University of Modena e Reggio Emilia
Department of Physics, Informatics and Matematics

Master's degree in Computer Science

Interference Among AI-Based Components and Real-Time Tasks in Embedded Autonomous Systems

Supervisor:
Prof. Paolo Valente
Co-supervisor:
Prof. Marko Bertogna

Candidate:
Emanuele Sebastiano Reggiani

Accademic Year 2025-2026

Abstract

Autonomous systems such as robots combine AI-based components, like computer-vision networks, with real-time control tasks. To reduce cost, size, and power consumption, both increasingly run on the same embedded system-on-chip, with several CPU cores and a GPU. The two kinds of software then share caches, main memory, the power budget of the chip, and the operating system, so the activity of one can delay the other. For a real-time task, this interference lengthens the wake-up latency, the time between the instant the task should start and the instant it runs, and leaves less time before its deadline.

On these platforms, interference has mostly been studied with synthetic workloads that load the memory as heavily as possible. They show how much a task can suffer, but not whether real AI components cause interference of the same size, nor through which channels it reaches the task. Studies of real vision applications mostly measure the application's own performance.

This thesis takes a first step by measuring how an industrial computer-vision application affects the mean wake-up latency of a periodic real-time task on an NVIDIA Jetson Orin Nano. The application, tkHPick, performs instance segmentation and pose estimation for robotic manipulation. The reference is MEMINTERF, a synthetic memory stressor from HeSoC-mark. The real-time task is `cyclictest`, in two versions. The first only wakes up and goes back to sleep, so it shows the delays caused by the operating system and the platform. The second also reads, at every activation, as much new data as a high-speed vision system processes per frame.

With the first task, interference arrives mainly through the scheduler, the peripheral interrupts, and the over-current throttling of the board. Memory contention weighs little, because the accesses of the task stay in the private caches of its core. With the second, memory becomes a main channel: on an isolated core the mean latency grows about twice as much, by 39% with tkHPick and 52% with MEMINTERF. A real AI application can therefore disturb a real-time task almost as much as a synthetic stressor, even when the task has a CPU to itself.

Table of Contents

1 Introduction **7**

2 Background **9**

2.1 Real-time tasks and wake-up latency 9

2.2 Real-time Linux 10

2.3 Multicore scheduling and CPU isolation 11

2.4 Shared resources on heterogeneous SoCs 12

2.5 Performance monitoring and kernel tracing 13

3 Related work **15**

3.1 Characterizing memory interference 15

3.2 Evidence from computer-vision applications 16

3.3 CPU isolation and shared-memory protection 16

3.4 Linux latency and kernel activity 17

3.5 Positioning of this thesis 17

4 Experimental design and methodology **19**

4.1 Hardware and software configuration 19

4.1.1 NVIDIA Jetson Orin Nano 19

4.1.2 Operating system and power configuration 20

4.2 The victim and the interferents 20

4.2.1 The victim 20

4.2.2 The H^3 Vision victim 20

4.2.3 Synthetic memory workload 21

4.2.4 Industrial perception application 21

4.3 Experimental conditions and acquisition protocol 22

4.3.1 CPU placement and isolation 22

4.3.2 Repeated acquisitions and counter groups 24

4.4 Measured quantities and attribution scopes 25

4.4.1 Latency and processor activity 25

4.4.2 CPU-wide and process measurements 26

4.5 Statistical treatment and interpretation limits 26

4.5.1 Latency statistics 26

4.5.2 Counter normalization and interpretation 27

5 Measurement infrastructure and diagnostic tools 29

5.1 Automated experiment execution 29

5.1.1 Controller and acquisition worker 29

5.1.2 Workload lifecycle and placement 29

5.1.3 Acquisition records 30

5.2 The H^3 Vision victim: extending `cyclictest` 31

5.3 Performance counter acquisition 32

5.3.1 Event selection and counter passes 32

5.3.2 CPU, process, and thread measurements 33

5.3.3 Counter output and interrupt snapshots 34

5.4 Kernel profiling and execution tracing 34

5.4.1 Sampling profiles 34

5.4.2 Function tracing and execution context 35

5.5 Targeted scheduler measurements 35

5.5.1 Observing branch decisions 35

5.5.2 Tracing shared scheduler state 36

5.5.3 Direct measurement of instruction cost 37

5.6 Per-phase measurement of the wake-up 38

6 Results: a minimal real-time task 41

6.1 tkHPick increases the wake-up latency 42

6.2 With tkHPick, the victim executes more instructions 42

6.3 The kernel takes a longer path when the victim goes to sleep 43

6.3.1 The long path 44

6.3.2 Why the scheduler takes the long path 45

6.4 A cpuset partition removes the scheduler coupling 46

6.5 Peripheral interrupts keep CPU0 exposed 49

6.6 What remains on the isolated CPU3 49

6.6.1 Measuring the phases of a wake-up 50

6.6.2 What the increase is made of 51

6.6.3 Why the memory wait is small in absolute terms 52

6.6.4 Over-current throttling 53

6.7 A minimal task barely feels memory contention 54

7 Results: a task with the memory footprint 55

7.1 On the isolated CPU3, the H^3 Vision footprint amplifies the interference 55

7.2 The kernel does the same work but waits longer 56

7.3 The H^3 Vision work evicts the kernel from the private caches 58

7.4 tkHPick rivals MEMINTERF 59

8 Conclusions and future work 61

9 References 63

1. Introduction

Autonomous systems increasingly integrate components based on artificial intelligence. A robotic arm that picks objects from a bin, for example, must first recognize and locate them: computer-vision neural networks separate the individual objects in the image (*instance segmentation*) and estimate their position and orientation in space (*pose estimation*). The same system also runs periodic real-time tasks, such as the control of the arm, which are activated at regular intervals and must complete their work within a deadline. For reasons of cost, size, and power consumption, the two classes of software increasingly run on the same embedded platform: a heterogeneous system-on-chip, such as the NVIDIA Jetson Orin Nano used in this thesis, which combines several CPU cores and an integrated GPU. The tasks then share hardware resources, such as the caches, the interconnect, the main memory, and the power budget of the chip, as well as software resources, such as the state of the kernel and the interrupts. Activity on one core can therefore change the timing of a task on another, a phenomenon called interference. One of the quantities that interference affects in a periodic real-time task is the wake-up latency, the time between the instant at which the task should start again and the instant at which it actually resumes execution. This latency adds to the response time of every activation and consumes part of its deadline: with a period of one millisecond, tens or hundreds of microseconds are already a significant fraction of the available time.

Interference on NVIDIA Tegra SoCs has long been studied, but mostly with synthetic workloads [7, 6]. Purpose-built interferents, such as those of the HeSoC-mark suite, load the memory in a controlled and repeatable way to characterize the platform. This approach starts from the interferent: it measures how much a task can suffer when the memory is under maximum load, but not whether the components of a real system actually generate interference of that magnitude. Studies that use real vision applications focus mainly on the performance of the application itself, such as its throughput, execution time, or accuracy [41, 2]. To the best of our knowledge, no study has yet examined in depth how much the interference generated by a real computer-vision application affects the mean wake-up latency of a separate real-time task, and through which channels it arrives, whether the scheduler, the interrupts, the shared memory, or the power budget. Without such an analysis, it is not known whether the problem actually exists in a real system, nor how large it is compared with what a synthetic interferent produces.

This thesis takes the first step of such an analysis. The AI component is tkH-Pick, an industrial computer-vision application supplied by the company, which performs instance segmentation and pose estimation for robotic manipulation and runs its neural-network inference on the integrated GPU. As a reference, we compare it with MEMINTERF, a synthetic interferent from the HeSoC-mark suite [13] built to load the memory as much as possible. The real-time task is `cyclictest`, the tool commonly used to measure the wake-up latency on Linux [16], in two forms. The first is the minimal task, `cyclictest` as it is: it wakes up, reads the clock, records the latency, and goes back to sleep. This is the least that a periodic task must do, and it takes a negligible time, since each iteration executes a few hundred user-space instructions and touches very little data. The task is therefore not a source of interference, for itself

or for the other cores, and it has none of the internal mechanisms of a more complex task. This makes it suited to observing what happens around it, such as the delays introduced by the scheduler, bursts of interrupts, and slowdowns of the whole chip. The second form, the H^3 Vision victim, adds to `cyclictest` a section that, after every wake-up, reads about 1.3 MB and writes 2 kB of data that are not in the cache, as much as a high-speed vision system for robots [15] handles at every frame. This section takes the task outside the private caches of its core and shows whether the shared memory becomes a new source of interference. This first study analyses only the mean latency: the high percentiles and the maximum latency are measured and described, but their analysis is left to future work. Likewise, only the wake-up latency is measured, and the time the H^3 Vision victim takes to complete its own work is outside the scope of this thesis.

With the minimal task (Chapter 6), when the victim is bound to a CPU through CPU affinity only, both interferents raise the mean latency to about 1.5–1.6 times that of the baseline. With tkHPick the victim also executes more kernel instructions, because when it goes to sleep the scheduler takes a longer path, which couples the victim CPU to the load of the other cores. A cpuset partition removes this coupling, but CPU0 keeps receiving the peripheral interrupts. On the isolated CPU3 only a small increase remains, 0.4 to 0.6 μs. Because the task is minimal, more than 99.4% of its accesses are served by the private caches of the core, so waiting for memory accounts for a fraction of a microsecond. With tkHPick most of the rest is over-current throttling, which no partition of the CPUs can remove. With the H^3 Vision victim (Chapter 7), memory comes into play. On CPU0 with CPU affinity only, the scheduler and the interrupts probably hide the effect of its footprint, but on the isolated CPU3 the increase of the mean latency is about twice that of the minimal task: 52% with MEMINTERF and 39% with tkHPick. The work of the task fills the private caches with its own data and pushes out the instructions and data that the kernel needs at the next wake-up. The kernel then executes the same instructions as before but waits longer for memory, where it competes with the traffic of the interferents. Once the real-time task accesses memory beyond its private caches, a real interferent such as tkHPick perturbs it by an amount comparable to that of a synthetic interferent built to disturb as much as possible.

2. Background

This chapter introduces the concepts on which the rest of the thesis relies. We first recall the model of a periodic real-time task and the meaning of its wake-up latency, and then describe how Linux with `PREEMPT_RT` handles timers, interrupts, and real-time threads. We continue with multicore scheduling and the mechanisms Linux offers to restrict where tasks run, and with the hardware resources that the cores of a heterogeneous system-on-chip share. The last section presents the performance counters and kernel tracing tools used to observe this activity. The specific platform, workloads, and configurations are described in Chapter 4.

2.1 Real-time tasks and wake-up latency

A real-time system is one whose correctness depends not only on the results it produces but also on the time at which they are produced [5]. Its activities are commonly modeled as tasks. A periodic task is released every period T, and each release, or job, must complete within a relative deadline D, often equal to the period. In a hard real-time task a missed deadline is a failure, whereas a soft real-time task tolerates occasional misses at the cost of degraded service.

Between the nominal release of a job and the moment it starts executing, the operating system has to notice that the release time has arrived, make the task runnable, select it for execution, and switch to it. The resulting delay, called wake-up or scheduling latency [4], adds to the response time of every job and consumes part of its deadline, while its variation from one activation to the next appears as release jitter. For a task with a period of one millisecond, latencies of tens or hundreds of microseconds are already a significant fraction of the available time.

Schedulability analysis relies on upper bounds on these latencies and on execution times. A measurement, however, only observes the latencies that occurred during a finite run, so the largest observed value is a lower bound on the worst case, not a guarantee. Experimental studies therefore examine the whole latency distribution and in particular its tail: high percentiles such as P99.9 describe rare but long latencies that the mean hides, and each of them is an activation at risk of missing its deadline. On Linux, this latency is typically measured with `cyclictest`, which repeatedly sleeps until an absolute time and records how late its thread runs again [16].

On a multicore processor, tasks running on different cores are not independent. They share hardware resources, such as caches, interconnect, and main memory, and software state, such as the data structures of the operating-system kernel. Activity on one core can therefore change the timing of a task on another, a phenomenon referred to as interference [7]. It can travel through the state that the kernel shares among CPUs, such as that of the scheduler (Section 2.3), through the interrupts directed at a CPU (Section 2.2), through the memory hierarchy, and through the power budget of the chip (Section 2.4). This thesis studies how much interference a computer-vision application causes to a periodic task, and through which channels.

2.2 Real-time Linux

Scheduling classes. The Linux scheduler is organized into scheduling classes that are consulted in a fixed order of precedence: deadline tasks (`SCHED_DEADLINE`), real-time tasks (`SCHED_FIFO` and `SCHED_RR`), normal tasks (`SCHED_OTHER`, handled by the Completely Fair Scheduler), and finally the idle task [27]. Real-time threads have a static priority between 1 and 99, and a runnable real-time thread always preempts normal threads. A `SCHED_FIFO` thread keeps the CPU until it blocks, yields, or is preempted by a thread of higher priority. Every CPU has its own runqueue, with a separate queue for each class. When no task is runnable, the CPU executes the idle task, which can place the core in a low-power state; leaving that state again takes time, which adds to the latency of the next wake-up.

The `PREEMPT_RT` kernel. In a standard kernel, interrupt handlers and code protected by spinlocks cannot be preempted, and the length of these sections bounds how quickly a high-priority thread can start running. The `PREEMPT_RT` patch set reduces them [43]: most interrupt handlers run in kernel threads with their own scheduling priority, most spinlocks become sleeping locks with priority inheritance, and deferred interrupt work runs in thread context. A real-time thread can therefore preempt most kernel activity. Some sections, such as the core of the scheduler and the low-level interrupt entry, still run with preemption or interrupts disabled, and their duration remains part of the latency [4].

Timers and the wake-up path. High-resolution timers (hrtimers) allow a thread to sleep until a precise instant [19]. They are programmed on a per-CPU hardware timer and, when they expire, raise a local timer interrupt. A thread that calls `clock_nanosleep` with an absolute time blocks until then. At expiry, the CPU leaves idle if necessary, handles the timer interrupt, and the timer handler wakes the thread up. With `PREEMPT_RT`, most hrtimers expire in softirq context, and therefore in a kernel thread, but the timer on which a real-time thread sleeps is marked to expire in the timer interrupt itself [20]. The scheduler then selects the thread, switches context to it, and the thread returns to user space. Each of these steps adds to the wake-up latency. The kernel code executed during a system call, including the scheduler code that runs when the thread blocks, executes in the context of that thread, so the operating system and the performance counters attribute it to the thread itself.

Interrupts and per-CPU kernel activity. Besides the threads of applications, each CPU executes kernel activity that does not belong to them. Device interrupts are delivered to the CPUs allowed by their interrupt affinity, configurable through `/proc/irq` [24]. On Arm platforms, the interrupt controller delivers each interrupt to a single CPU chosen from that mask, reported by the kernel as its effective affinity [17], so an interrupt whose mask includes every CPU is in practice handled by one of them only. Softirqs carry deferred work, such as timer processing, scheduler load balancing, and read-copy-update (RCU) callbacks. In a standard kernel they run when an interrupt completes or in the per-CPU `ksoftirqd` thread; with `PREEMPT_RT` they always run in

thread context. Other per-CPU kernel threads, such as `rcuc`, handle further deferred work. Finally, a CPU can interrupt another with an inter-processor interrupt (IPI), for example to ask it to run the scheduler. All of this activity can run on a CPU dedicated to a real-time task and delay it.

