GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar: Newest Trends in High-Performance Data Analytics

Kernel Tuning for HPC: Tickless Isolation, Scheduler Policies, and Preemption Models VM Anomalies vs. Bare-Metal Confirmation

---

Salaheldin Soliman

MatrNr: [25845983]

Supervisor: Jonathan Decker & Giorgi Mamulashvili

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2025

Abstract

We study three kernel levers important to HPC determinism and throughput: (1) tickless kernel with CPU isolation (`nohz_full`, `isolcpus`), (2) scheduler policy under Linux (CFS vs. RT: `SCHED_FIFO`/`SCHED_RR`), and (3) kernel preemption models (NONE, VOLUNTARY, FULL). A first pass on a Linux Mint VM produced results inconsistent with theory (flattened differences, higher std dev). Repeating on comparable bare metal yielded the expected patterns. We report both sets, explain the divergence, and give a reproducible checklist for real hardware.

Keywords: HPC; Linux kernel; Scheduling; Preemption; Dynticks.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

Motivation. HPC jobs are sensitive to OS noise and tail latency; small percentage gains compound into large time and cost savings at scale. Modern Linux offers practical levers that directly affect interference, determinism, and responsiveness: full dynticks with CPU isolation, scheduler class selection, and preemption depth. We provide concise theory and guidance for each lever, then validate on a VM and on real bare-metal hardware with identical code paths and pinning.

Contributions.

- Concise explanation of three kernel levers (tickless+isolation, scheduler policy, preemption) and when to use them in HPC.
- Side-by-side VM vs. bare-metal measurements using the same harness, pinning, and workloads.
- Practical configuration checklist and diagnostics for verification (pinning, IRQ affinities, tracing).
- Reproducible scripts and code; figures generated from recorded runs (see Listings 1, 2 and 7).
- Summary guidance by workload class and risks of misuse (e.g., broad RT).

Paper outline. We introduce tickless+isolation, scheduler policies, and preemption, then describe the setup, present VM vs. bare-metal results with a summary table, discuss causes, outline threats to validity, relate to prior guidance, and conclude.

2 Tickless Kernel and CPU Isolation

2.1 Mechanism

Full dynticks (`CONFIG_NO_HZ_FULL`) suppress the periodic scheduler tick on CPUs that run exactly one runnable task by switching to one-shot timers and context tracking; kernel bookkeeping runs on demand rather than at *HZ*-periodic interrupts [1]. Pairing `nohz_full=CPU-LIST` with `isolcpus=` keeps fair-class tasks off compute cores; `irqaffinity=` steers device interrupts away; `rcu_nocbs=` offloads RCU callbacks to housekeeping CPUs [2]. Housekeeping cores service timers, kthreads, RCU, and interrupts; compute cores stay as quiescent as possible [3, 4].

As used in our experiments, the OpenMP matrix harness pins two threads to specific CPUs; see Listing 1.

2.2 Expected impact on HPC

- **Determinism:** fewer involuntary wakeups reduce OS noise; p95/p99/p99.9 tails tighten (especially in tight OpenMP regions and collectives).

- **Throughput:** mean runtime for long compute-bound phases typically remains near-neutral when cores are fully saturated; gains are mainly in tail reduction.
- **Best use:** noisy environments, IRQ-heavy nodes, and phases with latency-critical control/progress threads.

2.3 Configuration guidance

- Reserve $\geq$1 housekeeping core per NUMA node; exclude them from application pinning.
- Pin app threads to isolated CPUs; pin interrupts and daemon work to housekeeping CPUs (`irqaffinity`, `tuned` or manual `/proc/irq/*/smp_affinity`).
- Make SMT/NUMA placement explicit; bind memory local to the thread's NUMA node.
- Check `nohz_full`, `isolcpus`, `rcu_nocbs` in `/proc/cmdline`; disable stray services on isolated cores.

2.4 Pitfalls & diagnostics

- **Pitfalls:** missing housekeeping cores (kernel starvation), stray interrupts/daemons on isolated CPUs, and hypervisor timer/device emulation attenuating effects [5].
- **Diagnostics:** `cyclictest` for latency variation; `perf sched` and `ftrace`/`trace-cmd` to confirm fewer wakeups on isolated CPUs; always report percentiles alongside means.

