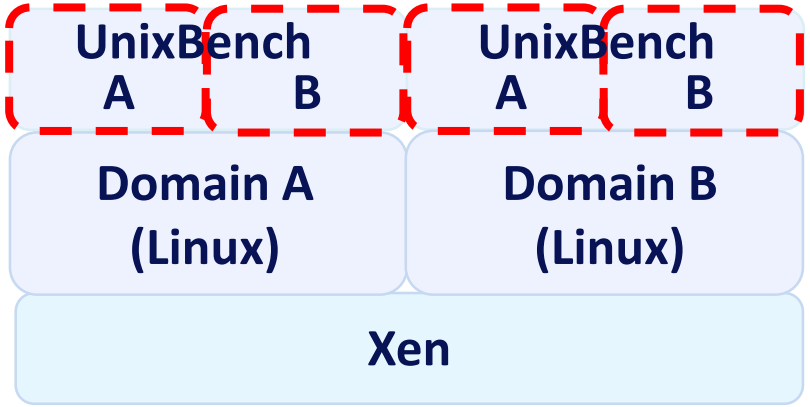
A: Let the scheduler make the decision, rather than UnixBench waving each other…

1ft view:UnixBench Context1

Recall UnixBench behavior…

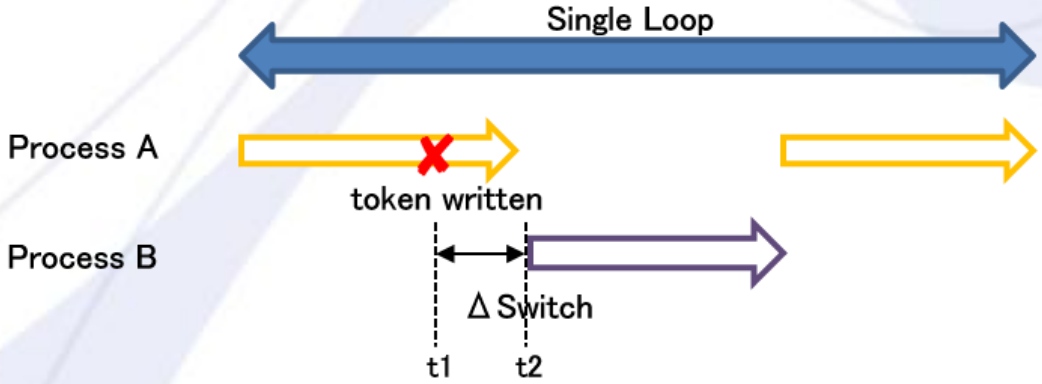
each process needs to switch to the other after passing its token.





In other words, we want to keep the Δ short.