2.3 Multicore scheduling and CPU isolation

Load balancing. Per-CPU runqueues avoid contention on a single global queue, but the number of runnable tasks can become uneven across CPUs: one CPU may have several tasks waiting while another is idle. Linux corrects this by migrating tasks. Real-time tasks are pushed to or pulled from other CPUs so that the highest-priority runnable ones are executing, while normal tasks are redistributed by the load balancer of the fair class. The load balancer runs periodically, triggered by the scheduler tick through the `SCHED` softirq, on behalf of idle CPUs, and when a CPU is about to become idle. Balancing is organized hierarchically over *scheduling domains*, groups of CPUs within which the scheduler balances the load, nested so that they follow the hardware topology [22]. Each set of CPUs that is balanced separately from the others, such as an exclusive cpuset partition, has its own *root domain*, which holds the state shared by those CPUs. Looking for a task to move requires inspecting the state of other CPUs, locking their runqueues, and checking each candidate, and this work is executed by the CPU performing the balance.

Newly-idle balancing. When a CPU is about to become idle, the scheduler tries to pull work from a CPU that has more runnable tasks than it can execute, because an idle CPU while tasks wait elsewhere is wasted capacity. This is *newly-idle balancing*. It belongs to the fair class and pulls only normal tasks, but it runs whenever the CPU is about to become idle, whatever the class of the task that has just blocked: a real-time thread that goes to sleep triggers it as well. Checking every CPU at each sleep would be expensive, so `newidle_balance` first makes two inexpensive checks. The first compares how long the CPU is expected to stay idle with the cost of migrating a task. The second reads an indicator kept in the root domain, the flag `rd->overload`, which says whether at least one of its CPUs is overloaded, that is, has more than one runnable task. If the expected idle time is too short, or if no CPU is overloaded, `newidle_balance` returns at once: we call this the short path. Otherwise it looks for a task to pull, calling `load_balance` on each scheduling domain of the CPU: we call this the long path. The indicator says only that some CPU is overloaded, not whether its waiting tasks may run elsewhere, so the long path may examine tasks and find none that can be moved. Chapter 6 shows how often each path is taken under the workloads of this thesis.

CPU affinity. Each thread has an affinity mask listing the CPUs on which it may run, and both the scheduler and the load balancer place the thread only on those CPUs [28]. Affinity constrains the thread, not the CPU: pinning a real-time task to a CPU and the other applications to the remaining ones keeps these applications apart,

but any thread without a restrictive mask can still run on the real-time CPU, and that CPU still takes part in load balancing with the others.

Control groups and cpusets. *Control groups* (*cgroups*) organize processes into a hierarchy to which resource controllers apply limits [12]. The *cpuset* controller restricts the CPUs and memory nodes that the tasks of a group may use. In *cgroup v2*, a cpuset can also be configured as a *partition root*, which gives it exclusive ownership of its CPUs and a separate set of scheduling domains; we call such a group a *cpuset partition*. Other ways of reserving CPUs include the `isolcpus` and `nohz_full` boot parameters, which remove CPUs from load balancing and reduce the scheduler tick on them [23]. All these mechanisms act on scheduling alone. A cpuset partition does not change where interrupts are delivered, which their own affinity decides (Section 2.2), and an isolated CPU still shares caches, interconnect, and memory with the other CPUs. The configuration used in this thesis is described in Section 4.3.

2.4 Shared resources on heterogeneous SoCs

Cache hierarchy. Processors hide the latency of main memory behind a hierarchy of caches that keep recently used data and instructions close to the core [11]. Data move between levels in fixed-size lines. An access served by a cache is a hit; on a miss, the line is fetched from the next level and installed, which Arm performance events call a refill [1]. In Arm multicore clusters, the first levels are private to each core, while a further level is shared by the cores of a cluster, and a snoop control unit keeps the private copies coherent. Beyond the cluster, requests travel through the system interconnect, on some SoCs through a system-level cache shared by all processing elements, to the memory controller and to DRAM. The performance counters of a core count the requests that miss in the cluster cache, but cannot tell whether the system cache or DRAM serves them.

Main memory. DRAM is organized in banks, each of which keeps one row open at a time. The memory controller queues and reorders the requests coming from all the processing elements, so the service time of a request depends on the other requests in flight: it grows when requests queue, when they compete for the same bank, or when they force a different row to be opened [47]. The available bandwidth is likewise shared.

Heterogeneous SoCs and interference channels. An embedded system-on-chip (SoC) such as those of the NVIDIA Jetson family integrates CPU clusters, a GPU, and other accelerators that all use the same DRAM, without memory dedicated to the GPU. A GPU-intensive workload, such as neural-network inference, therefore generates memory traffic that competes with that of the CPUs [7, 6]. A task can suffer interference through the eviction of its lines from a shared cache, through contention in the interconnect, and through queuing and bank conflicts in the memory controller. The effect depends on how often the task leaves its private caches: a memory-bound task is exposed to every shared level, whereas a task whose working set fits in its

private caches rarely reaches them. Chapter 3 reviews how these effects have been characterized, typically with synthetic workloads designed to stress memory.

Power and frequency management. Embedded SoCs also share a power budget. Dynamic voltage and frequency scaling adapts the clock of each component to the load, and the Jetson power modes set the frequency limits of the CPUs, the GPU, and the memory [36]. To protect the hardware, the module monitors the current on its input power rail and, when it exceeds a configured limit, raises an over-current (OC) alarm, upon which the firmware briefly throttles the clocks of the CPUs and of the GPU [36]. The effective frequency of a CPU then falls below the one that the operating system keeps requesting. This couples the activity of the GPU to the speed of every CPU, through a channel unrelated to memory.

2.5 Performance monitoring and kernel tracing

Performance counters. The performance monitoring unit (PMU) of each core provides hardware counters that can be programmed to count events such as cycles, retired instructions, cache accesses and refills, and stall cycles [1]. The number of counters is small, so collecting more events than available counters requires either time multiplexing, with counts scaled to the measurement interval, or separate runs. On Linux, counters are configured through `perf_event_open` [26] and used through the `perf` tool. They can count the events of a CPU, or those of a task, in which case the kernel saves and restores them at every context switch; privilege filters further separate user-space from kernel execution. In counting mode, `perf stat` reports totals over an interval [32]. In sampling mode, `perf record` interrupts the core every given number of events and records the instruction pointer and call chain [30]; the result is a statistical profile, and a sample can be attributed to an instruction slightly after the one that caused it [46].

Derived metrics. The cycle counter advances with the clock of the core while the core is not idle, so comparing the counted cycles with the elapsed time shows whether the core ran at its nominal frequency: at a lower effective frequency, the same interval contains fewer cycles. Stall events explain where the cycles go: a frontend stall occurs when no instructions are available for dispatch, for example after an instruction-cache miss, and a backend stall when dispatched operations cannot proceed; a further event counts the backend stalls spent waiting for data from memory [1]. Counting instructions separates a change in the amount of work from a change in the time that the same work takes.

Kernel tracing. Tracing records individual events instead of totals. Tracepoints are static instrumentation points placed in the kernel source at significant events, such as context switches, entry into and exit from idle, interrupt entry, timer expiry, and system calls [18]. The ftrace function tracer records the entry of nearly every kernel function, which reveals the path the kernel follows [44]. Kprobes place dynamic probes at almost

any kernel instruction, or at the return of a function, and can read the processor registers at that point [14]. Extended Berkeley Packet Filter (eBPF) programs can be attached to these probes and tracepoints: they are checked by a verifier before loading, run inside the kernel when the event occurs, keep state in maps, and can read the time and the performance counters through helper functions [21]. Data can thus be filtered and aggregated where it is produced.

Instrumentation overhead. Every probe executes additional instructions on the CPU where the event occurs, so tracing perturbs the activity it observes. Latency measured with active instrumentation is longer than without it, and the work of the probes is counted together with that of the observed code. For this reason, the latency reported in this thesis is measured without probes, while the performance counters are read by `perf` on a whole CPU, a process, or a thread, without adding probes to the measured path. Instrumented measurements are used only to compare conditions measured with the same probes.

3. Related work

When perception software runs on other CPU cores, the wake-up latency of a periodic real-time task may change; understanding the extent of this change requires considering both shared resources and the operating-system work needed to manage activity on those cores. This chapter reviews studies of memory contention, computer-vision workloads, resource isolation, and Linux scheduling latency, relating their findings to the experimental questions addressed in this thesis. The reviewed work provides the basis for comparing an industrial perception application with a synthetic interferent and examining the latency increase that remains after CPU scheduling isolation.

3.1 Characterizing memory interference

Cavicchioli et al. [7] used controlled memory-access workloads to characterize interference between CPU cores and integrated GPUs on Tegra K1, Tegra X1, and an Intel platform. Their experiments examined how concurrent CPU and GPU activity affects memory performance through workloads that allow the access pattern or the number of competing processors to vary while keeping the workload simple. Although the measurements describe the memory behavior exercised by the benchmarks, relating that behavior to a periodic thread's wake-up latency requires an additional system-level measurement.

Capodieci et al. [6] extended this characterization to Jetson Nano, TX2, and Xavier and introduced HeSoC-mark, a suite of memory benchmarks and interferents for different processing engines. Their evaluation also used `cyclictest` under contention, providing a direct precedent for studying periodic-task latency on Tegra. By reusing `meminterf` from this suite as a synthetic interferent, the thesis adopts a reference methodology against which the industrial workload can be compared.

Using profiling metrics and machine-learning models, Masola et al. [33] studied the slowdown of GPU kernels on Jetson AGX Xavier under memory-intensive CPU activity. Their characterization shows why the relationship between memory traffic and interference depends on the workload being measured: aggregate bandwidth alone cannot describe the degradation of every kernel, whose access behavior also matters. Sustained memory traffic therefore provides a controlled interference scenario without establishing that an industrial application will produce effects of the same magnitude or distribution.

These studies motivate the synthetic reference used in this thesis: a simple interferent helps characterize the platform while establishing a repeatable condition for comparison. The industrial workload introduces a further question, namely whether perception software measurably changes the wake-up latency of a task whose configuration is otherwise unchanged. Keeping the victim fixed allows this question to be examined without also changing the application being timed.

3.2 Evidence from computer-vision applications

Otterness et al. [41] evaluated the NVIDIA TX1 by combining benchmarking and black-box observations of GPU behavior with computer-vision case studies involving image classification and road-sign recognition. Their work connects platform behavior to applications representative of autonomous-driving workloads while distinguishing throughput improvements from the timing behavior required by real-time applications. This distinction matters because a satisfactory average processing rate does not, by itself, describe the delays experienced during individual executions.

Bechtel and Yun [2] studied an industrial augmented-reality head-up display on Jetson Nano, combining visual simultaneous localization and mapping with neural-network-based head-pose estimation. They evaluated contention between application components and interference from synthetic aggressors, observing effects on both execution timing and the accuracy of the estimated trajectory. The study proposes RT-Gang++, a framework that combines scheduling with protection against contention on shared resources, and provides an industrial case in which realistic vision applications and their interference have already been examined.

Whereas the cited application studies examine the performance of the vision workload as a central outcome, this thesis assigns the industrial perception application, referred to as tkHPick, the role of a non-synthetic interferent. The quantity measured is the response of a separate periodic CPU thread: `cyclictest` evaluates the platform's ability to wake that thread on time while the industrial application runs. Since the victim is a measurement workload rather than a complete control application, its latency cannot be interpreted as the execution time of an industrial control function.

The comparison between tkHPick and HeSoC-mark separates application realism from experimental control: the synthetic condition has deliberately simple memory-access behavior, while tkHPick includes perception processing, recorded-input playback, and shared-memory services. The measured effect of tkHPick is therefore that of the configured application stack, without assuming that every additional delay originates from one component or from memory alone. Using both conditions connects controlled memory characterization to the timing effects of the application supplied for the industrial case.

3.3 CPU isolation and shared-memory protection

CPU scheduling isolation changes task placement and scheduler load balancing without isolating the memory hierarchy. Linux CPU affinity restricts where a task may execute, while a valid *cgroup v2 cpuset* partition creates a separate *scheduling domain* [28, 12, 22]. Neither control allocates private cache capacity or DRAM banks to the selected core.

Shared-memory protection can instead target the contended resources. Examples include DRAM bank partitioning with PALLOC [47], cache partitioning on Arm DynamIQ systems studied by Pradhan et al. [42], and memory-bandwidth regulation

through per-core budgets with MemGuard [48]. These techniques provide context for the problem; evaluating mitigation strategies is outside the scope of this thesis.

In this thesis, the affinity–cpuset comparison measures the latency increase that persists after CPU scheduling isolation while retaining the physical cache-sharing relationship. A residual increase shows a limit of the evaluated configuration, but does not establish memory contention as its sole cause. The result is interpreted alongside memory measurements and kernel diagnostics, since interrupt handling and other kernel activity may still affect the measurement.

3.4 Linux latency and kernel activity

Bristot de Oliveira et al. [4] analyzed scheduling latency in PREEMPT_RT Linux using a model of kernel synchronization and tracing of the relevant delay components. Their work addresses a limitation of `cyclictest`: the measured latency does not identify the operations responsible for the delay. It provides a basis for examining kernel activity alongside latency distributions, while also discussing the perturbation caused by tracing.

In this thesis, kernel diagnostics support the interpretation of the latency measurements. More instructions attributed to the victim can indicate additional operating-system work even when its user-space loop is unchanged [29], whereas a longer delay may also reflect the same work taking more time. The scheduler investigation examines why activity on other CPUs changes this work; additional scheduler execution alone does not demonstrate a scheduling error or quantify its contribution to individual latency outliers.

3.5 Positioning of this thesis

The primary aim of this thesis is to characterize the increase in a periodic task's wake-up latency when an industrial computer-vision application runs concurrently on the Jetson Orin Nano. With the victim configuration kept fixed, the experiment compares a baseline, a synthetic memory interferent from HeSoC-mark, and tkHPick. It then evaluates how much of the latency increase persists after CPU scheduling isolation by comparing CPU affinity with a separate cpuset partition. To support the interpretation of these measurements, the kernel investigation examines the additional work attributed to the victim.

Previous studies have demonstrated memory contention on Tegra, evaluated realistic vision workloads, and analyzed Linux scheduling latency. Drawing on those findings, this thesis brings the corresponding questions together in one experimental case, using a non-synthetic perception stack as the interferent of a periodic CPU thread alongside a synthetic reference and an explicit comparison of scheduling configurations. Its contribution lies in the evidence obtained for this application and platform and in the diagnosis supported by the measurements, without requiring a claim that memory interference or the insufficiency of CPU placement was previously unknown.

Chapter 4 defines the configurations and the limits of their interpretation, while Chapter 6 presents the latency changes and assesses the evidence concerning kernel and memory-system activity, and Chapter 7 repeats the analysis of the isolated CPU with a victim that has the memory footprint of a high-speed vision system. The observed degradation, the effects of the isolation configuration, and the explanation of the measured activity are thus treated as separate conclusions.

4. Experimental design and methodology

The measurements of Chapters 6 and 7 compare the wake-up latency of a periodic real-time thread, the victim, when the other cores of its cluster run nothing, a synthetic memory workload, or an industrial perception application. This chapter fixes what is compared: the platform, the two victims (a minimal task and one with the memory footprint of a high-speed vision system), the two interferents, the two ways of keeping them apart, CPU affinity and a *cpuset* partition, and how each condition is acquired, measured, and summarised.

4.1 Hardware and software configuration

4.1.1 NVIDIA Jetson Orin Nano

The experimental platform is an NVIDIA Jetson Orin Nano 8GB module mounted on a Seeed Studio A603 carrier board [45], which the company supplied together with tkHPick, the perception application studied in this thesis (Section 4.2.4). The module has six Arm Cortex-A78AE CPU cores, an integrated NVIDIA Ampere GPU with 1,024 CUDA cores and 32 Tensor cores, and 8 GB of 128-bit LPDDR5 main memory [37, 39]. The CPU cores and the GPU sit on the same system-on-chip and share memory resources, so the platform is a realistic host for an embedded perception workload and for the interference it causes.