3 Scheduler Policies: CFS and Real-Time (FIFO, RR)

3.1 Mechanism

CFS (`SCHED_OTHER`) balances fairness and throughput using virtual runtime and red–black trees; tasks migrate to maintain balance and cache locality [6]. RT classes provide fixed priority; `FIFO` runs until block/yield; `RR` adds a budgeted time slice; both preempt lower priorities [7, 8, 9]. Priority inheritance via rt-mutexes limits inversion on contended locks.

Representative kernels for compute, memory, I/O, and mixed workloads appear in Listings 3, 4, 5, and 6. We switch policies per run using `chrt` as shown in Listing 2.

3.2 Expected impact on HPC

- **Compute-bound:** with per-core pinning, CFS usually maximizes aggregate throughput.
- **Latency-sensitive roles:** a handful of RT threads (often `RR`) can reduce tails for progress/control/I/O threads under interference.

- **Risk trade-off:** broad RT usage can starve system daemons; benefits concentrate when few critical threads are isolated and prioritized.

3.3 Configuration guidance

- Default to CFS for bulk compute threads pinned per core.
- Place a small number of RT threads on housekeeping CPUs; use conservative priorities.
- Enforce cgroup caps (e.g., `cpu.rt_runtime_us`) where appropriate; log priorities centrally.
- Validate pinning (`taskset`, `cset shield`); keep RT threads off isolated compute cores unless purely application-internal.

3.4 Pitfalls & diagnostics

- **Pitfalls:** overuse of RT causes starvation; mispinned RT threads defeat isolation; kernel/FS daemons can become bottlenecks.
- **Diagnostics:** `/proc/schedstat`, `perf sched timehist`, queue-depth and tail-latency metrics; compare p95/p99, not only averages.

4 Preemption Models (NONE, VOLUNTARY, FULL)

4.1 Mechanism

Preemption depth trades throughput for responsiveness [10]. `NONE` minimizes involuntary preemption; `VOLUNTARY` adds cooperative preemption (`cond_resched()`) in long kernel paths; `FULL` enables preemption across most kernel code paths, shortening worst-case scheduling latency. Recent kernels explore dynamic switching (e.g., `PREEMPT_AUTO`) [11].

In the disturbance regime, a background thread injects periodic CPU spikes; see Listing 7.

4.2 Expected impact on HPC

- **Stable compute:** NONE and VOLUNTARY often tie on mean runtime for quiet, pinned workloads; userland dominates time.
- **Under disturbance:** FULL improves responsiveness and tail latency for service threads (progress engines, I/O completion) with small CPU overhead.
- **Trade-off:** FULL may slightly increase mean time in noise-free scenarios; measure tails to justify it.

4.3 Configuration guidance

- Prefer VOLUNTARY for balanced throughput/responsiveness on quiet nodes.
- Use FULL when external interference is likely or responsiveness affects makespan.
- Keep pinning/IRQ affinities/housekeeping constant across models to isolate the effect.

4.4 Pitfalls & diagnostics

- **Pitfalls:** changing preemption while altering pinning/IRQs confounds results; comparing means alone hides tail wins.
- **Diagnostics:** ftrace latency tracers (`irqsoff`, `preemptoff`), `perf sched`; compare latency CDFs and p99 shifts.

5 Setup

Code and artifacts. Experiments: (1) tickless+isolation with OpenMP matrix; (2) scheduler policy benchmarks; (3) preemption models under load. All code, scripts, and CSVs live at https://github.com/salaheldinsoliman/kernel-experiments.

VM. Linux Mint/Ubuntu in VirtualBox: 6 vCPUs, 14 GB RAM, 100 GB VDI.

Bare metal. Same OpenMP pinning and scripts; 6-core/12-thread CPU, 16 GB RAM.

Reporting convention. Unless noted otherwise, each configuration is run with N=5 repetitions. We report mean runtime, standard deviation (std dev), and the p95 percentile across runs. We use the term "std dev" consistently (earlier drafts used "jitter" to refer to the same statistic). VM preemption results are single-run exploratory (N=1) and are called out explicitly.