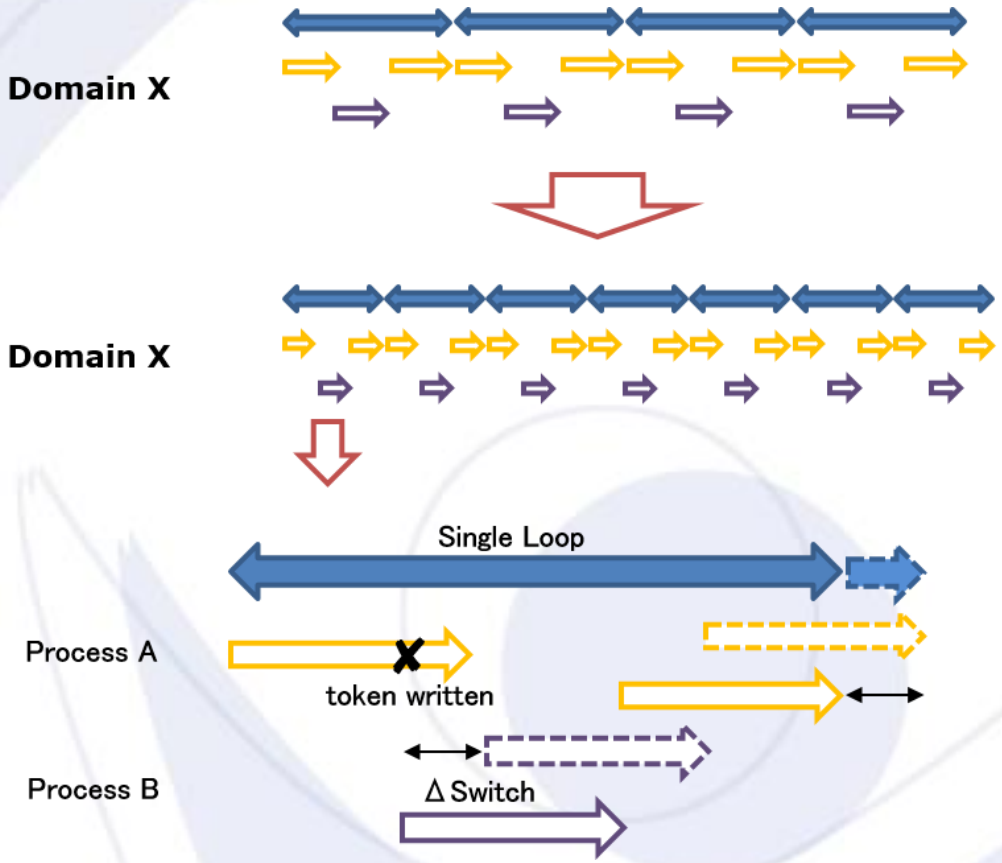
- **Controlled by Linux scheduler.**



10ft view: Linux CFS

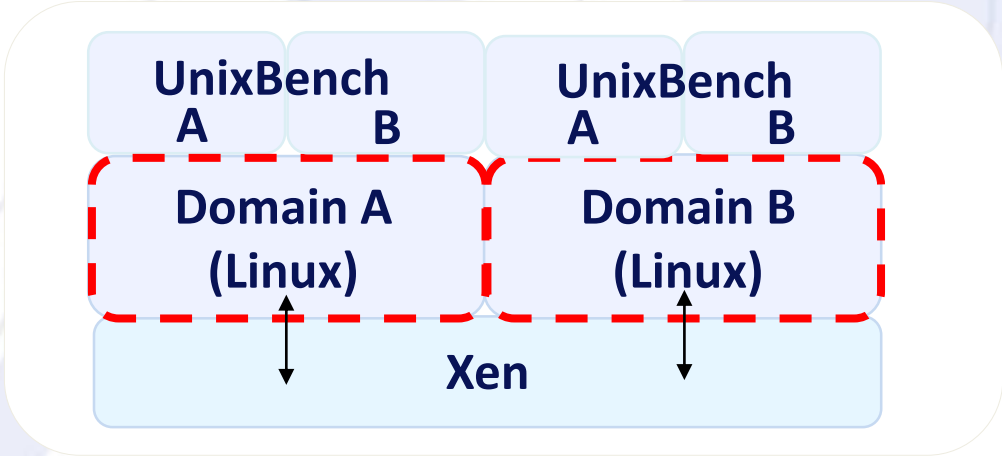
How does CFS decide who runs next?

- **CFS is a fair scheduler trying to balance processes' runtime**
  - **Greater runtime imbalance, more aggressive switching**



But where does the imbalance come from?

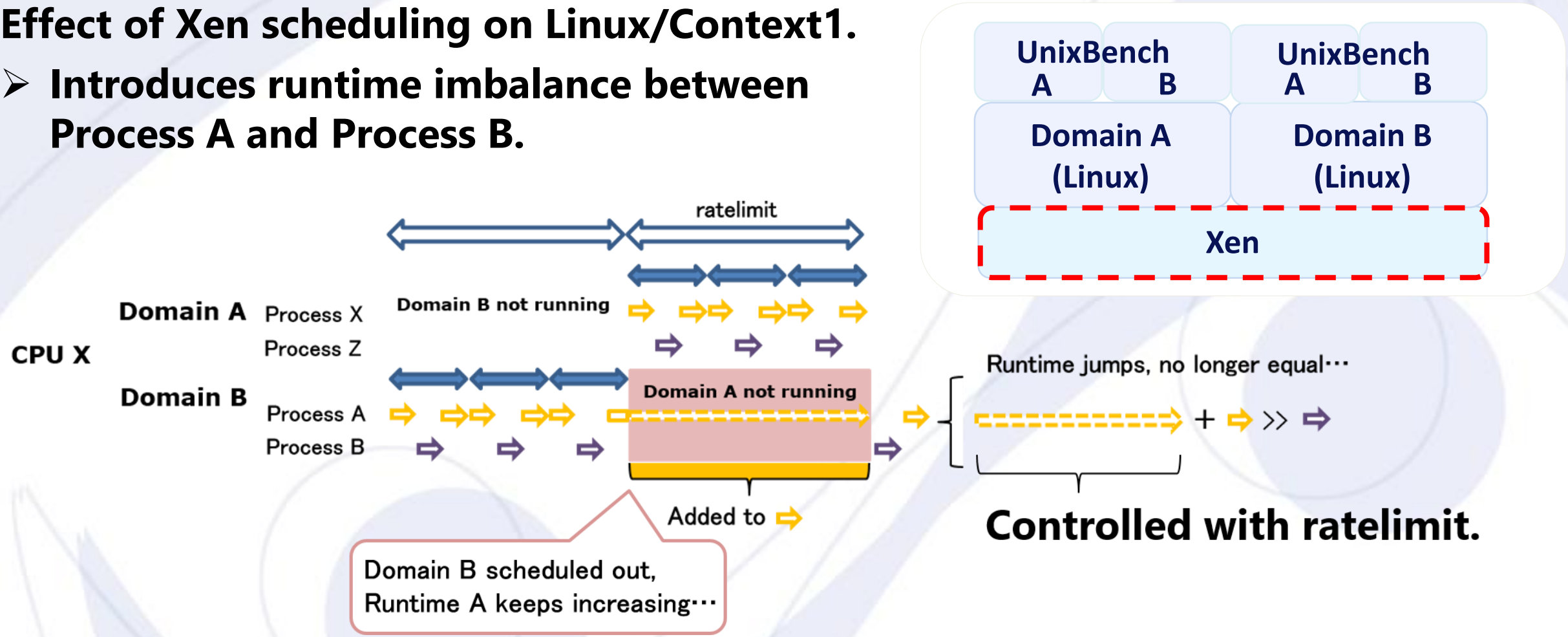
- **Interaction between Linux ↔ Xen**



100ft view:Xen Credit2

Effect of Xen scheduling on Linux/Context1.