The CPU cores, numbered CPU0–CPU5, form a four-core cluster and a two-core cluster, each with its own shared L3 cache. Every core also has private L1 instruction and data caches and a private unified L2 cache, as described in the NVIDIA module datasheet [38] and summarized in Table 4.1.

Cache	Capacity per instance	Sharing
L1 instruction	64 KiB	Private to each core
L1 data	64 KiB	Private to each core
L2 unified	256 KiB	Private to each core
L3 unified	2 MiB	CPU0–CPU3
L3 unified	2 MiB	CPU4–CPU5

Table 4.1: CPU cache hierarchy and sharing groups on the experimental platform.

The two clusters still share resources beyond their L3 caches, including the 4 MB system cache described by NVIDIA [38]. No cache partitioning is configured.

4.1.2 Operating system and power configuration

The board runs Ubuntu 22.04.5 LTS with NVIDIA Linux for Tegra R36.4.3 and the `5.15.148-rt-tegra` kernel with `PREEMPT_RT`. Jetson Linux R36.4.3 is based on Linux 5.15 and an Ubuntu 22.04 root file system, and NVIDIA provides a real-time kernel configuration for this release [40, 35]. The measurement tools are `cyclictest` 2.20, which measures the wake-up latency of the victim, and `perf` 5.15.148, which gives access to the performance counters.

The board operates in the `MAXN_SUPER` power mode, with a nominal CPU frequency of 1.728 GHz [36]. The firmware can still lower the effective clock when an over-current alarm fires (Section 2.4); Section 6.6.4 shows that this happens with tkHPick.

4.2 The victim and the interferents

4.2.1 The victim

The thesis uses two victims, both built on the measurement thread of `cyclictest`. The first, the *minimal task*, is the original tool. The tool sleeps until the next activation of this thread and measures the time between the requested wake-up time and the moment the thread runs again [16, 3]. This wake-up latency shows how punctually the system activates a periodic real-time thread, and we compare it with and without interferents on the other cores.

The configuration of each victim is the same in all experimental conditions. A single measurement thread runs under `SCHED_FIFO` with priority 90 and is pinned to CPU0 or CPU3, depending on the condition. Its activation period is 1 ms, a nominal frequency of 1 kHz, and memory locking keeps its pages resident in RAM [25]. Chapter 5 gives the complete command.

For activation i, the wake-up latency is defined as

$$L_i = t_i^{\text{wake}} - t_i^{\text{target}}, \tag{4.1}$$

where t_i^{target} is the intended activation time and t_i^{wake} is the timestamp the victim takes when it runs again after the timed sleep. The latency is expressed in microseconds and therefore includes the time the kernel needs to wake up the victim and switch to it, but not the execution of a control function.

For each acquisition, we keep the JSON summary, the activation count, and the latency histogram. Samples beyond the histogram range are still counted through the overflow information, so an unusually large latency is not lost, although its exact value cannot be recovered from the bins.

4.2.2 The H³ Vision victim

The minimal task does almost nothing between two wake-ups, so its code and data stay in the private caches of its core. A real task instead processes new data at every

activation. The second victim adds this work to `cyclictest`: after each wake-up, once the latency has been recorded, the thread reads and writes an amount of data taken from a real high-speed vision system for robots, and then goes back to sleep.

The amount comes from H^3 Vision, a high-speed vision platform for robots that acquires and processes a 1024×1024 colour image with 10 bits per pixel at 1000 frames per second [15]: every millisecond, a new image of about 1.3 MB has to be handled. At every activation the H^3 Vision victim reads about 1.3 MB and writes 2,000 bytes of results, five times the private L2 cache of a core and more than half of the L3 cache of the cluster (Table 4.1).

Each new image arrives in memory and is not in the cache. The H^3 Vision victim reproduces this condition: each activation reads a different region of a 64 MiB ring, so the data it reads have not been touched since about 64 MB of traffic earlier. It reads the input in order, computes a value from it, and writes the output, but it does not reproduce the image processing of the real system. Its period is 1 ms, the frame period of the vision system and the period of the minimal task, so the latency of the two victims is measured in the same way and can be compared directly. Chapter 5 describes how `cyclictest` was extended.

4.2.3 Synthetic memory workload

The synthetic interferent is the original `meminterf` program from the HeSoC-mark project, called MEMINTERF in the rest of the thesis [13]. It allocates and initializes two buffers and then copies their contents back and forth, with the following options:

- `-test=memcpy`: each iteration copies the whole source buffer to the destination and then copies it back in the opposite direction.
- `-size=128`: each buffer is 128 MiB, so every process allocates 256 MiB.
- `-iterations=1000000`: the loop limit is high enough to keep the interferent running for the whole acquisition window. The campaign orchestrator stops the process when the measurement interval ends.

One process is pinned with `taskset` to each of the three interfering CPUs. Each process has a single thread that copies memory without ever blocking, so each interfering CPU always has one runnable user task. The buffers are far larger than the caches listed in Table 4.1, so MEMINTERF serves as a memory-intensive reference for the comparison with tkHPick. It is used as a benchmark: it shows how far a synthetic memory stressor reproduces the interference of tkHPick, and the thesis does not analyse how it generates its traffic.

4.2.4 Industrial perception application

The industrial interferent, tkHPick, was supplied by the company and derives from an autonomous-driving software stack. In the configuration used here, it performs instance segmentation and object pose estimation for robotic manipulation, which makes

it a concrete case of perception software running next to a periodic real-time thread. We treat tkHPick as a black box: its inputs, active components, configuration, and resource placement define the experimental workload. The analysis looks at the interference its execution produces and does not examine how the perception algorithms are implemented.

tkHPick consists of four processes. RouDi, the iceoryx service, manages shared memory and service discovery using the memory-pool configuration shipped with the application [8]. The instance-segmentation process consumes color images and produces instance detections for the pose-estimation process. Pose estimation combines color images, depth data, point clouds, and the segmentation results to estimate object poses, with iterative closest point (ICP) refinement enabled. Finally, the recorded-data player `tkCore_bag_play` reads `cem_multi.mcap` and supplies the recorded sensor input. MCAP associates messages with channels, schemas, and timestamps [34], and the player restarts after each successful completion so that the recording is replayed for the whole acquisition.

The processes are multithreaded: in a snapshot taken during a diagnostic run, they had 67 threads in total (23 in the player, 13 in instance segmentation, 25 in pose estimation, and 6 in RouDi). The instance-segmentation and pose-estimation processes run their neural-network inference on the integrated GPU.

Figure 4.1 shows the data flow between these components. The recording and playback configuration are the same in every comparison. All four processes share the set of interfering CPUs, and the scheduler decides how they are distributed within it.

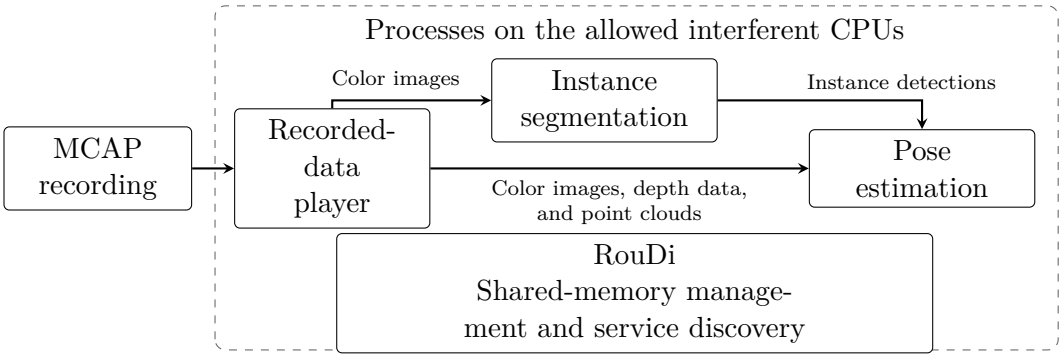


Figure 4.1: Functional organization of tkHPick. Arrows show the inputs used by the perception processes, while RouDi provides shared-memory management and service discovery.

4.3 Experimental conditions and acquisition protocol

4.3.1 CPU placement and isolation

The experiment compares three scenarios. In the baseline, the victim and the measurement infrastructure run without any interferent. In the MEMINTERF scenario,

three MEMINTERF processes run alongside the victim, and in the tkHPick scenario the four processes of tkHPick do.

Each scenario is run with the victim on CPU0 and on CPU3, using the placements in Table 4.2. In both cases the interferents occupy the remaining cores of the same L3 domain and the collection processes stay on CPU4–CPU5, where they compete less with the victim for CPU time, so the two victim positions differ only in the core the victim runs on. This core matters for interrupts: with the interrupt policy of the platform, the peripheral interrupts are handled by CPU0, because each interrupt is delivered to a single CPU of its mask (Section 2.2), while CPU3 receives practically none.

Victim CPU	Interferent CPUs	Collector CPUs
CPU0	CPU1, CPU2, CPU3	CPU4, CPU5
CPU3	CPU0, CPU1, CPU2	CPU4, CPU5

Table 4.2: CPU placement for the two victim locations. MEMINTERF uses one process per interfering CPU, whereas the four processes of tkHPick share the listed CPU set. The baseline contains no interferent.

For each placement, we compare CPU affinity with a cpuset partition. CPU affinity is the usual way to keep a real-time thread apart from the other applications, but it leaves the victim CPU in load balancing with the other cores. The cpuset partition also removes the victim CPU from load balancing, so comparing the two shows how much the scheduler weighs on the latency, with and without the interferents. With CPU affinity, the victim is restricted to its CPU and the interferents to the complementary set shown in the table. Affinity only limits where a thread may run: it keeps the interferents apart, but other eligible tasks can still run on the victim CPU [28].

In the cpuset partition, the victim runs in a *cgroup v2* cpuset that contains only its CPU and is configured as a *partition root*. The kernel then removes that CPU from the CPUs available to every other cgroup [12] and gives it a *root domain* and *scheduling domains* of its own, so the scheduler no longer balances load between that CPU and the others [22]. No other user task can run on the victim CPU, while per-CPU kernel threads and interrupts still can (Section 6.4). The memory hierarchy and the placement of the interferents stay as they are.

Both configurations keep the default interrupt affinity of the platform, so they differ only in how threads are scheduled. Before every pass, the acquisition software checks that the default interrupt mask includes all six CPUs and records the requested and effective affinity of every interrupt. It also checks the effective CPU affinities and the cpuset membership, including the partition-root state and, with CPU affinity, the absence of leftover partitions, and saves the resulting masks with the measurement metadata.

The H^3 Vision victim is measured on CPU3 with the cpuset partition, the configuration that Chapter 6 identifies as the best isolated, with the same three scenarios and the same placement of the interferents, and the same three CPU-wide counter passes as the cpuset campaign. The minimal task is measured again in the same campaign,

so that the two victims are compared under the same conditions. As in the other campaigns, the interrupts keep the default affinity of the platform. The H^3 Vision victim is also measured on CPU0 with CPU affinity only, in a separate campaign that collects only the processor-activity pass; its latency opens Chapter 6.

4.3.2 Repeated acquisitions and counter groups

Two victim locations, three scenarios, and two configurations give twelve conditions for the minimal task; the campaign of Chapter 7 adds the three scenarios for each of the two victims on the isolated CPU3, and the campaign on CPU0 the three scenarios for the H^3 Vision victim with CPU affinity only. The measurements therefore come from four campaigns: CPU affinity and cpuset partition for the minimal task, the campaign of Chapter 7 on CPU3, and the campaign of the H^3 Vision victim on CPU0. In each campaign every condition is repeated three times; the repetitions are grouped into three blocks, and each block contains all the conditions of the campaign in an order randomized with the seed 424242. Repetitions of the same condition are thus never collected back to back.

The Cortex-A78AE performance monitoring unit (PMU) has only six programmable event counters [1, Section C2.1], which is not enough to collect all events in one run. We split them into three groups (processor activity, L1 caches, and L2 and L3 caches). With CPU affinity each group is measured at two scopes, the victim CPU and the victim process, so each repetition consists of six counter passes. With the cpuset partition the victim is the only user task on its CPU, and the three groups are measured on the CPU alone. Each pass lasts 300 s. Every pass is a separate execution with the same workload configuration, and the passes run in the same order in every repetition.

Parameter	Configuration
Victim CPU	CPU0 or CPU3
Scenario	Baseline, MEMINTERF, or tkHPick
Configuration	CPU affinity or cpuset partition
Independent repetitions per condition	3
Counter passes per repetition	6 (CPU affinity), 3 (cpuset partition), 1 (H^3 Vision on CPU0)
Acquisition duration per pass	300 s
Warm-up or settling interval	10 s
Histogram range and resolution	1,000 μs, 1 ns
Interrupt-affinity policy	Platform default in both configurations

Table 4.3: Acquisition parameters used throughout the experimental comparison. Histogram overflows are retained.

The workloads are restarted for each pass. The interferents run for 10 s before the victim starts its 300 s measurement, and the baseline waits for the same 10 s. After each pass, the workloads are stopped before the next acquisition begins.

The processor-activity pass counts cycles, user-space instructions, kernel instructions, backend stalls, memory stalls, memory and bus accesses, and the software event `task-clock` over the same 300 s window. Because they come from one execution, execution cost can be normalized by the amount of work done without mixing counters from different runs. The other groups contain the access and refill pairs needed for the cache comparisons. Chapter 5 gives the exact commands and event encodings. The diagnostic runs described there are shorter, from 3 to 30 s, because they count events that occur at every wake-up: at 1,000 activations per second, even the shortest run contains about 3,000 sleeps of the victim, and the per-phase measurement about 60,000 iterations per level and scenario.

4.4 Measured quantities and attribution scopes

4.4.1 Latency and processor activity

The analysis starts from the wake-up latency, which shows how the activation of the victim changes under interference, and then turns to the processor activity behind that change. Retired instructions measure how much work is executed, and cycles and stalls show how expensive that work is. Cache accesses and refills describe memory activity. We count both kernel and user-space instructions, because a user-space loop that does not change can still trigger a different amount of kernel work.

Measurement	Role in the analysis
Wake-up latency	Latency in the activation of the victim
Retired instructions	Amount of execution counted in the monitored scope
Cycles and task-clock	Processor cycles and execution time in that scope
Stall cycles	Cycles in which the core waits for instructions or data
Cache accesses and refills	Activity at the selected cache level
Memory and bus accesses	Memory-related activity during the workload
Interrupts, IPIs, and softirqs	Interrupt and scheduler activity reaching the victim CPU
Interferent CPU counters	Activity on the three interferent cores
Module power and requested frequency	Cause of the time lost to throttling

Table 4.4: Measured quantities and their purpose. IPI denotes an inter-processor interrupt.

The hardware events follow the definitions in the Cortex-A78AE technical reference manual [1, Sections C2.3 and C3.4]. Backend stalls count the cycles in which the backend cannot accept new operations from dispatch, and the memory-stall event counts cycles stalled on last-level cache misses within the core clock domain. In the analysis they are called waiting for data from memory. Frontend stalls, used only in the per-phase measurement of Section 5.6, count the cycles in which no instructions

are available for dispatch. In the analysis they are called waiting for instructions; on this victim they follow instruction-cache misses (Section 7.3). Access and refill events describe activity at their cache level, and `BUS_ACCESS` counts data beats between the core and the snoop control unit (SCU).

Interrupt activity is measured from the per-CPU counts in `/proc/interrupts` and `/proc/softirqs` before and after each window [24]. We use the victim CPU's column, together with the interrupt labels and IPI entries reported by the kernel. These counts give the number of interrupts the CPU handled; measuring how long the handlers ran would require a separate measurement.

4.4.2 CPU-wide and process measurements

With CPU affinity, counters are collected at two scopes [32]. CPU-wide counters include everything executed on the victim CPU, while process counters follow the `cyclictest` process, both its manager and the victim thread. With the cpuset partition only CPU-wide counters are collected: the victim is the only user task on its CPU, but the counts also include the kernel threads, ticks, and interrupts handled there.