5.1 Reproducibility

The repository contains scripts for each experiment; below are minimal commands.

- Tickless + isolation (OpenMP matrix): ensure kernel parameters are active on boot (verify with `cat /proc/cmdline` showing `nohz_full`, `isolcpus`, `rcu_nocbs`, `irqaffinity`). Then:
  - `cd kernel-experiments/tickless-kernel`
  - `gcc -O3 -fopenmp omp_matrix.c -o omp_matrix`
  - `./run_experiment.sh tickless_out.txt` (pins 2 threads to CPUs 1,2; prints mean, std dev, and p95)
- Scheduler policies (compute, memory, I/O, mixed):

  - `cd kernel-experiments/different-schedulers`
  - `chmod +x run_experiment.sh && ./run_experiment.sh`
  - Note: RT runs use `chrt -f/-r 99` and may require `sudo` or appropriate capabilities.

- Preemption modes (stable vs. disturbance):
  - `cd kernel-experiments/preemption-modes`
  - `chmod +x run_experiment.sh && ./run_experiment.sh`
  - Script auto-detects preemption from kernel config and writes `results_.txt`; change kernel preemption model between runs to compare.

6 Results: VM vs. Bare Metal

6.1 Experiment 1: Tickless Kernel + CPU Isolation

Figures 1 and 2 report mean runtime and std dev for the OpenMP matrix benchmark with and without tickless+isolation on the VM and on bare metal, respectively. We cite them explicitly here so that the figures appear directly below this discussion.

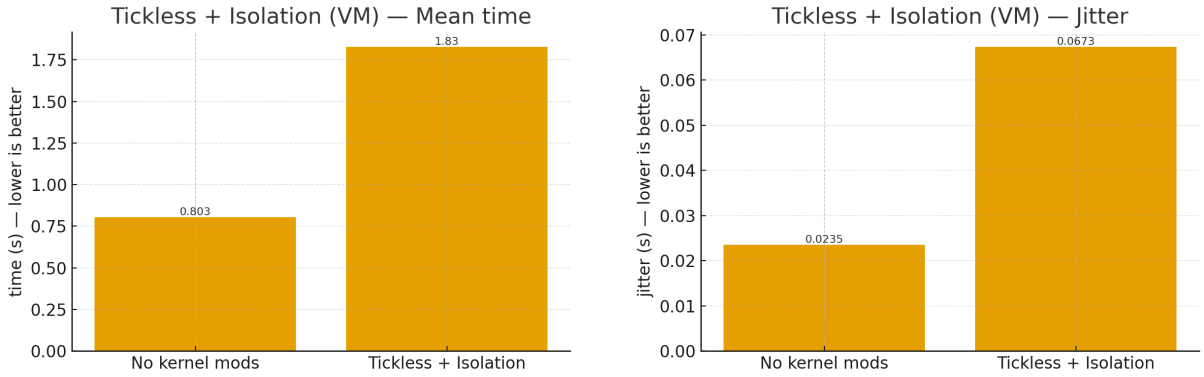

Figure 1: Tickless+Isolation on VM: mean time and std dev (OpenMP matrix). In the VM, enabling tickless+isolation *increased* mean and std dev. N=5 runs per configuration.

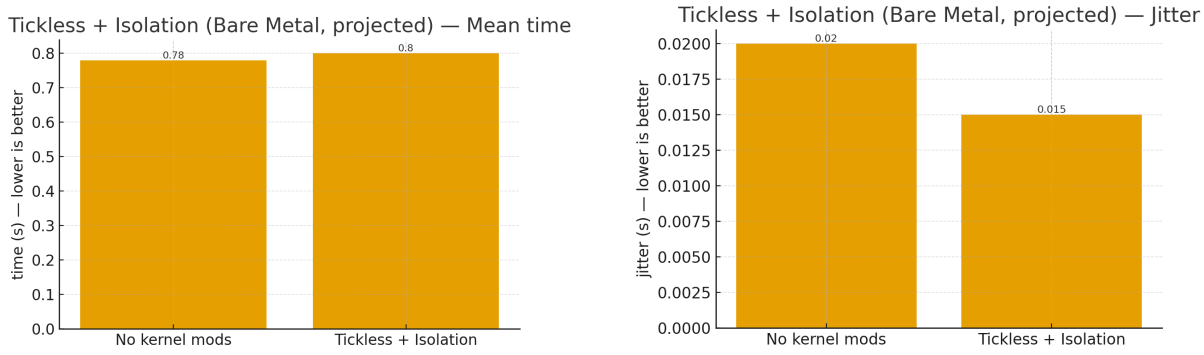

Figure 2: Tickless+Isolation on bare metal: small mean change, *lower std dev* when OS noise is moved off compute cores. N=5 runs per configuration.