- Introduces runtime imbalance between Process A and Process B.

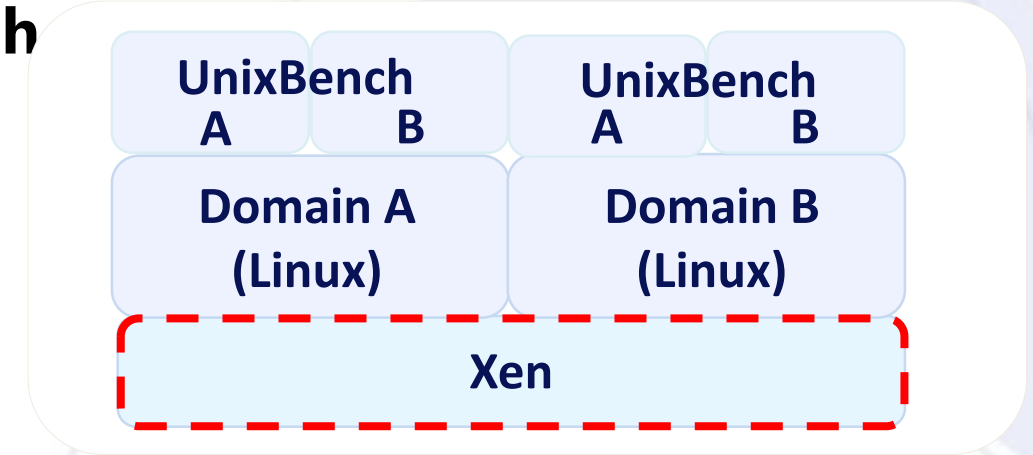


What is ratelimit?

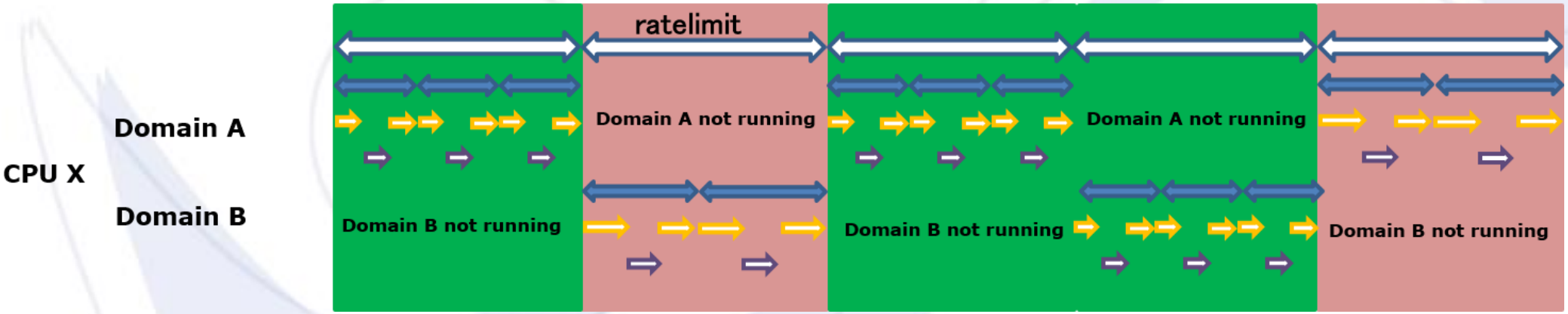
Parameter for controlling rate of context switch

- In other words: how long a domain is allowed to run continuously

```
$ sudo xl sched-credit2 -s
Cpupool Pool-0: ratelimit=1000us
$ sudo xl sched-credit2 -s -r 20000
$ sudo xl sched-credit2 -s
Cpupool Pool-0: ratelimit=20000us
```