Process counts also include kernel code executed on behalf of the monitored process, and the `u` and `k` modifiers separate user-space from kernel instructions [29]. The instructions attributed to the victim can therefore grow through additional kernel execution while the periodic loop itself stays the same. The interferents are measured separately, with CPU-wide counters on their three cores.

4.5 Statistical treatment and interpretation limits

4.5.1 Latency statistics

For each condition, the latency statistics come from the CPU-wide processor-activity pass of each of the three repetitions; the histograms of the other passes come from different monitoring configurations and are not used. For repetition r, let $h_{r,j}$ be the number of samples in the 1 ns bin j, o_r the number of overflow samples, and $N_r = \sum_j h_{r,j} + o_r$ the number of activations. When $o_r = 0$, the mean latency is

$$\mu_r = \frac{1}{N_r} \sum_j j \, h_{r,j}, \tag{4.2}$$

computed from the histogram rather than from the rounded average of the `cyclictest` summary. In every run reported in this thesis the maximum stays below the 1 ms range of the histogram, so $o_r = 0$. A percentile p is the first bin whose cumulative count reaches $\lceil pN_r \rceil$; a percentile whose rank falls beyond the bins would be marked unavailable, and the maximum reported by the tool is kept apart from the histogram.

Chapter 6 reports the mean, P99, and P99.9 as the mean over the three repetitions, and the maximum as the largest value observed in them. Chapter 7 computes the same statistics in the same way on the processor-activity pass of its campaign. Each scenario is compared with the baseline measured on the same CPU and in the same

configuration. Rare events, such as latencies above 200 μs, are counted on the same histograms merged over the three repetitions, that is, over 900,000 activations per condition. The largest latency observed in these finite windows is an experimental maximum, not a worst-case bound.

4.5.2 Counter normalization and interpretation

Counters are compared per wake-up, dividing each count by the activations of the same run (Section 6.2). Ratios between events of the same pass use the raw counts; ratios between events of different passes, such as the share of accesses served by the private caches, use the values per wake-up of each pass. Frontend, backend, and memory stalls are reported separately, because the definitions of the last two overlap.

Refill counts and bus accesses describe memory-system activity; they do not measure the victim's DRAM service time or memory-controller contention. A latency increase cannot be decomposed by subtracting a change in instruction count, since the two are different quantities. On the isolated CPU3, Chapter 6 splits it with a separate per-phase measurement (Section 5.6) that reads the counters inside the latency window. For tkHPick, the counters describe the activity on its cores; showing a slowdown of the application would require a measure of completed work and a reference execution.

Chapter 5 describes the collection tools. Chapter 6 measures the latency with CPU affinity, then the instructions of the victim and the kernel path that adds them, then the effect of the cpuset partition and of the peripheral interrupts on CPU0, and finally what remains on the isolated CPU3. Chapter 7 repeats the analysis of the isolated CPU3 with the H^3 Vision victim. The event groups come from separate windows, so the comparison characterizes operating conditions and cannot link a single latency outlier to a specific cache event.

5. Measurement infrastructure and diagnostic tools

The comparisons of Chapters 6 and 7 rest on two kinds of measurement: long campaigns that repeat every condition under the same automated procedure, and short diagnostic runs that look inside the kernel of the victim CPU. This chapter describes the tools behind both: the acquisition infrastructure, the extension of `cyclictest` that gives the H^3 Vision victim its memory footprint, the counter collection, the tracing of kernel paths, the measurement of scheduler decisions and of their cost, and the per-phase measurement of the wake-up on the isolated CPU.

5.1 Automated experiment execution

5.1.1 Controller and acquisition worker

The comparison needs the same workload arrangement to be reproduced across victim CPUs, scenarios, and configurations. The `jetson-campaign` infrastructure does this with a Python controller on the host computer and an acquisition worker on the Jetson. YAML configuration files define the workload commands, CPU sets, measurement durations, repetitions, and event passes, and the controller generates the whole experiment from that single description instead of from hand-edited commands.

The controller copies the acquisition code and configuration over SSH and starts the acquisition worker in a workspace on the target. From then on the acquisition worker runs the measurement interval locally, including process startup, counter collection, and termination. The host only controls the experiment and retrieves its results; the periodic timing of the victim and the acquisition deadline are handled on the board. The configuration and the revision of the orchestration code are saved with the outputs, together with identifiers for the workload resources, so the commands behind any measurement can be recovered from its record.

5.1.2 Workload lifecycle and placement

For each counter pass, the acquisition worker applies the following procedure:

1. Resolve the victim and interferent CPU sets, check that the required events are available, and create the victim partition when the *cpuset* partition is selected.

2. Launch the three MEMINTERF processes or the four processes of tkHPick and wait for the warm-up interval. The baseline waits for the settling interval specified in the protocol.

3. Start the victim and the configured collectors, recording the commands, process identities, and start and stop times.

4. Terminate the processes belonging to the acquisition, collect the final snapshots, and parse the outputs before the pause that precedes the next pass.

The victim is started with the same command in every pass, here with the victim on CPU0:

```
cyclictest -a 0 -t 1 -p 90 --policy=fifo -m -i 1000 -D 300 \
           -N -h 1000000 -q --json=<file>
```

The options pin the measurement thread to the victim CPU (`-a`), create a single thread (`-t 1`) under `SCHED_FIFO` with priority 90, lock the memory of the process (`-m`), set a period of 1,000 μs (`-i`) and a duration of 300 s (`-D`), report times in nanoseconds (`-N`), and build a histogram of 1,000,000 bins of 1 ns (`-h`) that is saved in the JSON output [3].

The player of tkHPick restarts each time the recording completes successfully, so recorded input is available for the whole measurement window. The workload configuration defines which commands and resources belong to a condition, and process-lifetime checks detect components that fail. These checks say nothing about how much perception work the application completes.

CPU affinity is applied separately to the victim, the interferents, and the collection processes. With the cpuset partition, a launcher places the victim in the selected cgroup before executing `cyclictest`, and the acquisition worker records the effective CPU set and partition state. Placing the collectors on CPU4–CPU5 only determines where the collection programs execute; the monitored CPU is still CPU0 or CPU3. A `perf` process running on CPU4, for example, can read events counted on CPU0. Moving the collector does not move the kernel work needed to manage events on the monitored CPU, which stays there.

The process manager records each process identifier together with its start time and tracks the descendants created by the launched workloads. Cleanup first requests termination and then stops any owned processes that remain, so the next pass begins with freshly started workloads. The same ownership information lets the acquisition worker tell its own processes apart from other applications on the board.

5.1.3 Acquisition records

Each acquisition produces a directory with the raw `cyclictest` output, performance-counter files, process logs, placement metadata, and interrupt snapshots. The latency histogram and overflow information are saved next to the summary, and the statistics are reconstructed later with the rules of Section 4.5. Failed attempts keep their outputs and failure reason in separate directories, and a replacement acquisition never overwrites the earlier record. After the transfer to the host, a file manifest with hashes confirms that the retrieved files match the outputs saved on the target.

5.2 The H³ Vision victim: extending `cyclictest`

The H³ Vision victim (Section 4.2.2) is a modified `cyclictest`, built from rt-tests 2.2, the version of the package installed on the Jetson. All the changes are in the source of `cyclictest`, and without the new options the measurement loop is the original one: an A/B comparison between the original and the modified binary, both without work, gave the same mean latency, 2.65 and 2.64 μs.

Where the work runs. In the loop of the measurement thread, the work is inserted right after the latency of the activation has been added to its histogram and before the next release time is computed:

```
t0  scheduled release (timer expiry)
    interrupt -> wake-up -> scheduler -> return to user space
t1  clock_gettime()       latency    = t1 - t0   (unchanged)
    WORK: read job-read bytes, write job-write bytes
t2  end of the work
    clock_nanosleep() until the next release
```

The latency is therefore measured exactly as in the minimal task, and the work can influence it only through the state it leaves in the hardware for the next wake-up. If the work overran the period, the releases already in the past would be skipped and counted; with the H³ Vision footprint at 1 ms this never happened.

Data that are never in the cache. At startup, after the memory has been locked, the tool allocates a 64 MiB ring, locks it, touches every page, and divides it into slots, each holding the input and the output region of one activation. For the H³ Vision victim a slot is about 1.3 MB, so the ring holds 52 slots, and each activation uses the next one. When the victim returns to a slot, about 64 MB of traffic has passed through the cache hierarchy, so its lines are no longer in the L1, L2, or L3 caches or in the system cache, as if another task had just written them. Since the memory is locked and already touched, the measurement causes no page faults.

The work. Each activation reads all the 8-byte words of every 64-byte line of the input region in sequential order and adds them up, then writes the output region linearly with values that depend on the sum:

```
for (i = 0; i < job.lines; i++) {
        p = (uint64_t *)(base + i * JOB_LINE);
        sum += p[0] + p[1] + p[2] + p[3] + p[4] + p[5] + p[6] + p[7];
}
for (i = 0; i < words; i++)
        w[i] = sum + i;
job.sink = sum;   /* volatile: the reads cannot be removed */
```

The sum keeps the compiler from removing the reads, and the dependence of the output on the input follows the structure of a control task, which reads its inputs, computes, and writes its command.

Options and output. The new options are `-job-read` and `-job-write`, the bytes read and written per activation (about 1.3 MB and 2,000 bytes for the H^3 Vision victim), and `-job-ring` for the size of the ring. The JSON summary of `cyclictest` gains, for the measurement thread, an object with the configuration and the number of activations and of skipped releases. The build script verifies the checksum of the rt-tests 2.2 archive before applying the patch.

5.3 Performance counter acquisition

5.3.1 Event selection and counter passes

The acquisition worker collects hardware events through `perf stat`, using an explicit mapping for the `armv8_cortex_a78` PMU. Before collection, the mapping is checked against the aliases exposed under `/sys/bus/event_source/devices/armv8_cortex_a78/events`, and an event is used only when its alias and encoding match the expected mapping. The names in the output therefore correspond to the event definitions in the Cortex-A78AE manual [1, Section C2.3].

Table 5.1 lists the processor and memory-activity events used in the analysis. Hardware events are selected with the syntax `armv8_cortex_a78/ALIAS/`, using the alias in the table. User-space and kernel instructions use the same retired-instruction event with different privilege filters, selected through the `u` and `k` modifiers [29]; for example, `armv8_cortex_a78/inst_retired/k` selects kernel instructions. The processor-activity pass collects these two instruction counts together with cycles, backend and memory stalls, memory and bus accesses, and the software event `task-clock`. Because they share a window, the normalizations of Chapter 4 draw on a single execution and a single attribution scope.

Quantity	Linux PMU alias	Encoding
CPU cycles	`cpu_cycles`	`0x11`
Retired instructions	`inst_retired`	`0x08`
Backend stalls	`stall_backend`	`0x24`
Memory stalls	`stall_backend_mem`	`0x4005`
Memory accesses	`mem_access`	`0x13`
Bus accesses	`bus_access`	`0x19`

Table 5.1: Processor and memory-activity events. Encodings follow the Cortex-A78AE technical reference manual [1, Table C2-1]; `task-clock` is a software event and has no hardware encoding.

Cache activity is collected in two passes, one for the L1 data and instruction caches and one for the L2 and L3 caches, each with cycles and the access, refill, and long-miss events of its levels. Table 5.2 gives their encodings. The long-miss event counts reads only for L1 data, L2, and L3, while L1 instruction uses its instruction-cache event. All of these are event counts with their own hardware definitions; none measures how long an individual memory access takes [1, Table C2-1]. Splitting the event families

keeps each pass within the available programmable counters, but the running-time information still has to be checked when several collectors are active.

Cache	Access	Refill	Long miss
L1 data	`0x04`	`0x03`	`0x39`
L1 instruction	`0x14`	`0x01`	`0x4006`
L2	`0x16`	`0x17`	`0x4009`
L3	`0x2B`	`0x2A`	`0x400B`

Table 5.2: Cache-event encodings used in the two cache passes [1, Table C2-1]. Access and refill events are acquired together for each level.

5.3.2 CPU, process, and thread measurements

The collection command determines whose execution contributes to a counter. For the victim CPU, `perf stat -a -C` selects CPU-wide collection on the designated core. For the victim process with CPU affinity, `perf stat` launches the pinned `cyclictest` command and follows its inherited execution [32].

Table 5.3 summarizes the measurement scopes. The process measurement includes the manager and the victim thread, whereas the scheduler diagnostics select only the victim. For those diagnostics, the victim is identified by its thread identifier (TID), scheduling policy, priority, and CPU placement, since the process identifier alone would not separate it from the manager.

Scope	Selection	Counted execution
Victim CPU	`-a -C CPU`	Activity on the selected CPU
Victim process	Command and inheritance	Manager, victim thread, and inherited execution
Victim thread (diagnostics)	`-t TID -no-inherit`	The victim only
Interferent CPUs	`-a -C` on the three interferent CPUs	All activity on those cores

Table 5.3: Counter attribution in the acquisition infrastructure. Privilege filters further restrict each scope to user-space or kernel execution when specified.

Kernel instructions are attributed to the selected task whenever it is the current task during kernel execution, so a measurement of the victim thread can include scheduler work besides the code that implements its timed sleep. For this reason, CPU counts, process counts, and victim-thread counts are kept separate when the experiments are compared with the diagnostics. Each output also records its own collection interval, because collectors started separately cannot be combined into an exactly synchronized breakdown of all CPU activity. A separate 15 s `perf stat` on the victim's TID counts its kernel instructions, kernel cycles, and context switches, to check that the number of context switches per activation does not change with the load.

The interferents are measured CPU-wide on their three cores, so the counts include all their processes and threads, but also any other activity on those cores. They do not measure completed perception frames or application slowdown.

5.3.3 Counter output and interrupt snapshots

The acquisition worker requests comma-separated `perf stat` output using `-x,` and disables digit-grouping separators with `-no-big-num`. The parser matches complete event names and retains the count, event running time, and running-time percentage exposed by `perf` [32]. Unsupported events and collection errors are stored as unavailable values. Coverage, the fraction of enabled event time during which the counter actually ran, must be at least the configured 90%. For a task event, this is a different quantity from the fraction of wall time during which the periodically sleeping task executes.

Before and after an acquisition, snapshots of `/proc/interrupts` and `/proc/softirqs` retain the per-CPU interrupt and softirq counts [24]. The parser compares matching CPU columns and labels, so IRQ and IPI activity on the victim CPU can be examined alongside the other measurements. The snapshots are taken around the acquisition and can include setup or teardown activity. Their differences therefore count interrupts over the snapshot interval; they do not give handler execution time, and the interval does not coincide exactly with the PMU window.

5.4 Kernel profiling and execution tracing

5.4.1 Sampling profiles

Aggregate counters measure how much activity there is, but they do not say which functions are responsible. We therefore use `perf record` in separate diagnostic windows to sample execution on the victim CPU [30]. The CPU-clock profile samples CPU-wide on the victim CPU with the `cpu-clock` event at a requested frequency of 99 Hz, using the monotonic clock for timestamps (`-clockid mono`). Call chains are recorded with `-call-graph dwarf,8192`, which requests an 8,192-byte user-space stack dump for unwinding; kernel call chains depend on the kernel's unwinder configuration.

The resulting `perf.data` file is converted with `perf script`, keeping process and thread identifiers, CPU, timestamp, and symbols. The FlameGraph tools then collapse the stacks and draw them as an SVG [10]. Because the collapsed stacks keep the identifiers, the victim can be told apart from the manager and from other execution on the core.

In a flame graph, frame width represents the number of samples containing that frame, including samples in its descendants; the horizontal axis does not represent chronological time [9]. Similarly, inclusive percentages in `perf report` overlap between callers and callees [31]. These views locate sampled activity within call chains, but their widths and percentages are neither function-call counts nor instruction counts. We use

them to choose which functions to observe more closely, and their latency distributions are kept separate from the main counter acquisitions.