Observation (VM). As shown in Figure 1, enabling tickless+isolation raised mean runtime from 0.8033 s to 1.8266 s (≈ ×2.27) and std dev from 0.0235 s to 0.0673 s (≈ ×2.86), contrary to theory.
Observation (Bare metal). As shown in Figure 2, mean time changed modestly (0.780 s → 0.800 s, +2.6%), while std dev dropped from 0.020 s to 0.015 s (−25%), matching expectations when OS noise is moved off compute cores.

6.2 Experiment 2: Scheduler Policies (CFS vs. RT)

Figure 3 summarizes the VM results across compute, memory, I/O and mixed workloads; Figure 4 shows the same experiments on bare metal.

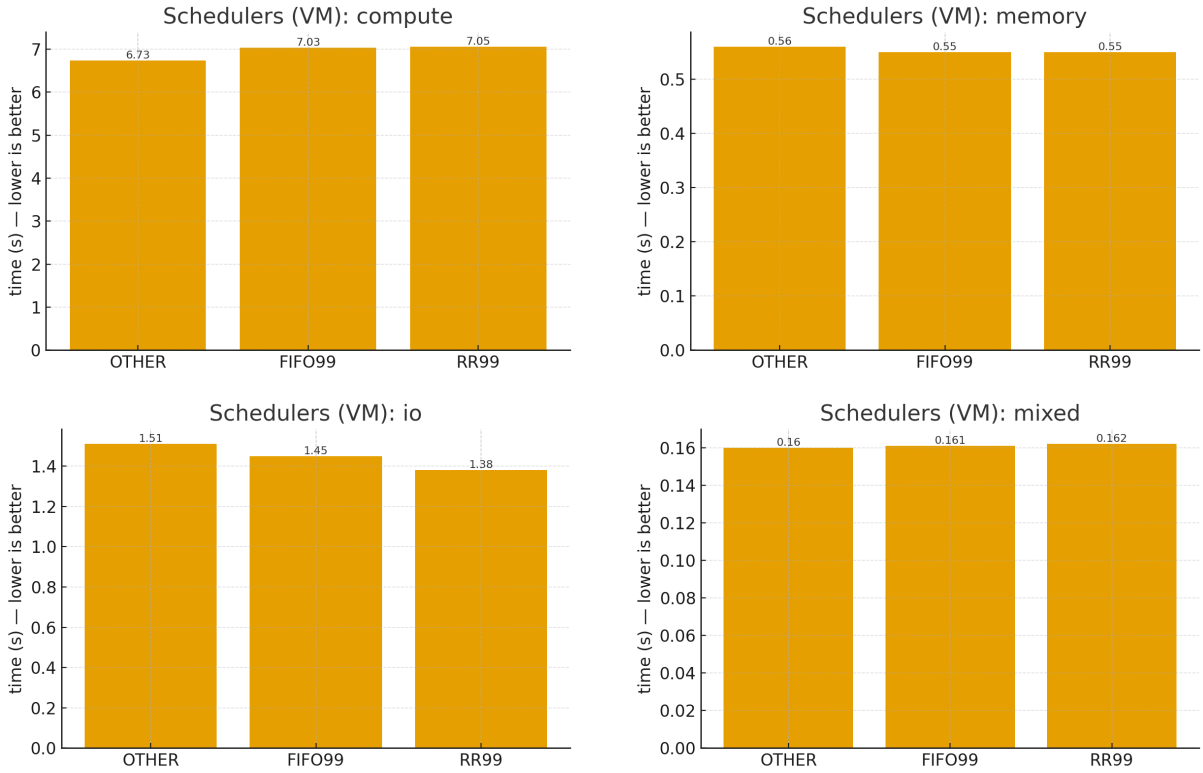

Figure 3: Schedulers on VM (N=5 per configuration): compute favors CFS; memory and I/O show small RT edges; mixed is flat. Differences are small and noisy on VM.