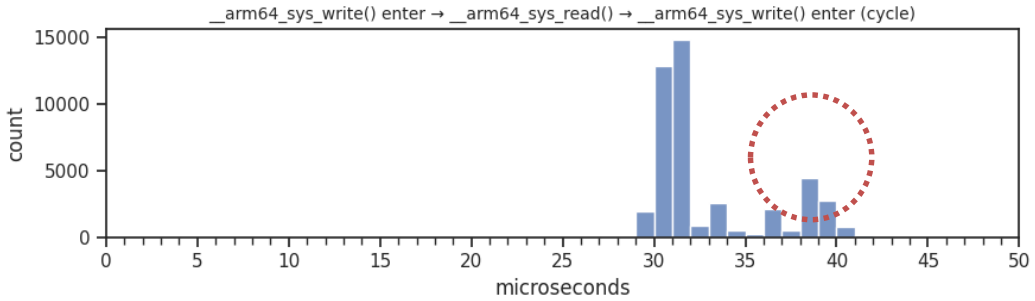


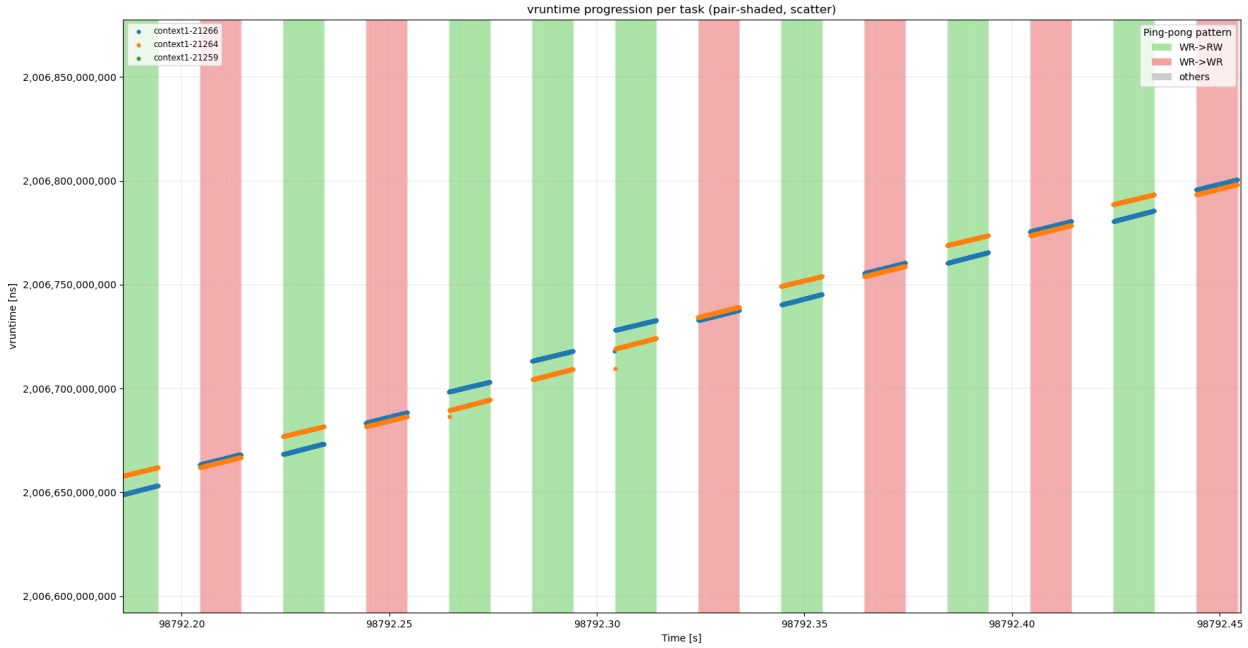
✓ What ratelimit will work the best?



Finding the best ratelimit

In the case of ratelimit = 5 ms.



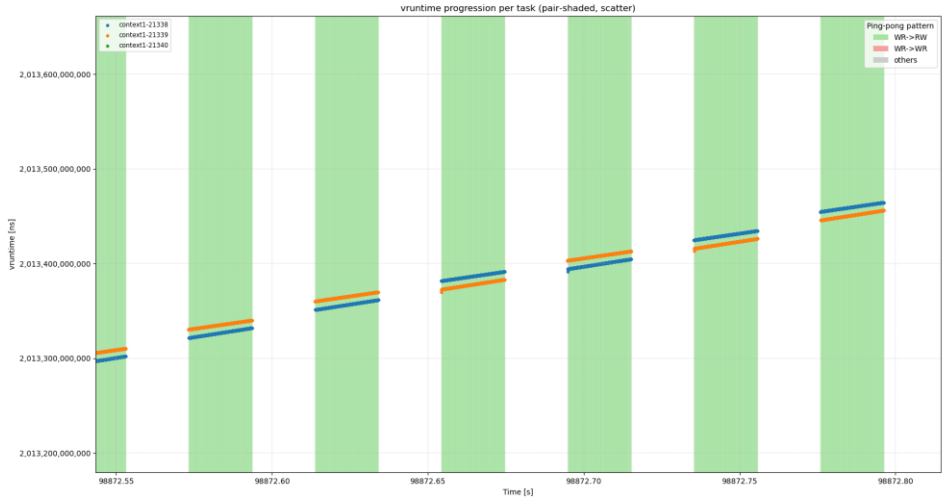


- ✓ **Let processes pass without being stopped — with the right ratelimit.**

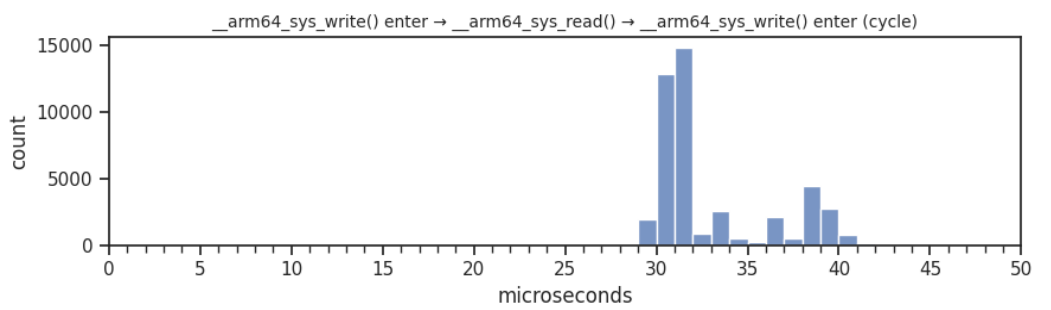
[Watch a Traffic Jedi Hit 240 Consecutive Green Lights in Manhattan]