Diagnostics restricted to the victim also use instruction-event sampling, with `instructions:k` and a period of 100,000 counted instructions. These samples are unreliable in scheduler code: the sampled instruction pointer can lag behind the event that triggered collection, and interrupt masking can delay delivery and leave parts of the execution underrepresented [46]. A function missing from a sampled profile may still have executed, and a sampled percentage alone cannot quantify that function's share of the victim's total kernel instructions.

5.4.2 Function tracing and execution context

The kernel configuration enables the ftrace function tracer, which records function entries and their callers [44] and so allows ordered kernel paths to be compared beyond what the sampled stacks show. The function-graph tracer is not enabled, so the trace contains function entries only, without paired returns or durations.

A function trace of the victim uses a dedicated tracefs instance, `set_ftrace_pid` to select the victim's TID, and `tracing_cpumask` to select the victim CPU. The `tracing_on` control bounds recording into the buffer, which is read after collection. Before comparing two traces, per-CPU buffer statistics are checked for overwritten or dropped events, and the output is normalized by removing timestamps and process prefixes that differ between runs, while preserving the function and caller sequence [44]. Comparing the sequences then shows additional calls, and normalized call frequencies summarize recurring differences. Neither measures the number of retired instructions.

For targeted execution-context checks, we use extended Berkeley Packet Filter (eBPF) programs attached to kprobes on the timed-sleep entry and selected scheduler functions [14, 21]. In the recorded five-second diagnostics, the filter selects the victim's TID and the program counts entries into `newidle_balance`, `load_balance`, and `can_migrate_task`, retaining the current thread and CPU. A kernel stack is captured at the first selected `load_balance` entry to establish its calling context. These observations tie a function to the thread executing it; its instruction cost comes from the separate PMU measurement described below.

The tracing work runs on the CPU that reaches the instrumented kernel path, even when the control process runs on CPU4–CPU5. Recording and probe handling add execution to that path, so tracing is done in diagnostic windows kept apart from the main latency comparison.

5.5 Targeted scheduler measurements

5.5.1 Observing branch decisions

Function entries show which path the scheduler took, but not which condition made it take that path. We wrote targeted probes for `newidle_balance`, the scheduler path used when a runqueue becomes idle, to observe the conditions that decide whether it

proceeds to load balancing. The instrumentation is based on the source and disassembly of the running `5.15.148-rt-tegra` kernel, whose image is matched against the build notes exposed by the board. This check is needed because internal function offsets, and the register that holds a given operand, depend on the compiled kernel image.

The tracefs kprobe interface supports entry probes, probes at a symbol offset, and return probes, with register operands included in the event [14]. We place probes at the verified conditional instructions and read operands already loaded by the kernel. The selected observations cover:

- pending wake-ups and the CPU-active condition;
- the estimated idle interval and the migration-cost threshold;
- the `overload` operand read from the scheduler *root domain*;
- the accumulated balancing cost and the `SD_BALANCE_NEWIDLE` flag for a *scheduling domain*;
- entries and returns of `newidle_balance`, and entries into `load_balance` and the timed-sleep system call.

The five-second acquisition selects CPU0 and the victim's TID, using event filters in a dedicated tracefs instance [18]. The analyzer pairs entries with returns, classifies the observed exit reason, and assigns nested `load_balance` entries to their enclosing invocation. Buffer-loss statistics and missed-probe counts are saved with the trace, so an incomplete acquisition can be told apart from a path that was genuinely not taken.

A second set of probes follows each long path down to the candidate tasks. It records the source runqueue chosen by `load_balance`, every task examined by `can_migrate_task`, the branch that rejects it (affinity, already running, cache-hot, or per-CPU kernel thread), the tasks detached and attached, and the next task picked, so that each search can be followed to its outcome. The capture lasts about 3 s per scenario, with the victim on CPU0 and CPU affinity.

5.5.2 Tracing shared scheduler state

The `overload` operand belongs to a root-domain structure shared by several CPUs. Observing only the victim's core would capture a reader of that state and could miss the CPUs that write it. A second diagnostic therefore records the victim's reads together with the selected scheduler operations that update the flag, and the runnable-load information those operations use. Read events keep the victim filter, whereas writer events are collected on all online CPUs and matched through the root-domain address.

For the verified kernel image, the event definitions retain the CPU, task identifiers, scheduling policy, runnable counts, and relevant state operands at the selected enqueue and balancing sites. With these, the trace can relate activity elsewhere in the scheduling domain to the state the victim's path reads later. The recorded process context identifies the thread executing an operation, but it does not list every task in the affected runqueue. Likewise, timestamp order and a matching structure address make it possible to analyze updates to the shared state, though not to pair every read with one specific earlier store.

5.5.3 Direct measurement of instruction cost

These diagnostics guided the analysis; Chapter 6 reports only the results needed for its conclusions. To measure the instruction cost of the selected scheduler segment, we implemented a collector that combines two kprobes with direct PMU reads. The first probe is placed at `newidle_balance.constprop.0+156`, where register `x0` contains the loaded `overload` operand; the second probe records the return from the same function. The measured interval therefore runs from this check to the function's return; earlier instructions, and invocations that exit before reaching the check, are not included.

The collector opens two pinned kernel-only instruction events through `perf_event_open`: a victim-thread event for the total instruction count in the window, and a CPU0 event for the boundary reads [26]. The CPU event is accessed by the eBPF helper `bpf_perf_event_read_value`, which returns the count together with enabled and running times [21]. The probes select the victim's TID, and the measured kernel segment executes with interrupts and preemption disabled, without a task switch. Under these conditions, the difference between the two CPU counter reads belongs to the victim's segment, even though the counter itself is CPU-wide. The boundary reads need no PMU sampling interrupt, so the segment is observed directly.

For invocation j, the measured instruction count is

$$\Delta I_j = P_j^{\text{return}} - P_j^{\text{check}}, \tag{5.1}$$

where P_j^{check} and P_j^{return} are the readings at the two boundaries. Each invocation is assigned to a class c according to whether the observed `overload` operand is zero or nonzero. For the N_c matched invocations in that class, the collector reports the sum, minimum, maximum, and mean

$$\overline{I}_c = \frac{1}{N_c} \sum_{j \in c} \Delta I_j. \tag{5.2}$$

A nonzero operand lets the scheduler perform further checks, so the class reflects the observed branch condition; invocations in the same class can still follow different call sequences.

An explicit control map arms the eBPF programs only after the PMU events have been enabled. At the end of collection, new entries are disarmed and any pending return is allowed to complete before the counters are disabled. The output records read failures, nested or unmatched invocations, nonmonotonic counts, missed probes, and counter coverage. A segment measurement is valid only with matching boundaries and events that ran continuously, with equal enabled and running times, so the instrumented instruction counts need no multiplexing correction.

The diagnostic comparison uses three pairs of nominally 15-second windows per baseline or tkHPick condition. Each pair contains a `count` window measuring the victim's kernel instructions without probes and a `measure` window collecting both the victim's total and the instrumented segments. The order is reversed in the second pair, and the same victim stays active throughout. Rates use the recorded acquisition duration, and the victim-thread totals are kept separate from the process totals of the main campaign.

The segment count includes descendant functions and part of the probe execution between the two reads. It is an inclusive instruction cost under instrumentation, and adding the count of a nested function to it would count some execution twice. Comparing the windows with and without probes checks the overall perturbation, but their difference also contains run-to-run variation and cannot be subtracted as an exact per-invocation overhead. The segment count therefore shows what a path costs under instrumentation, not how much it contributes to the victim without probes.

5.6 Per-phase measurement of the wake-up

To locate the extra latency within a wake-up, an eBPF program splits every iteration of `cyclictest` on the isolated CPU3 into phases delimited by kernel events (Figure 6.11). A probe is attached to each event: `power:cpu_idle` for the exit from and the entry into idle, `irq:irq_handler_entry` for the timer interrupt, `timer:hrtimer_expire_entry` for the expiry of the `cyclictest` timer, `sched:sched_switch` for the switches to `cyclictest` and to the idle task, and `raw_syscalls:sys_exit` and `raw_syscalls:sys_enter` for the return to user space and the next `clock_nanosleep` [18]. A further kprobe on `hrtimer_start_range_ns` records when the timer of `cyclictest` is due, so that the exit from idle that starts an iteration can be recognised [14].

At each event the program reads the time and five PMU counters of CPU3, `cpu_cycles` (`0x11`), `inst_retired` (`0x08`), `stall_frontend` (`0x23`), `stall_backend` (`0x24`), and `stall_backend_mem` (`0x4005`) [21, 1], and adds the difference from the previous event to the phase that has just ended. Activity on CPU3 outside the iteration of `cyclictest` is accumulated in a separate phase. The phases account for all the activity of CPU3: their sum matches the totals that the counters report directly, and more than 99.9% of the iterations are recorded completely.

The probes have a cost: each of them adds work on CPU3 and lengthens the phases. Every measurement is therefore repeated at three levels: without probes, which gives the latency of the same runs without instrumentation; with probes only at the edges of the latency window; and with all the probes. Each level consists of two 30 s runs per scenario, with the victim in the cpuset partition of CPU3 and the same `cyclictest` options as in the campaigns except the duration. Phase durations are compared only between the baseline and the loaded scenarios measured at the same level.

During these runs, the over-current event counter of the power monitor (`oc3_event_cnt`) is read once per second. In further runs with the probes at the edges of the window, the loader also records every 10 ms the time that CPU3 spends outside idle, that is, in all the phases except the sleep, the cycles counted in those phases, and the same counter. The time lost to throttling in an interval is the part of this time not covered by the counted cycles at the nominal frequency of 1.728 GHz, that is, the time outside idle minus the counted cycles divided by 1.728 GHz. The idle time is left out because the cycle counter does not advance while the core is idle (Section 2.5). These series relate the over-current events to the time lost to throttling (Figure 6.15). In the same runs, the loader also reads the input power of the module

(`VDD_IN`) from the power monitor and the frequency that the kernel requests for the CPUs.

The program recognises the measurement thread of `cyclictest` by its name, so it works unchanged with the H^3 Vision victim, whose work falls in the phase after the return to user space, outside the latency window. For the H^3 Vision victim, and for the minimal task measured in the same session, the per-phase measurement is taken without probes and with all the probes; the latency reported in Chapter 7 comes from the campaign, and the per-phase runs give only the shares of its components.

6. Results: a minimal real-time task

Chapter 4 defined two victims: the measurement thread of the original `cyclictest`, the simplest real-time task, which wakes up, reads the clock and goes back to sleep, and the H^3 Vision victim, which after every wake-up reads and writes as much data as a high-speed vision system for robots handles in one frame. With CPU affinity only and the interferents on the other cores of the cluster, the mean latency of the H^3 Vision victim on CPU0 grows by 45% with MEMINTERF and by 40% with tkHPick, and its maximum reaches 420 and 476 μs, almost half of its 1 ms period (Figure 6.1).

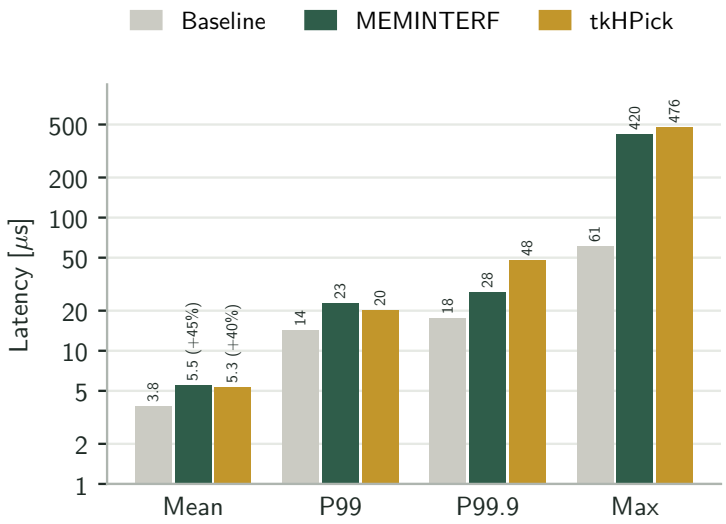


Figure 6.1: Wake-up latency of the H^3 Vision victim on CPU0 with CPU affinity only. Each bar is the mean of three 300 s repetitions; for the maximum, the bar is the largest value observed. Same logarithmic scale as the other latency figures of this chapter.

On CPU0, however, the H^3 Vision victim is not slower than the minimal task in the same configuration (Section 6.1): its mean latency is 3.8 μs in the baseline, as for the minimal task, and 5.5 and 5.3 μs with the interferents, against 6.2 and 5.9 μs. Here the effect of its own memory footprint does not emerge, probably because causes that do not depend on what the victim does dominate the latency.

Several causes add up in this latency. To separate them, this chapter starts from the minimal task, whose own work touches almost no memory, and Chapter 7 returns to the H^3 Vision victim.

The analysis starts from the simpler configuration, CPU affinity, and measures how much the wake-up latency of the victim grows when the interferents run on the other cores of its cluster. With tkHPick, the hardware counters show that the victim also executes more instructions, and the kernel traces show a longer search in the scheduler when the victim goes to sleep, which couples the victim CPU to the load of the other cores. The cpuset partition removes this coupling, but CPU0 keeps receiving the peripheral interrupts. On the isolated CPU3 only a small increase in latency remains, and the last part of the chapter breaks it down into memory contention and frequency throttling.

In all the figures, grey, green, and brass denote the baseline, MEMINTERF, and tkHPick. Unless stated otherwise, each bar is the mean over the three repetitions of a condition, and every scenario is compared with the baseline measured on the same CPU and in the same configuration.

6.1 tkHPick increases the wake-up latency

The first configuration is CPU affinity alone (Section 4.3): the victim is pinned to CPU0 or CPU3, and the interferents to the other three cores of its cluster (Table 4.2). Figure 6.2 reports the wake-up latency of Equation (4.1) through its mean, its percentiles, and its maximum, computed as described in Section 4.5.

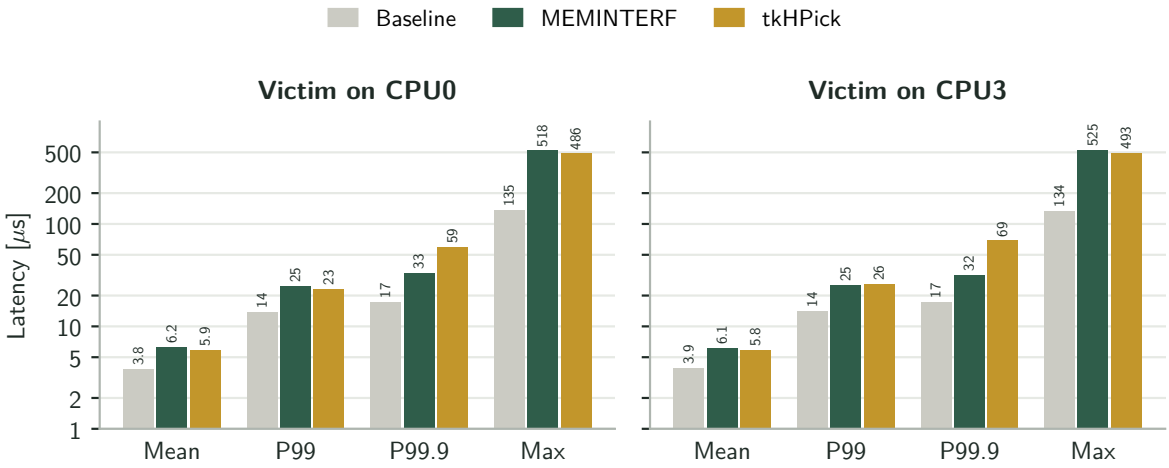


Figure 6.2: Wake-up latency with CPU affinity only. Each bar is the mean of three 300 s repetitions; for the maximum, the bar is the largest value observed. The logarithmic scale is the same in all the latency figures of this chapter.