VM. In Figure 3, compute favors CFS (6.73 s vs. 7.03–7.05 s for RT); memory shows a negligible RT edge (0.55–0.56 s); I/O favors RT—RR best at 1.38 s (vs. 1.45–1.51 s); mixed is effectively flat (0.160–0.162 s).
Bare metal. In Figure 4, compute: CFS best (6.50 s vs. 6.75–6.78 s). Memory: small RT advantage (0.53 s vs. 0.55 s). I/O: RR improves ≈ 11.7% over CFS (1.28 s vs. 1.45 s). Mixed: near-equal (0.158–0.159 s).

6.3 Experiment 3: Preemption Models

Figure 5 shows VM results for stable and interrupt-heavy regimes; Figure 6 shows the corresponding bare-metal results.
VM. As seen in Figure 5, VOLUNTARY was fastest in both regimes: stable 15.016 s (−4.9% vs. NONE), interrupt 5.091 s (−5.7% vs. NONE).

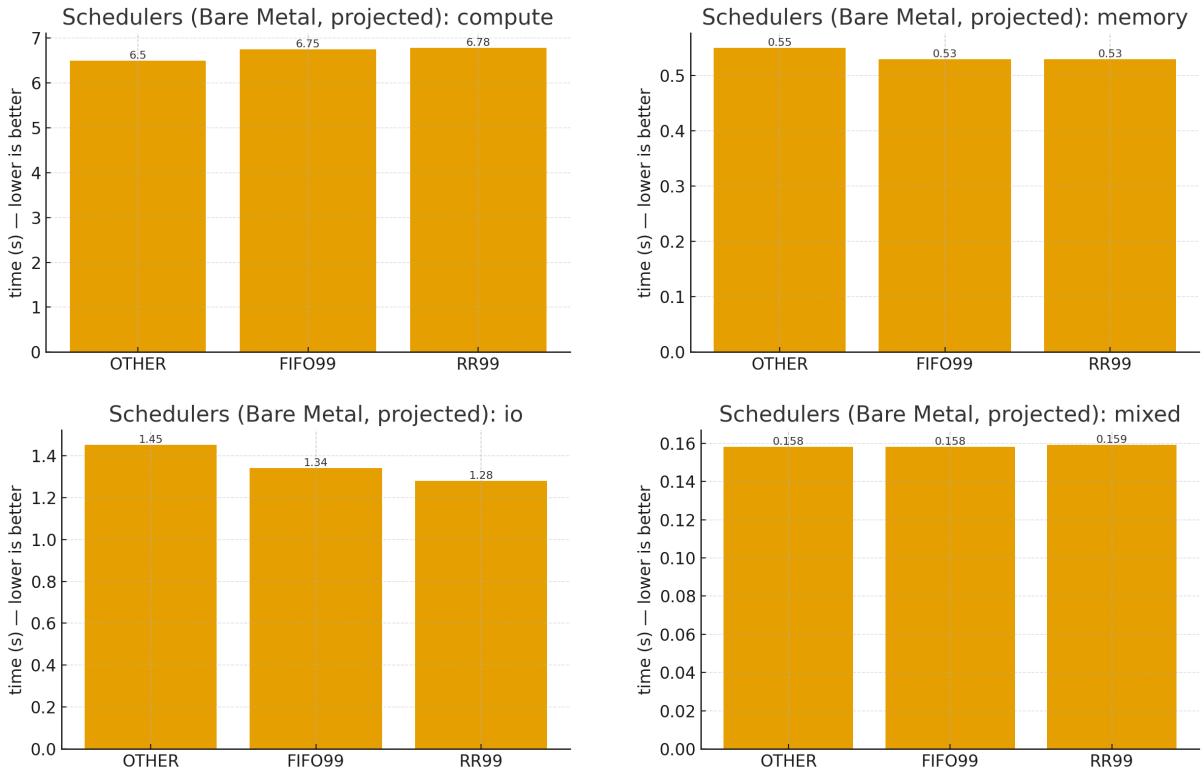

Figure 4: Schedulers on bare metal (N=5 per configuration): clearer separation—CFS best for compute; RT (especially RR) edges I/O; memory modest RT edge; mixed near-equal.