Finding the best ratelimit

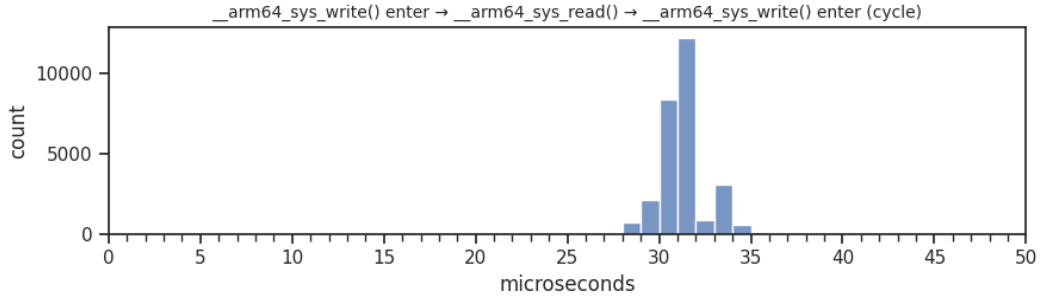
In this setup, 20 ms worked best.



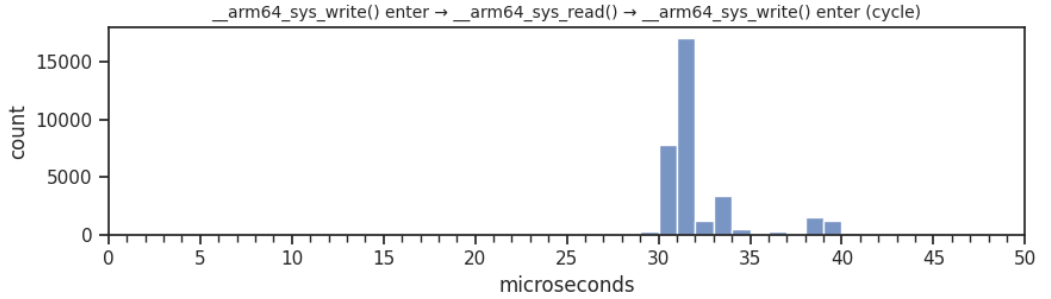
-r 1ms



-r 20ms



-r 128ms



- **The value and behavior are supported by source-code analysis. See Appendix.**

“Why Open Source?”

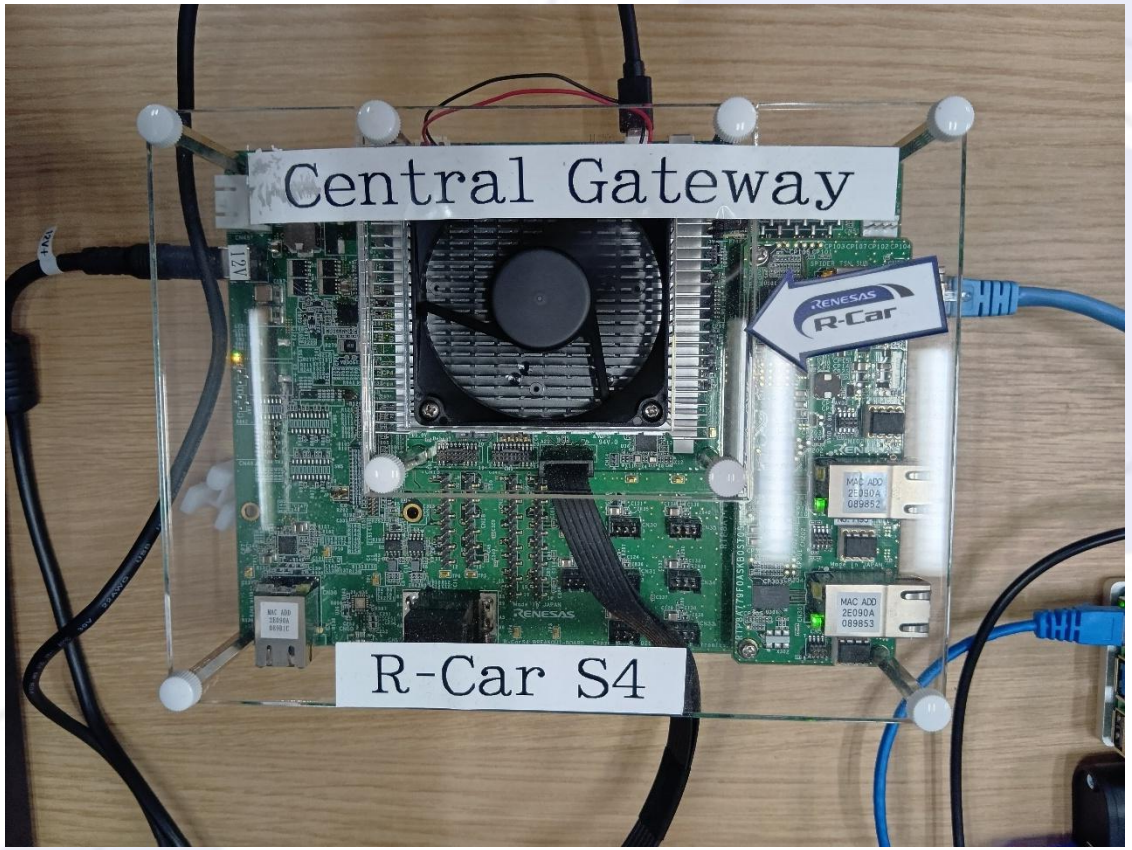
- ✓ **Able to analyze the workload (UnixBench)**
- ✓ **Able to analyze Linux scheduler**
- ✓ **Able to analyze Xen scheduler**