The mean latency grows by about 1.5 times with tkHPick and by about 1.6 times with MEMINTERF, and every percentile grows with it. The maxima reach about 500 μs, half of the 1 ms period, but they are rare: latencies above 200 μs occur about 140 to 160 times per million activations with tkHPick and 260 to 340 times with MEMINTERF, and never in the baseline. A latency of hundreds of microseconds takes up a large part of the period and can be enough to make a real-time task with a tight deadline miss it.

Both interferents therefore increase the latency of the victim by similar amounts.

6.2 With tkHPick, the victim executes more instructions

The two interferents increase the latency by similar amounts, so the next question is whether they do it in the same way. A first check is whether the victim executes the

same work in every scenario: the hardware counters count the instructions retired by the victim process during each run.

`cyclictest` records one latency value for every activation, while the hardware counters accumulate their counts over the whole run. Dividing a counter by the number of activations of the same run gives a value per wake-up, and the value reported is the mean over the repetitions. For an event counted C_r times in repetition r, which contains N_r activations, the reported value over R repetitions is

$$\overline{c} = \frac{1}{R} \sum_{r=1}^{R} \frac{C_r}{N_r}. \tag{6.1}$$

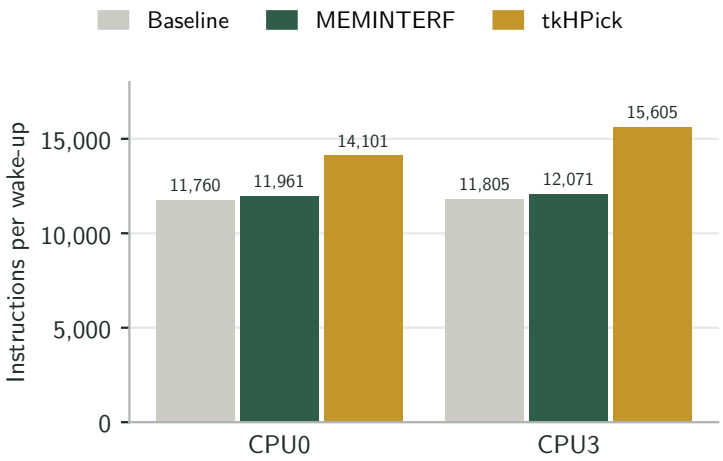


Figure 6.3: Instructions retired by the victim process per wake-up, with CPU affinity only.

Figure 6.3 shows that with MEMINTERF the number of instructions stays almost unchanged, about 2% more than in the baseline. With tkHPick it grows by about 20% on CPU0 and by about 32% on CPU3, that is, 2,300 and 3,800 more instructions for every wake-up. The extra instructions are not due to more activations or more context switches: every 300 s run counts the same 300,000 activations, and a dedicated measurement on CPU0 counts one context switch per activation both in the baseline and with tkHPick.

With MEMINTERF, therefore, the latency grows while the victim executes the same work: only tkHPick changes what the victim executes. The code of `cyclictest` and its period have not changed, and tkHPick runs on other cores. So where do the extra instructions come from?

6.3 The kernel takes a longer path when the victim goes to sleep

A kernel path is the sequence of function calls that the kernel executes to carry out an operation, and the path taken for the same operation can change with the load running

on the rest of the system. This section shows that this is what happens to the victim with tkHPick.

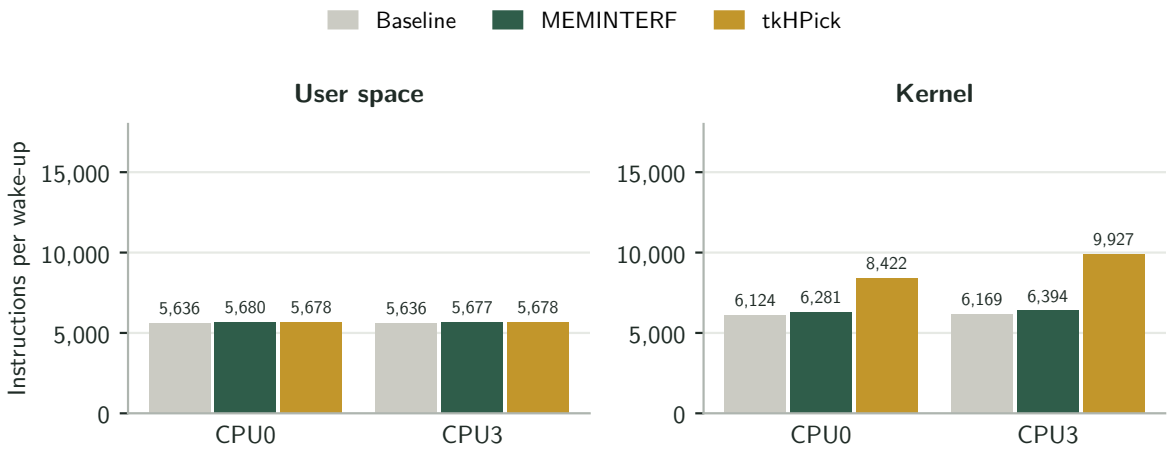


Figure 6.4: Instructions retired by the victim process per wake-up, split into user space and kernel, with CPU affinity only. The user-space counts include the fixed work of `cyclictest` at exit.

As Figure 6.4 shows, the user-space instructions, which correspond to the code of `cyclictest`, stay the same in every scenario. The kernel instructions grow by 38% on CPU0 and by 61% on CPU3 with tkHPick, and account for almost all of the increase seen in Section 6.2. With MEMINTERF they stay close to the baseline, about 3% more on CPU0 and 4% more on CPU3, so the diagnostic measurements that follow compare only the baseline and tkHPick.

6.3.1 The long path

The function tracer, restricted to the victim thread, shows which calls are added. With tkHPick every sleep executes more functions, and therefore more kernel code, and the new ones belong to load balancing: `load_balance`, `find_busiest_group`, and `can_migrate_task`, which are almost absent in the baseline.

Baseline: short path

```
 75 balance_fair
 76 newidle_balance
 77 __rcu_read_lock
 78 __msecs_to_jiffies
 79 __rcu_read_unlock
 80 put_prev_task_rt
```

tkHPick: long path

```
 85 balance_fair
 86 newidle_balance
 87 raw_spin_rq_unlock
 88 update_blocked_averages
     [... 26 load/frequency updates ...]
115 load_balance
116 find_busiest_group
     [... rq lock, idle_cpu ...]
123 kthread_is_per_cpu
124 can_migrate_task      <- rejected
125 kthread_is_per_cpu
126 can_migrate_task      <- rejected
128 find_busiest_group
132 load_balance          (next domain)
133 find_busiest_group
     [...]
141 put_prev_task_rt
```

Figure 6.5: Function-tracer excerpts at the same point of the victim's sleep path (original line numbers; `[...]` marks omitted lines). Between `newidle_balance` and `put_prev_task_rt` the baseline executes 4 functions, the tkHPick case 55.

The scheduler itself calls these load-balancing functions, on the path that puts the victim to sleep: the system call `clock_nanosleep` enters `schedule`, which reaches `newidle_balance` through `balance_fair` and from there calls `load_balance`. A kernel stack captured with eBPF at the entry of `load_balance` confirms this chain, with `cyclictest` as the running task.

6.3.2 Why the scheduler takes the long path

When the victim goes to sleep and no other task is ready on its CPU, the scheduler runs *newly-idle balancing*, described in Section 2.3. If the shared indicator `rd->overload` says that no CPU of the *root domain* is overloaded, `newidle_balance` returns at once: this is the short path. Otherwise it searches for a task to pull, calling `load_balance` on each scheduling domain of the CPU: this is the long path. On this platform the domains have two levels, the four cores of the victim's cluster and all six cores, so the long path almost always calls `load_balance` twice.

The search runs when the victim goes to sleep, before the context switch, while the victim is still the running task, so its instructions are attributed to `cyclictest`. It also runs after `cyclictest` has recorded the latency of the current activation, so it does not lengthen the measured latency directly. What it shows is that the load of the other cores reaches the kernel of the victim CPU through the scheduler.

The scheduler measurements below were taken with the victim on CPU0. MEMINTERF runs one single-threaded process on each of the other three cores, and the process never sleeps: each of those CPUs is busy with a single task, so it is rarely overloaded, which agrees with the kernel instructions of Figure 6.4 staying at the baseline level. tkHPick instead is made of many threads that keep waking up and going to sleep on the same three cores, so one of them often has more than one runnable task. As Figure 6.6 shows, in the baseline the long path is taken once in a five-second capture, about 5,000 invocations, while with tkHPick two invocations out of three take it.

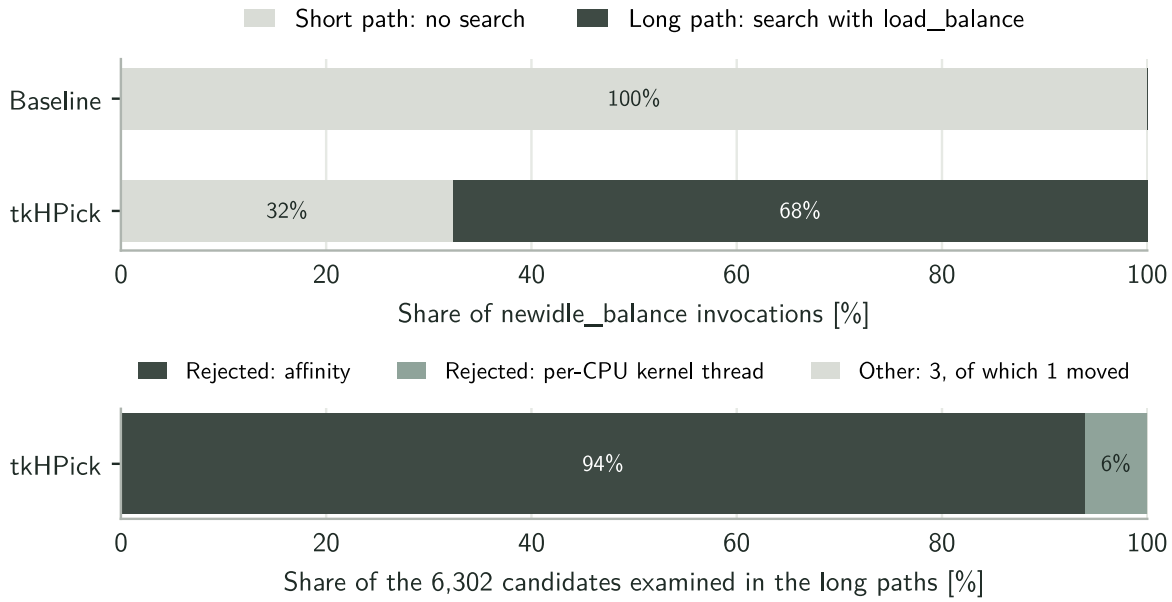


Figure 6.6: Newly-idle balancing in the victim's context, with the victim on CPU0. Top: outcome of the `newidle_balance` invocations; the short path also includes the early exits taken when the CPU is expected to stay idle for less than the cost of a migration. Bottom: outcome of the candidate tasks examined by `can_migrate_task` during the long paths with tkHPick, in a separate capture.

The search almost never finds a task to move: in a separate capture, only one of about 2,000 long paths moved a task, and in 99.9% of them the CPU then became idle as in the baseline. Most of the candidates it examines are threads of tkHPick, whose affinity allows them to run only on the other three cores of the cluster, and the kernel respects this constraint: `can_migrate_task` rejects 94% of the candidates because of their affinity (Figure 6.6, bottom, and Figure 6.5). Almost all the others are per-CPU kernel threads, which cannot move to another CPU. The scheduler is behaving as designed: it tries to keep every CPU busy and honours the placement chosen for each task, and the indicator cannot tell it in advance that none of the waiting tasks may be moved (Section 2.3). With CPU affinity only, the victim CPU shares the root domain with the cores of tkHPick, which is why the indicator is so often set when the victim goes to sleep.

6.4 A cpuset partition removes the scheduler coupling

Section 6.3 showed that, with CPU affinity only, the scheduler couples the victim CPU to the load of the other cores of its root domain. CPU affinity restricts only where the interferents run: the victim CPU stays in the shared root domain, where it keeps seeing the load of the other cores, and can still host any task that has no affinity of its own.

The second configuration places the victim CPU in a *cgroup v2* cpuset that contains

only that CPU, configured as a *partition root* [12]. When the partition is created, the kernel takes this CPU away from the CPUs that every other cgroup may use, so no other user task can be scheduled on it, and builds a separate set of scheduling domains for it [22]. The CPU then gets a root domain of its own: the load on the other cores no longer counts when the victim CPU decides whether to look for work, and a domain with a single CPU has no other CPU to balance with, so `newidle_balance` has nothing to search. Some kernel activity still runs there: per-CPU kernel threads such as `ksoftirqd` and `rcuc`, the scheduler tick, the local timer interrupt, occasional IPIs, and the peripheral interrupts routed to that CPU, which cpuset does not control. User space, instead, contains only the victim.

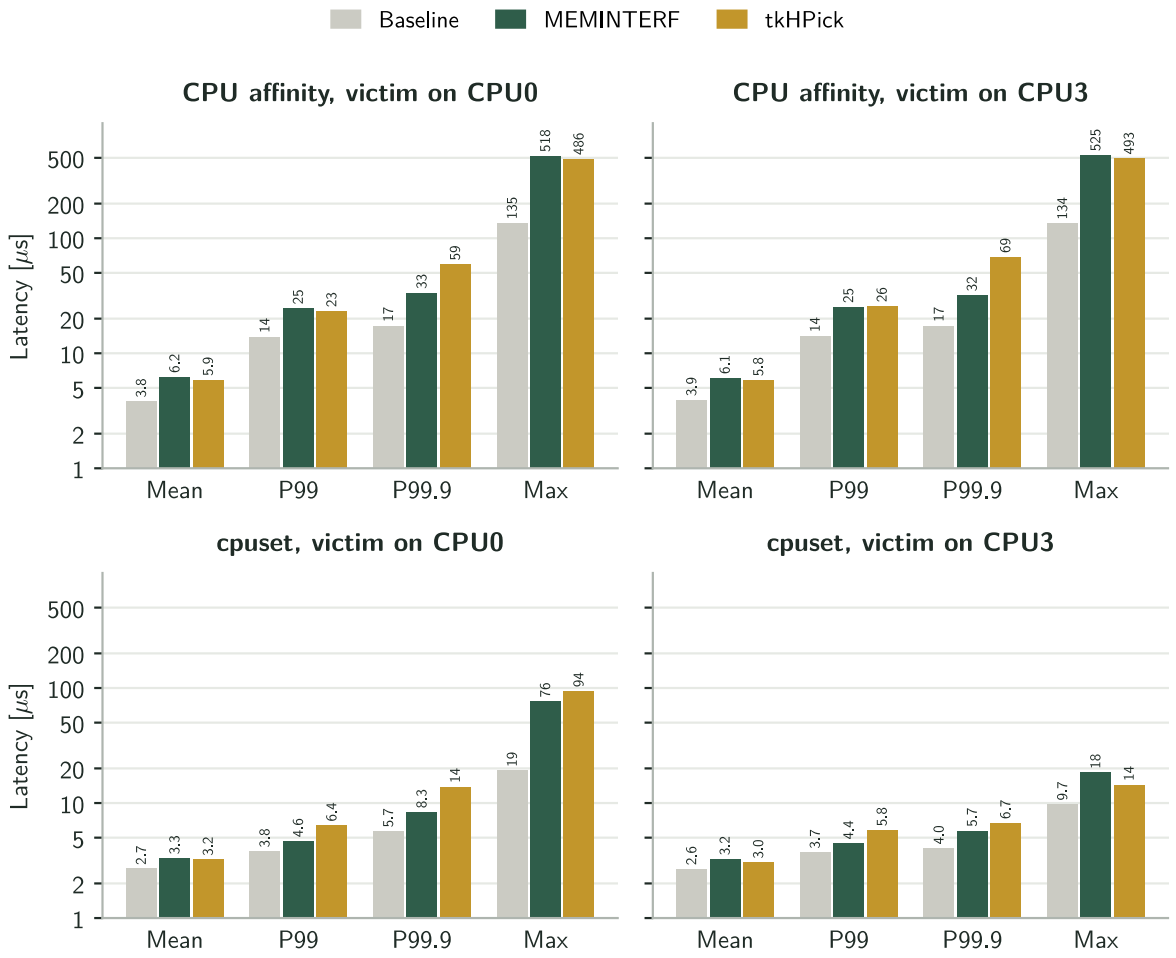


Figure 6.7: Wake-up latency with CPU affinity only (top, the same data as Figure 6.2) and with the cpuset partition (bottom), on the same scale. Each bar is the mean of three 300 s repetitions; for the maximum, the bar is the largest value observed.