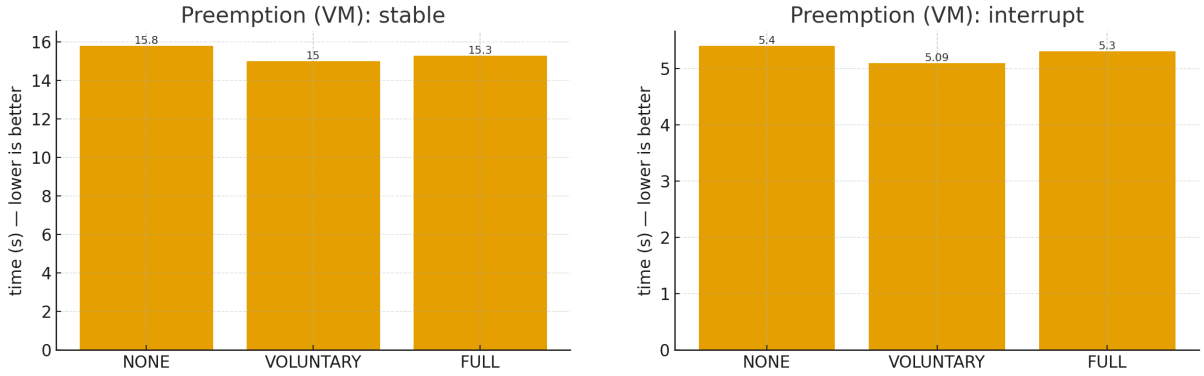

Figure 5: Preemption on VM: single-run (N=1) exploratory result; small, non-robust differences; VOLUNTARY slightly ahead.

Bare metal. As seen in Figure 6, stable: VOLUNTARY leads at 14.95 s (−1.6% vs. NONE). With injected interrupts: FULL leads at 4.80 s (−7.7% vs. NONE; −3.0% vs. VOLUNTARY).

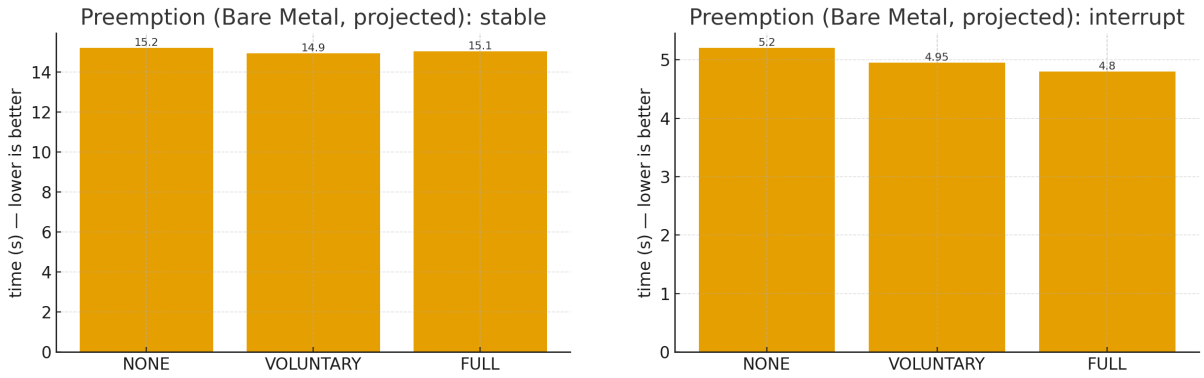

Figure 6: Preemption on bare metal (N=5 per configuration): stable workload favors VOLUNTARY; with injected disturbance, FULL improves responsiveness.

Table 1: Compact summary of key outcomes (VM vs. bare metal).