- ➢ **Understanding how each behaves leads to …**
  - **Better scores**
    - ➢ **because tuning was guided by workload behavior**
  - **Predictable behavior**
    - ➢ **because Linux and Xen are now aligned to work together**
  - **Reliable performance**
    - ➢ **because the result was understood and repeatable**

Appendix

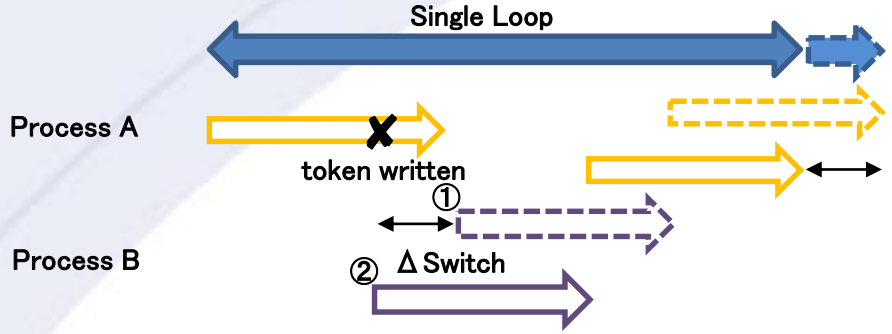
Setup

- Board: Renesas R-Car S4 Development Board / Spider
- SoC: R-Car S4 (r8a779f0), 8 × Cortex-A55 @ 1.2 GHz
- Xen: v4.20
- Linux: v5.10
- Schedulers: Xen Credit2, Linux CFS
- Workload: UnixBench Context1



UnixBench Context1's Scheduling points

Where does the scheduling happen for switch ① and ②?




```
[UnixBench/src/context1.c]
 67                  /* master, write p1 & read p2 */
   68                  close(p1[0]); close(p2[1]);
   69                  while (1) {
   70                          if ((ret = write(p1[1], (char *)&iter, sizeof(iter)))
   71                                          != sizeof(iter)) { // ② write(2) preemption (“Fast Pattern”)
[...]
   78                                  exit(1);
   79                          }
   80                          if ((ret = read(p2[0], (char *)&check, sizeof(check)))
   81                                          != sizeof(check)) { // ① read(2) blocking (“Slow Pattern”)
[...]
   88                                  exit(1);
   89                          }
```

Scheduling condition for write(2)

Under what condition does write(2) trigger scheduling by preemption?

- **On write(2) wakeup path, wakeup_preempt_entity() checks the vruntime gap between the currently running task("curr") vs wakeup target("se"). If the currently running task is ahead by more than >3ms of runtime(★), the wakeup target will be scheduled.**

```
[kernel/sched/fair.c] # v5.10
6900 static int
6901 wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
6902 {
6903         s64 gran, vdiff = curr->vruntime - se->vruntime;
6904
6905         if (vdiff <= 0)
6906                 return -1;
6907
6908         gran = wakeup_gran(se); # ★sysctl_sched_wakeup_granularity. Defualting to 1ms*(1+ilog2(ncpus))=3ms
6909         if (vdiff > gran)
6910                 return 1;
6911
6912         return 0;
6913 }
```

Runtime adjustment on read(2)

When a process wakes up from a blocking event, e.g. read(2), CFS may adjust its vruntime by -9ms(★).