Figure 6.7 compares the two configurations. With the partition the latency drops in every scenario, the baseline included: the mean goes from 3.8–3.9 μs to about 2.7 μs in the baseline and from about 6 μs to 3.0–3.3 μs with the interferents, and the maxima fall from about 500 μs to 76–94 μs on CPU0 and to 14–18 μs on CPU3. The latency drops with MEMINTERF and in the baseline as well, so the partition changes more than the extra work of tkHPick: it also keeps every other task off the victim CPU and stops the scheduler events that the other cores direct at it.

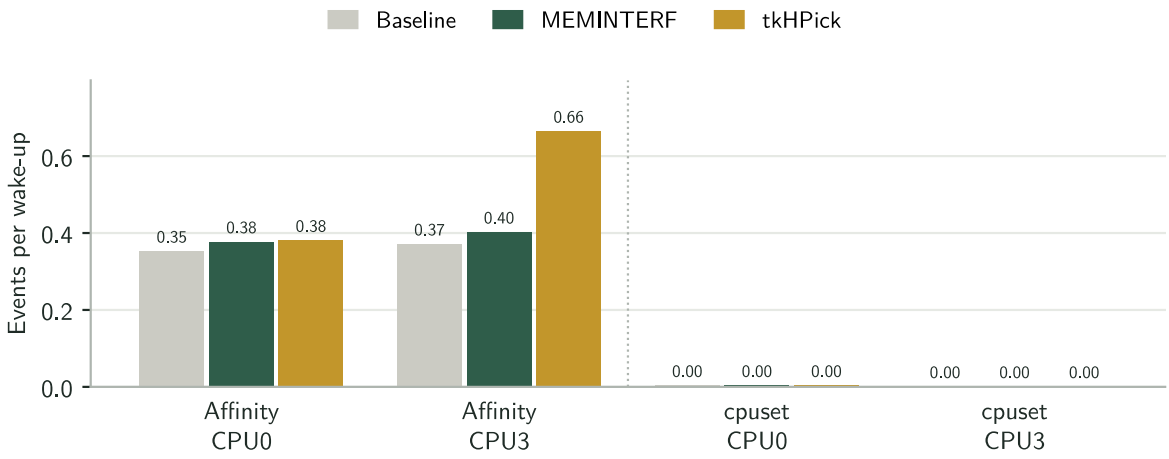


Figure 6.8: Rescheduling IPIs and `SCHED` softirqs reaching the victim CPU per wake-up, with CPU affinity only and with the cpuset partition.

With CPU affinity only, the victim CPU keeps receiving rescheduling IPIs, with which another core asks it to run the scheduler, and `SCHED` softirqs, in which it balances its load with the other cores. Together they arrive between once every three wake-ups and twice every three (Figure 6.8). With the partition they practically disappear, on both CPUs.

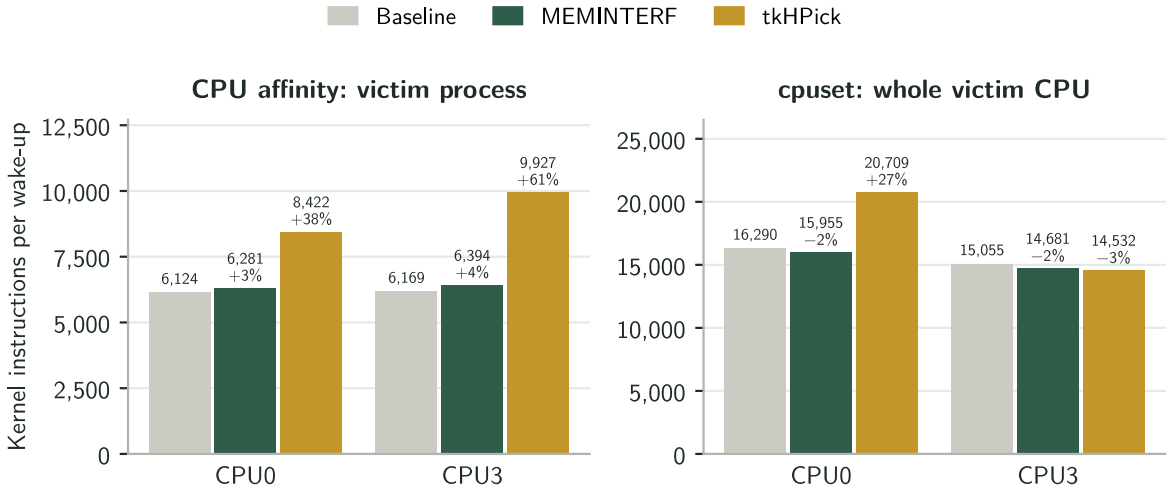


Figure 6.9: Kernel instructions per wake-up. Left: victim process, with CPU affinity only. Right: whole victim CPU, with the cpuset partition. The two panels count different scopes and have different scales; the percentages compare each scenario with the baseline of the same CPU and panel.

Figure 6.9 shows the kernel instructions per wake-up in the two configurations. The left panel repeats the counts of the victim process of Figure 6.4. With the partition the victim is the only user task on its CPU, and the counters were collected for the whole CPU: they also include the kernel work done there outside the victim process, such as interrupts, the per-CPU kernel threads, and the idle task. For this reason the values on the right are higher, and the comparison is between the scenarios within each

panel. With the partition, on CPU3 the increase disappears: with tkHPick the kernel instructions are 3% below the baseline, and with MEMINTERF 2% below. On CPU0, instead, tkHPick still adds 27%, which the next section explains.

On CPU3, therefore, the victim CPU executes the same kernel work with and without the interferents. The increase in latency that remains on CPU3 with the partition does not come from extra work, and Section 6.6 looks for its origin.

6.5 Peripheral interrupts keep CPU0 exposed

With the interrupt policy of the platform, the peripheral interrupts are handled by CPU0 (Section 4.3), and the cpuset partition does not change where they are handled. Figure 6.10 counts them per wake-up with the partition. With tkHPick, CPU0 receives more than five times as many as in the baseline, mostly from the GPU, while with MEMINTERF they stay at the baseline level and CPU3 receives practically none. Their handlers run on CPU0 and their instructions are counted in its kernel work, so they contribute to the 27% that remains on CPU0 with the partition.

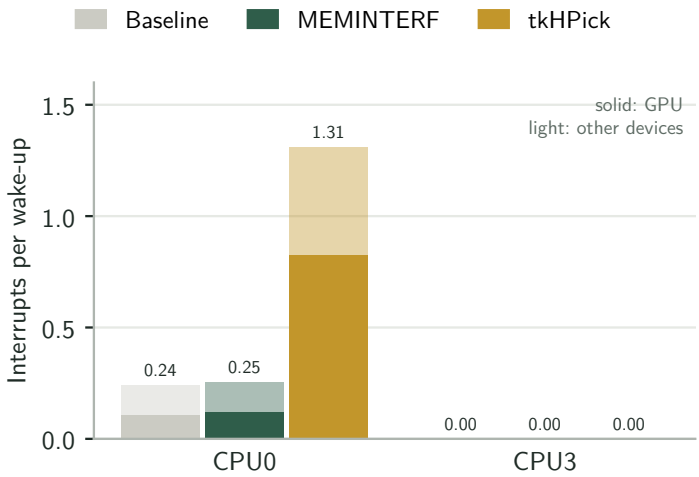


Figure 6.10: Peripheral interrupts reaching the victim CPU per wake-up, with the cpuset partition, split between the GPU (`host1x` and `gk20a`) and the other devices.

The partition cannot remove this interference, because it acts on scheduling only (Section 2.3). For this reason the rest of the chapter considers only CPU3, isolated with the partition, with the interferents on the other three cores of its cluster.

6.6 What remains on the isolated CPU3

With the partition, the coupling through the scheduler is gone, and on CPU3 the victim CPU executes the same kernel work as in the baseline (Section 6.4). The latency still grows, but only a little: in Figure 6.7 the mean latency on CPU3 rises from 2.65 μs in the baseline to 3.03 μs with tkHPick and 3.24 μs with MEMINTERF, that is, by

0.4 and 0.6 μs, less with tkHPick than with MEMINTERF. As a share of the baseline mean, the increase is 14.3% with tkHPick and 22.1% with MEMINTERF. The rest of this section breaks it down.

6.6.1 Measuring the phases of a wake-up

`cyclictest` reports only the total latency of each activation. To see in which part of the wake-up the extra time falls, and what CPU3 waits for during that time, every iteration of `cyclictest` on CPU3 is split into phases (Figure 6.11), measured with eBPF probes on kernel events as described in Section 5.6.

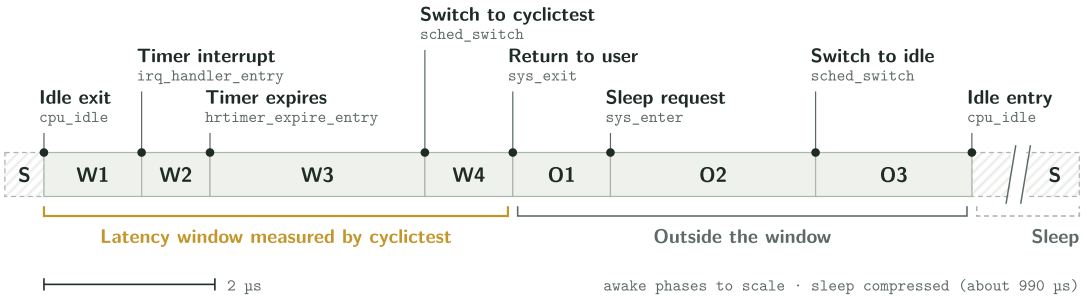


Figure 6.11: One iteration of `cyclictest` on CPU3. Above each phase boundary are the kernel event used as a probe and the tracepoint it is read from (from the `power`, `irq`, `timer`, `sched` and `raw_syscalls` categories). The awake phases are drawn in proportion to their durations in the baseline, measured with all the probes; the sleep is compressed.

Four phases cover the part of the latency window that the kernel executes:

- W1, from the exit from idle to the timer interrupt;
- W2, from the interrupt to the expiry of the `cyclictest` timer, which wakes the thread up;
- W3, from the wake-up to the context switch to `cyclictest`;
- W4, from the context switch to the return to user space.

Two short stretches of the latency are not instrumented: the hardware wake-up of the core before the exit from idle, and the clock read in user space after the return. The other phases, from the return to user space to the next entry into idle, fall outside the window. For each phase the probes give its duration and, from the hardware counters of CPU3, the cycles, the cycles stalled waiting for data from memory, and those stalled waiting for instructions (frontend stalls). Since the probes lengthen the phases, durations are only compared between the baseline and the loaded scenarios.

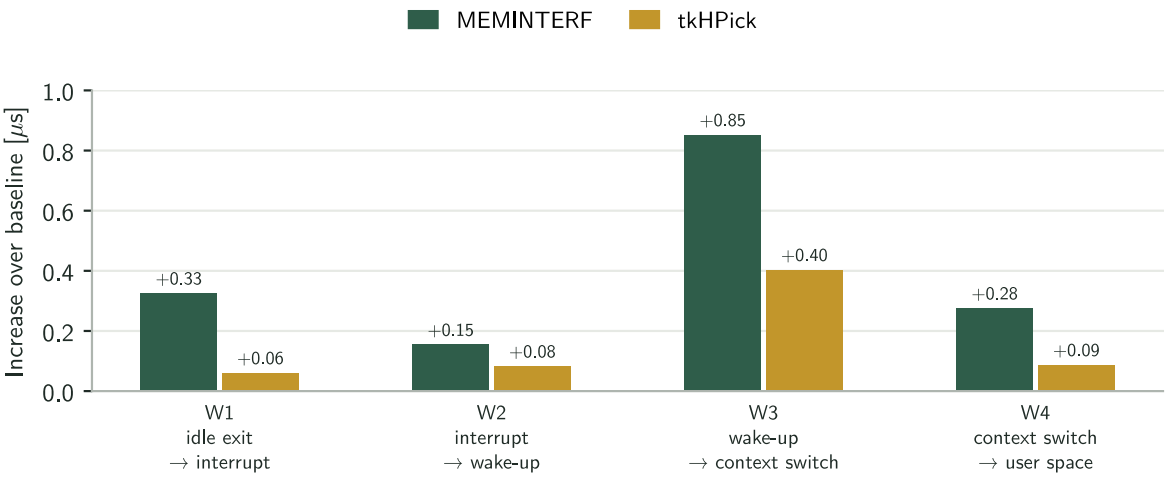


Figure 6.12: Increase of the duration of each phase of the latency window on CPU3 with the cpuset partition, with respect to the baseline, measured with all the probes.

Figure 6.12 shows how much each phase of the window grows with the interferents. With both, most of the increase falls in W3, between the wake-up and the context switch to `cyclictest`: 0.40 of 0.63 μs with tkHPick and 0.85 of 1.61 μs with MEMINTERF. With MEMINTERF the other phases grow as well, while with tkHPick they change little. These increases are measured with all the probes active and are larger than those without instrumentation; what matters here is how they are distributed among the phases.

6.6.2 What the increase is made of

For each phase, the counters split the elapsed time into the cycles spent waiting for data from memory, the cycles spent waiting for instructions, the other stalls, and the *time lost to throttling*, that is, the part of the elapsed time not covered by the cycles counted at the nominal frequency of 1.728 GHz. Section 6.6.4 shows that this time is due to throttling. Figure 6.13 applies the shares measured in the latency window to the increase of the mean latency without instrumentation, expressed as a share of the baseline mean, so that the parts add up to the 14.3% and 22.1% given above.

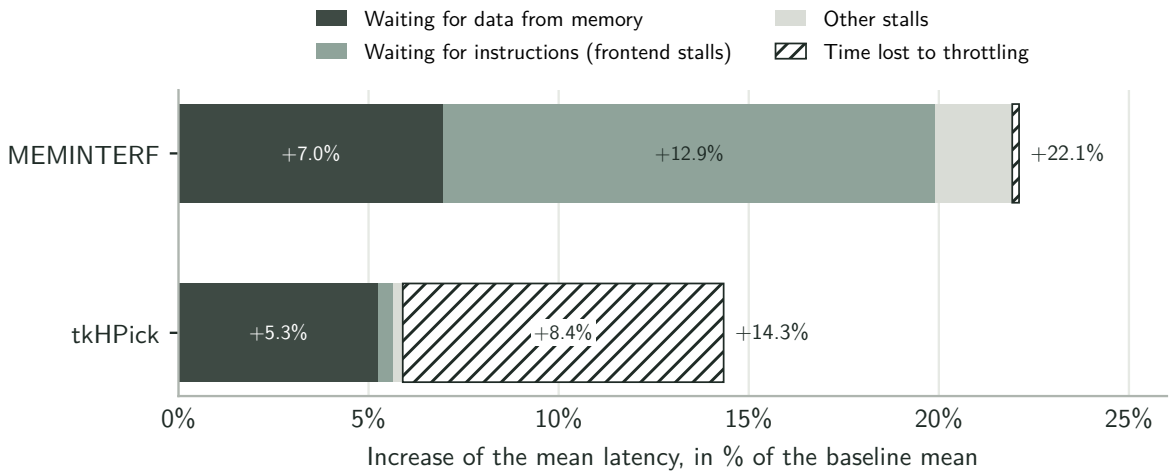


Figure 6.13: Increase of the mean latency on CPU3 with the cpuset partition, split into its components and expressed as a share of the mean latency of the baseline. The shares are measured inside the latency window with the per-phase probes and applied to the increase of the mean latency without instrumentation, so the split is an estimate. Other stalls also include the cycles spent executing instructions.