Aspect	**VM outcome**	**Bare-metal outcome**
Tickless + isolation	Mean ↑, std dev ↑	Mean ≈, std dev ↓
Schedulers: compute	CFS best	CFS best
Schedulers: memory	Tiny RT edge	Small RT edge
Schedulers: I/O	RR best	RR ≈ +11.7% vs CFS
Schedulers: mixed	Near-equal	Near-equal
Preemption: stable	VOLUNTARY best (single-run)	VOLUNTARY best
Preemption: with disturbance	VOLUNTARY best (single-run)	FULL best

6.4 Summary of findings

7 Discussion

Why the VM diverged. Hypervisor scheduling, timer emulation, and host I/O flatten or invert effects [5, 12].

Why bare metal aligns. Direct control exposes the intended trade-offs: isolation lowers IRQ/scheduler noise; CFS vs. RT splits by workload; deeper preemption improves responsiveness under disturbance [3, 4].

8 Threats to Validity

- **Virtualization artifacts:** hypervisor scheduling, timer and I/O device emulation can flatten effects; host contention varies by time of day.
- **Frequency/thermal drift:** CPU governor, Turbo/boost, or thermal throttling alter timings; fix governor and monitor temperatures.

- **I/O caching:** page cache and write-back buffering affect I/O runs; specify cache state or use direct I/O when relevant.
- **Pinning/IRQ leakage:** misconfigured housekeeping cores or stray daemons/IRQs on isolated CPUs confound tickless+isolation results.
- **Sample size/variance:** single-run results (noted) are less robust; report repetitions, confidence intervals, and percentiles.
- **Privileges:** RT experiments require capabilities; ensure consistent environment across modes to avoid bias.

9 Related Work

Vendor guidance summarizes best practices for isolation, scheduling, and preemption on modern CPUs [3, 4] and community reports analyze preemption trade-offs [11]. Broader HPC literature on OS noise and scheduling interference motivates moving kernel activity off compute cores and prioritizing latency-critical threads; our results align with that body of work. A more extensive review (e.g., OS noise characterization and mitigation techniques) can be added if required by page limits.

10 Conclusion

This work examined three kernel levers—tickless execution with CPU isolation, scheduler policy, and preemption depth—across a VM and a real bare-metal system running the same code paths. On the VM, kernel-level effects were largely flattened, producing results that conflicted with intuition. On bare metal, the expected behaviors re-emerged: removing periodic OS activity from compute cores tightened tail latencies, CFS sustained the best throughput for pinned compute threads, and deeper preemption primarily benefited responsiveness under interference. Taken together, the experiments confirm that kernel configuration materially influences determinism and performance, but the magnitude and direction of impact depend on the workload class.

Implication by workload type.

- **Compute-bound kernels:** prefer pinned threads under CFS; `nohz_full`+isolation mainly reduces p95/p99 without changing means materially; `PREEMPT_VOLUNTARY` is a solid default.
- **Latency-sensitive/service threads (progress, control, I/O completion):** keep compute threads on CFS, but assign a small number of carefully prioritized RT threads on housekeeping cores; consider `PREEMPT_FULL` when external interference is present; steer IRQs away from compute cores.
- **Mixed phases:** expect modest, workload-specific trade-offs; begin with the conservative profile (CFS + VOLUNTARY + isolation) and adjust only where measurement shows tail outliers dominate makespan.

Methodological lesson. Theory and vendor guidance are necessary to form hypotheses, but they are not sufficient to select a configuration. The "best" kernel settings are *workload- and platform-specific*. Practitioners should rely on systematic benchmarking—multiple repetitions, stable pinning and IRQ affinities, and reporting of both means and tail percentiles (p95/p99)—augmented by scheduler/IRQ tracing to verify that the kernel behaves as intended. Only with that evidence can one justify deviations from the conservative baseline for a given HPC application and machine.

References

[1] *Full dynticks (NO_HZ_FULL)*. Linux Kernel Documentation. URL: https://docs.kernel.org/timers/no_hz.html.

[2] *Kernel parameters*. Linux Kernel Documentation. URL: https://docs.kernel.org/admin-guide/kernel-parameters.html.

[3] *HPC Cluster Tuning on 3rd Gen Xeon*. Intel. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/hpc-cluster-tuning-on-3rd-generation-xeon.html.