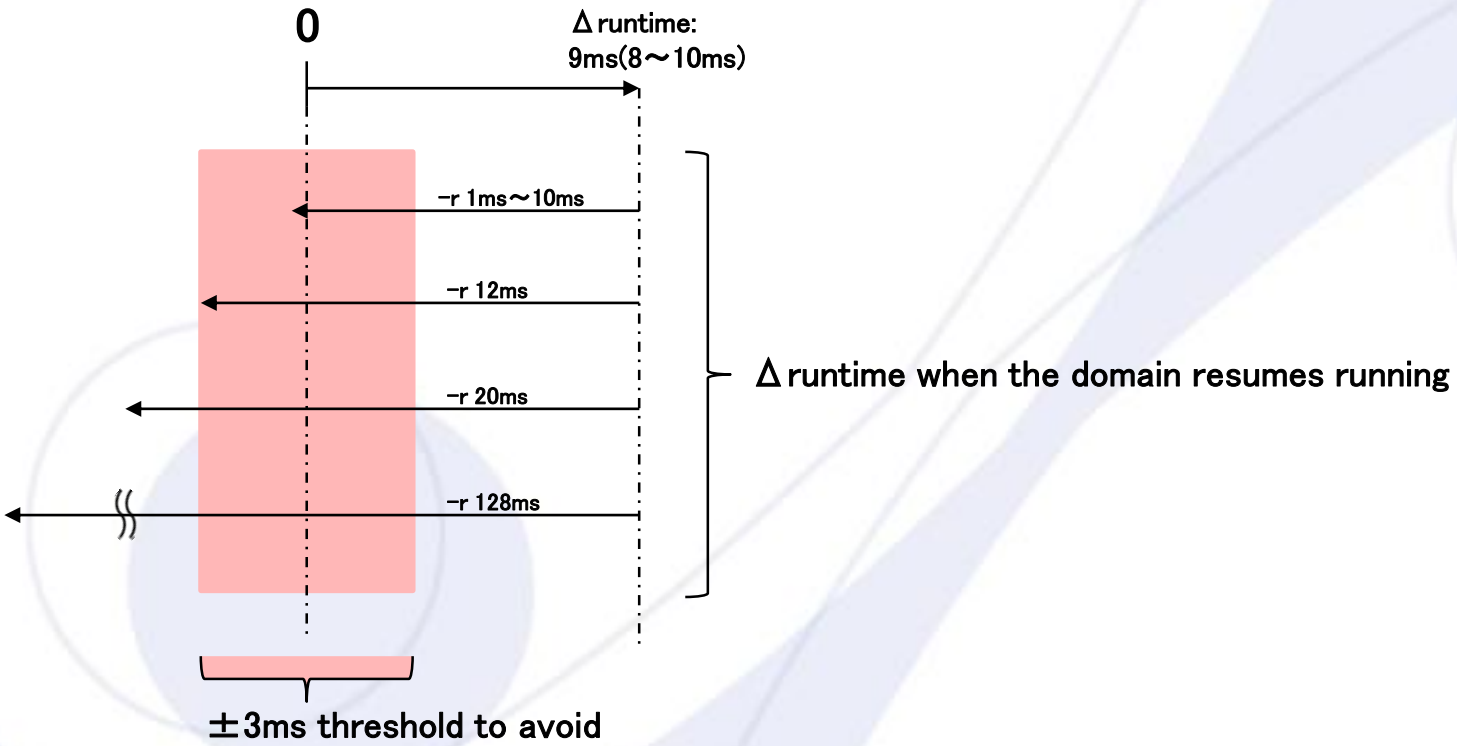
- ✓ **TheΔruntime after adjustment should not fall into the ±3ms threshold so that wakeup_preempt_entity() can continue to trigger scheduling on write(2) path (see slide 18).**

```
[kernel/sched/fair.c] # v5.10
4137 static void
4138 place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
4139 {
4140         u64 vruntime = cfs_rq->min_vruntime; # stealされた側の
[...]
4152         if (!initial) {
4153                 unsigned long thresh = sysctl_sched_latency;  # Defaulting to 6ms*(1+ilog2(ncpus))=18ms
4154
4155                 /*
4156                  * Halve their sleep time's effect, to allow
4157                  * for a gentler effect of sleepers:
4158                  */
4159                 if (sched_feat(GENTLE_FAIR_SLEEPERS))
4160                         thresh >>= 1;
4161
4162                 vruntime -= thresh; # ★thresh=9ms
```

Why ratelimit=20ms works the best?(1)

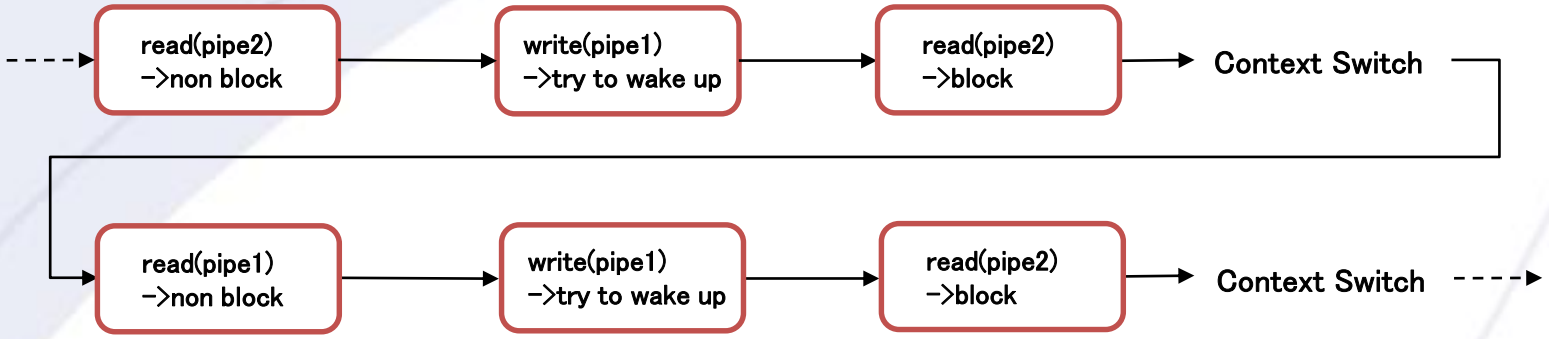
Effect of CONFIG_PARAVIRT_TIME_ACCOUNTING=n.