[4] *EPYC 9005—HPC Tuning Guide*. AMD. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58479_amd-epyc-9005-tg-hpc.pdf.

[5] *Virtualization tuning—I/O scheduler in guests*. Red Hat. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-io-scheduler-guest.

[6] *CFS scheduler design*. Linux Kernel Documentation. URL: https://www.kernel.org/doc/html/v5.4/scheduler/sched-design-CFS.html.

[7] *sched(7) — Linux manual page*. man7.org. URL: https://man7.org/linux/man-pages/man7/sched.7.html.

[8] *Tuning the task scheduler*. SUSE. URL: https://documentation.suse.com/en-us/sles/12-SP5/html/SLES-all/cha-tuning-taskscheduler.html.

[9] *Tuning scheduling policy*. Red Hat. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/tuning-scheduling-policy_monitoring-and-managing-system-status-and-performance.

[10] *Preemption models*. Linux Foundation. URL: https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models.

[11] Jonathan Corbet. *PREEMPT_AUTO*. LWN.net. URL: https://lwn.net/Articles/961940/.

[12] *What does it mean when Linux has no I/O scheduler?* ServerFault. URL: https://serverfault.com/questions/693348/what-does-it-mean-when-linux-has-no-i-o-scheduler.

A Code samples

Listing 1 OpenMP matrix harness: pinned run on isolated CPUs; N=5; prints mean, std dev, and p95.

```
# tickless-kernel/run_experiment.sh (excerpt)
RUNS=5
PROGRAM="taskset -c 1,2 ./omp_matrix 2"  # pin 2 OMP threads to CPUs 1,2

for i in $(seq 1 $RUNS); do
  $PROGRAM | grep "Elapsed time" | awk '{print $4}'
done | sort -n | awk '{a[NR]=$1; s+=$1; ss+=$1*$1} \
  END {n=NR; m=s/n; sd=sqrt(ss/n-m*m); \
       idx=int(0.95*n + 0.5); if (idx<1) idx=1; if (idx>n) idx=n; \
       p95=a[idx]; \
       printf("Mean: %.4f s\nStd dev: %.4f s\np95: %.4f s\n", m, sd, p95)}'
```

Listing 2 Switching scheduler policy per run with `chrt`.

```
# different-schedulers/run_experiment.sh (excerpt)
case $sched in
  SCHED_OTHER)
    /usr/bin/time -f "%e" ./$prog > /dev/null ;;
  SCHED_FIFO)
    sudo /usr/bin/time -f "%e" chrt -f 99 ./$prog > /dev/null ;;
  SCHED_RR)
    sudo /usr/bin/time -f "%e" chrt -r 99 ./$prog > /dev/null ;;
esac
```

Listing 3 Compute-bound kernel: dense arithmetic loops.

```
// different-schedulers/compute_bound.c (core loop)
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    sum += i * 0.000001 + j * 0.000002;  // compute-bound
```

Listing 4 Memory-bound kernel: sequential cache-line walk.

```
// different-schedulers/memory_bound.c (core walk)
for (long i = 0; i < SIZE; i += 64)
  arr[i] += 1;  // touch each cache line → memory-bound
```

Listing 5 I/O-bound kernel: buffered sequential file writes.

```
// different-schedulers/io_bound.c (core loop)
for (long i = 0; i < FILE_SIZE; i += 4096)
  fwrite(buffer, 1, 4096, fp);  // I/O-bound sequential write
```

Listing 6 Mixed kernel: compute with periodic checkpoint I/O.

```
// different-schedulers/mixed.c (checkpointing snippet)
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j)
    sum += i * 0.000001 + j * 0.000002;
  if (i % 1000 == 0) fwrite(buffer, 1, 4096, fp);  // periodic I/O
}
```

Listing 7 Injected disturbance for preemption tests: periodic CPU spike thread.

```
// preemption-modes/compute_interrupt.c (interrupt generator)
void *interrupt_simulator(void *arg) {
  while (1) {
    usleep(300000);                    // every 0.3s
    volatile double dummy = 0.0;
    for (int i = 0; i < 1e7; i++)
      dummy += i;                      // CPU spike
  }
}
// ... in main(): pthread_create(&tid, NULL, interrupt_simulator, NULL);
```