- **When the process wakes up, runtime accumulated while the other guest was running becomes visible to the guest scheduler.**
  - **Ratelimit should be set so that the adjusted runtime(see slide 19) stays outside the ±3ms threshold for scheduling on the write(2) path(see slide 18).**

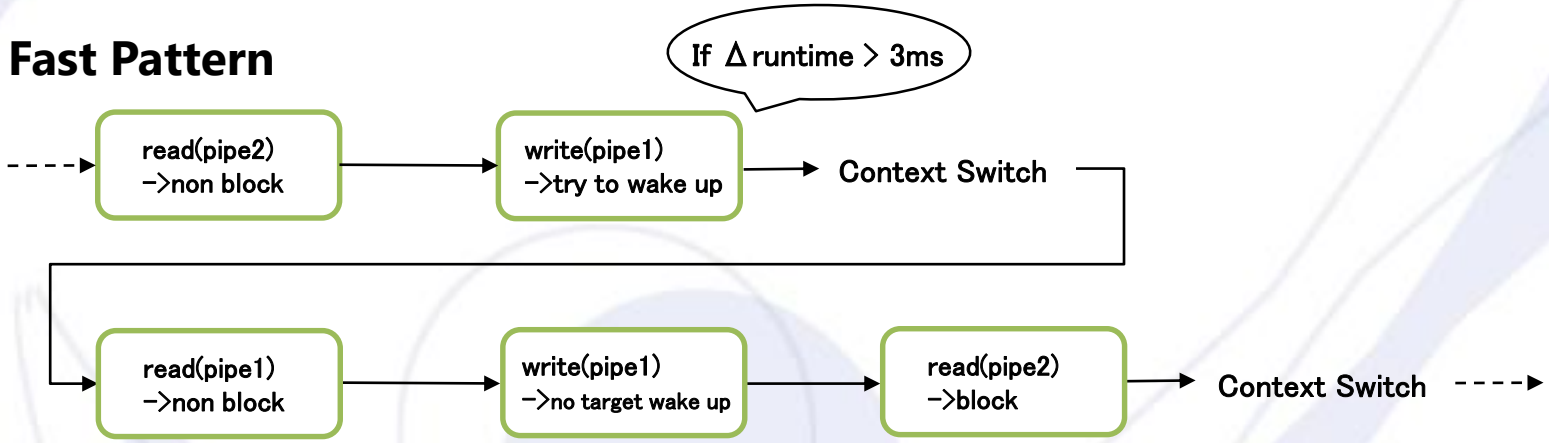


Why ratelimit=20ms works the best?(2)

Slow Pattern



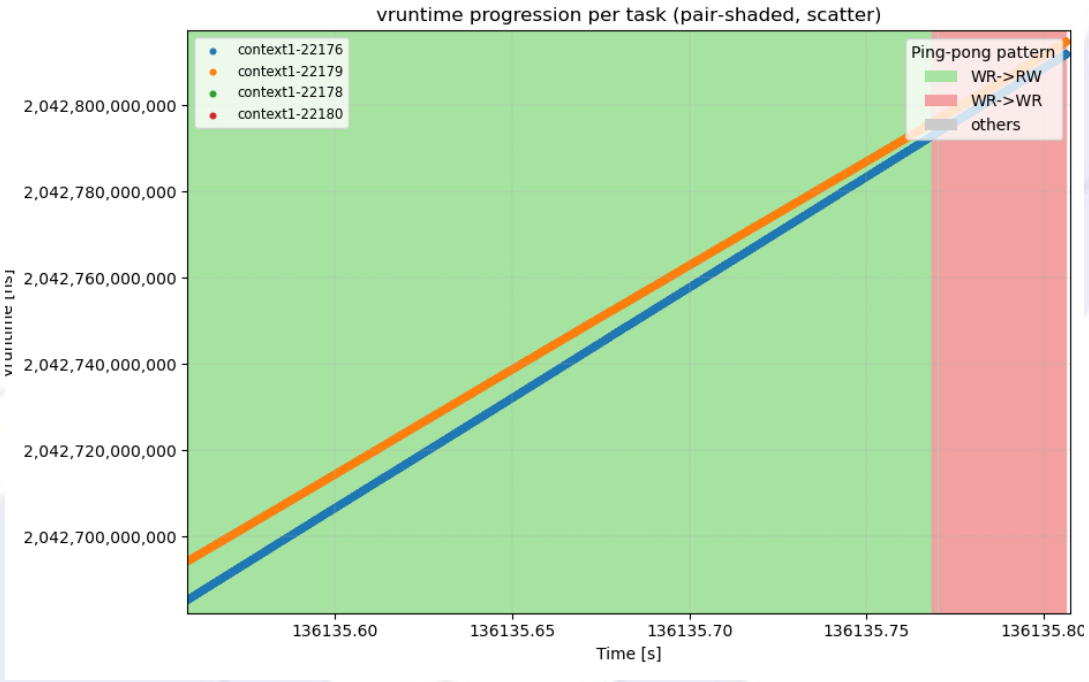
Fast Pattern



Why not set a large ratelimit?

From our measurement, Δruntime shrank by about 28us for every 1ms.

- **A larger ratelimit increases the chance of falling into ±3ms threshold as time elapses※**
  - ✓ **We want a ratelimit long enough to avoid the ±3ms threshold(see slide 18) on wakeup(see slide 19), but short enough to minimize the chance of running out of Δruntime.**



※ takes about 214ms given Δruntime of 6ms(@thresh(=9ms,see slide 18) - @gran(=3ms,see slide 17))