With tkHPick, waiting for data from memory accounts for about 5 of the 14 percentage points, about 0.14 μs. With all the probes it falls entirely in W3, between the wake-up and the context switch.

Most of the remaining points, about 8.4, are time lost to throttling: in every phase, inside and outside the window, CPU3 receives fewer cycles than it should at its nominal frequency, so the same code takes longer.

With MEMINTERF the time lost to throttling is practically zero, and the whole increase is made of counted cycles. Of its 22 points, about 7 come from waiting for data from memory, about 13 from waiting for instructions (frontend stalls), and 2 from other stalls.

6.6.3 Why the memory wait is small in absolute terms

Both interferents load the memory heavily, yet waiting for data and instructions adds at most about half a microsecond to the wake-up (Figure 6.13). The reason is that the work of `cyclictest` stays almost entirely inside the private caches of CPU3. In every scenario, more than 99.4% of the memory accesses of each iteration, for data and instructions, are served by the L1 and L2 caches of the core (Figure 6.14). Each iteration runs a short loop of a few hundred user-space instructions that reads the clock, computes the latency and updates its statistics, touching very little data.

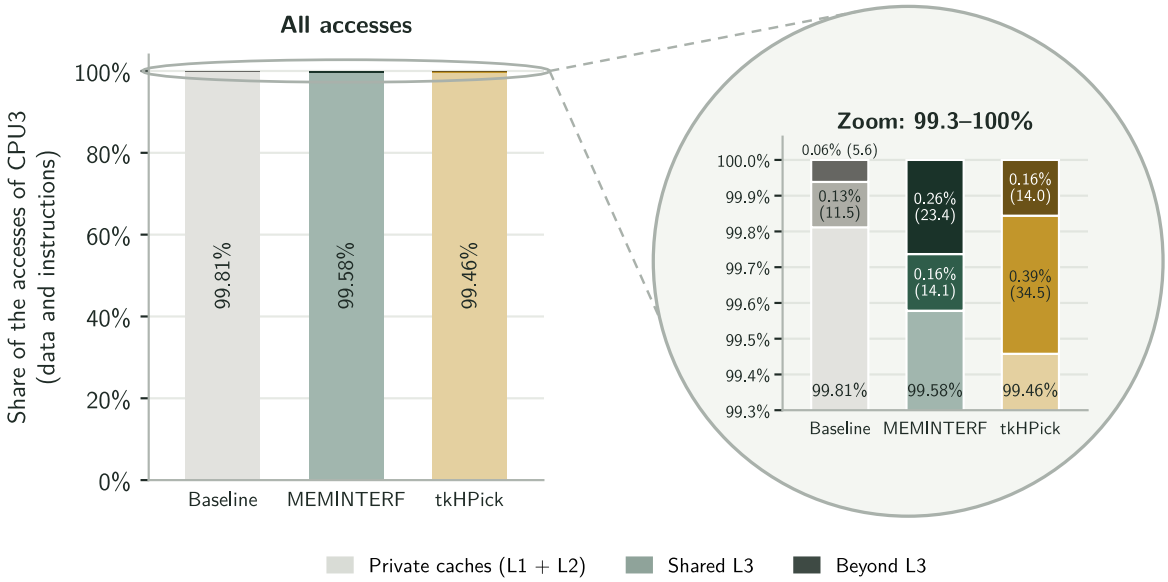


Figure 6.14: Where the memory accesses of CPU3 are served, with the cpuset partition. The lens enlarges the top of the bars, from 99.3 to 100%; the numbers in brackets are accesses per iteration. Accesses beyond the L3 are the L3 refills, served by the system cache or by DRAM.

Only a few tens of accesses per iteration, out of about 9,000, leave the private caches and reach the part of the hierarchy that the interferents also use. With the interferents more of them leave the private caches, from about 17 per iteration in the baseline to 37 with MEMINTERF and 48 with tkHPick, and more of them also miss in the L3, from about 6 to 23 and 14. The interferents share the L3 cache of the cluster with the victim (Table 4.1) and evict its lines: an access that would have been served by the L3 is then served beyond it, by the system cache or by DRAM, where it also competes with the traffic of the interferents. These counters do not show why more accesses leave the private caches, nor why, with MEMINTERF, waiting for instructions grows by about 13 points although the instruction accesses too are served almost entirely by the private caches. Since so few accesses are involved, the wait they add stays small.

6.6.4 Over-current throttling

The other part of the increase with tkHPick, about 8 of its 14 points, is time lost to throttling, and it comes from the power supply rather than from memory. The Jetson Orin Nano limits the power drawn by the module: when the current on its input rail (`VDD_IN`) exceeds the configured limit, an over-current (OC) alarm is raised and the BPMP firmware throttles the clocks of the CPUs and of the GPU in hardware [36], as introduced in Section 2.4. When tkHPick runs its inference on the GPU, the module draws about 14 W, against 11 W with MEMINTERF and 7 W at rest, and the OC3 alarm fires about 80 times per second, as counted by the over-current event counter of the power monitor (`oc3_event_cnt`). With MEMINTERF and in the baseline it never fires. The throttling also slows down the isolated CPU3, while the operating system keeps requesting the nominal frequency.

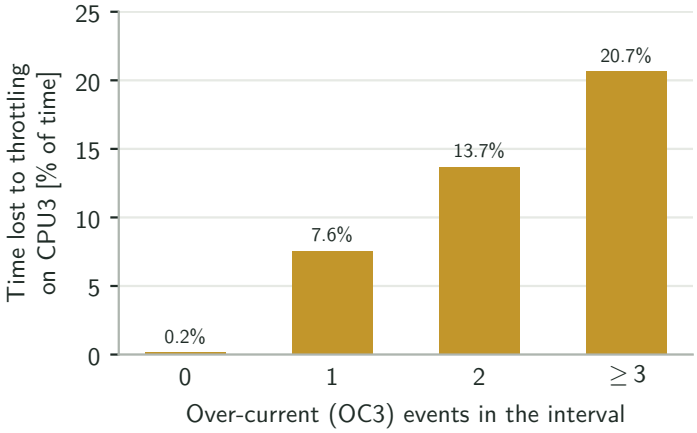


Figure 6.15: Time lost to throttling on CPU3, as a share of the time CPU3 is awake, in 10 ms intervals grouped by the number of over-current events, with tkHPick.

Figure 6.15 relates the two: in 10 ms intervals, the time lost to throttling on CPU3 grows by about 7% of the time CPU3 is awake for each OC3 event in the interval, and is practically zero in the intervals without events. This accounts for about 8 of the 14 points of the increase with tkHPick (Figure 6.13). Unlike memory contention, this slowdown does not depend on what the other cores do with memory: when the module draws too much power, the firmware slows down all the CPUs, and isolating CPU3 with a cpuset partition does not protect it.

6.7 A minimal task barely feels memory contention

Once the scheduler coupling is removed and CPU3 is isolated, the interferents still lengthen the wake-up of the minimal task, but only by 0.4 to 0.6 μs. Waiting for data and instructions from memory accounts for most of this increase with MEMINTERF and for less than half of it with tkHPick, and in both cases it amounts to a fraction of a microsecond, because more than 99.4% of the accesses of each iteration are served by the private caches of CPU3. With tkHPick, most of what remains is throttling, which no partition of the CPUs can remove. A real-time task that reads megabytes of data at every activation does not stay inside the private caches. Chapter 7 repeats the same analysis with such a task.

7. Results: a task with the memory footprint

Chapter 6 ended with a task that keeps almost all of its accesses inside the private caches of CPU3, so memory contention can lengthen its wake-up only a little. This chapter returns to the H^3 Vision victim (Section 4.2.2), which after every wake-up reads about 1.3 MB and writes 2 kB of data that are not in the cache, as much as a high-speed vision system for robots handles at every frame. The victim runs on CPU3, isolated with the cpuset partition, because Chapter 6 showed that there neither the scheduler nor the peripheral interrupts couple it to the interferents (Sections 6.4 and 6.5). The latency is measured exactly as before: the work of the H^3 Vision victim runs after the clock is read, so it can affect the latency only through the state it leaves in the caches for the next wake-up. Each scenario is compared with the baseline of the same task, and the minimal task is measured again in the same campaign, so that the two tasks can be compared directly.

7.1 On the isolated CPU3, the H^3 Vision footprint amplifies the interference

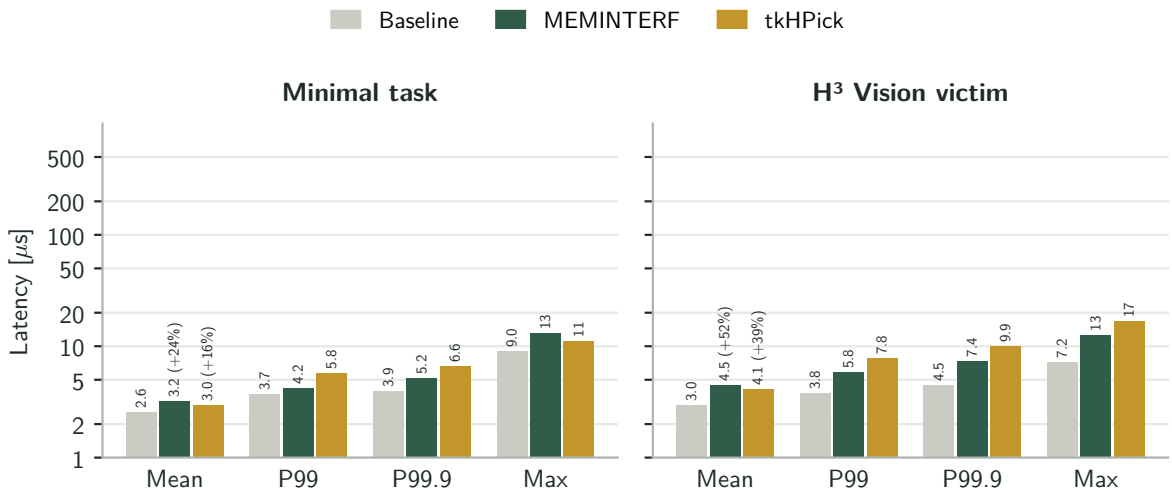


Figure 7.1: Wake-up latency on CPU3 with the cpuset partition, for the minimal task (left) and the H^3 Vision victim (right). The percentages compare the mean with the baseline of the same task. Each bar is the mean of three 300 s repetitions; for the maximum, the bar is the largest value observed. Same logarithmic scale as the latency figures of Chapter 6.

Figure 7.1 compares the two tasks. For the minimal task the interferents raise the mean latency by 24% with MEMINTERF and by 16% with tkHPick, slightly more than the 22% and 14% of Section 6.6, which come from a different campaign with a

baseline of 2.65 instead of 2.57 μs. For the H^3 Vision victim the increase is about twice as large: 52% with MEMINTERF and 39% with tkHPick, that is, 1.54 and 1.14 μs against 0.63 and 0.40 μs. The tail grows with it: the 99.9th percentile goes from 4.5 μs in the baseline to 7.4 μs with MEMINTERF and 9.9 μs with tkHPick.

With CPU affinity on CPU0 the relative increase of the H^3 Vision victim is not larger than that of the minimal task (Figure 6.1 and Section 6.1): there, the scheduler coupling and the peripheral interrupts probably dominate, and the effect of the footprint becomes visible once they are removed.

Even without interferents, the H^3 Vision victim wakes up about 0.4 μs later than the minimal task, 2.95 against 2.57 μs. Its work does not run inside the latency window, so what slows the next wake-up is what the work leaves behind: about 1.3 MB is five times the private L2 cache of the core and more than half of the L3 cache of the cluster (Table 4.1), so after every activation the code and data that the kernel needs for the next wake-up are no longer in the private caches. Section 7.3 measures this.

7.2 The kernel does the same work but waits longer

The wake-up of the H^3 Vision victim is broken down with the same per-phase probes as in Section 6.6 (Section 5.6). Inside the latency window the kernel executes between 15,500 and 15,900 instructions per wake-up in every case, for both tasks, with and without the interferents: the footprint of the H^3 Vision does not change the work of the wake-up, only how long it takes.

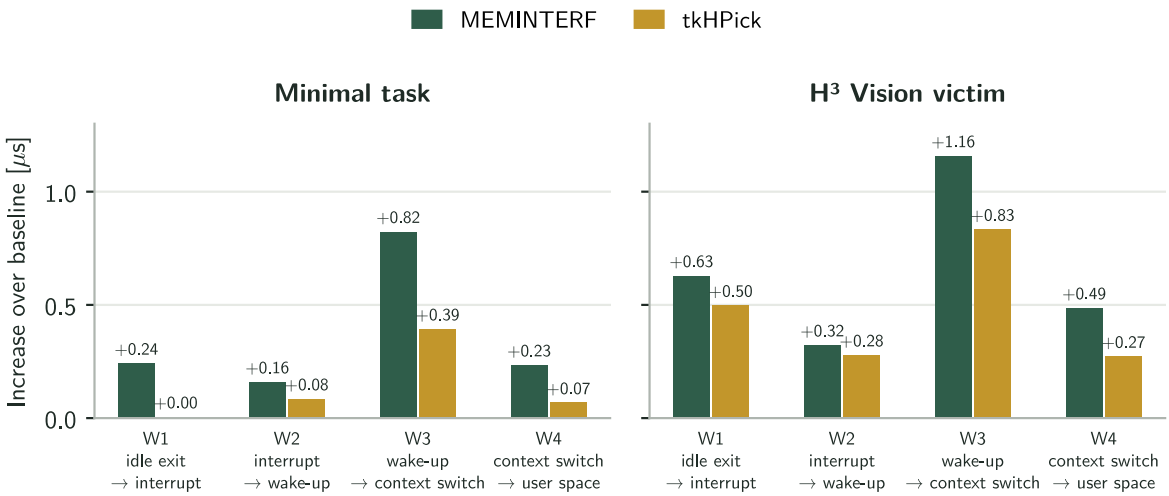


Figure 7.2: Increase of the duration of each phase of the latency window on CPU3 with the cpuset partition, with respect to the baseline of the same task, measured with all the probes.

For the minimal task most of the increase falls in W3, between the wake-up and the context switch (Figure 7.2). For the H^3 Vision victim every phase grows. W3 still grows the most, by 1.16 μs with MEMINTERF and 0.83 μs with tkHPick, but W1, from the exit from idle to the timer interrupt, now grows by 0.63 and 0.50 μs, against

0.24 and practically zero for the minimal task. W1 grows for the same reason as W3: it executes about 3,000 instructions in every case, and with MEMINTERF 0.45 of its 0.63 μs are spent waiting for data and instructions from memory. W1 runs the code that leaves idle and enters the timer interrupt, which was last used one period earlier, before the work of the H^3 Vision victim evicted it from the private caches: even without interferents it takes 0.2 μs longer than for the minimal task.

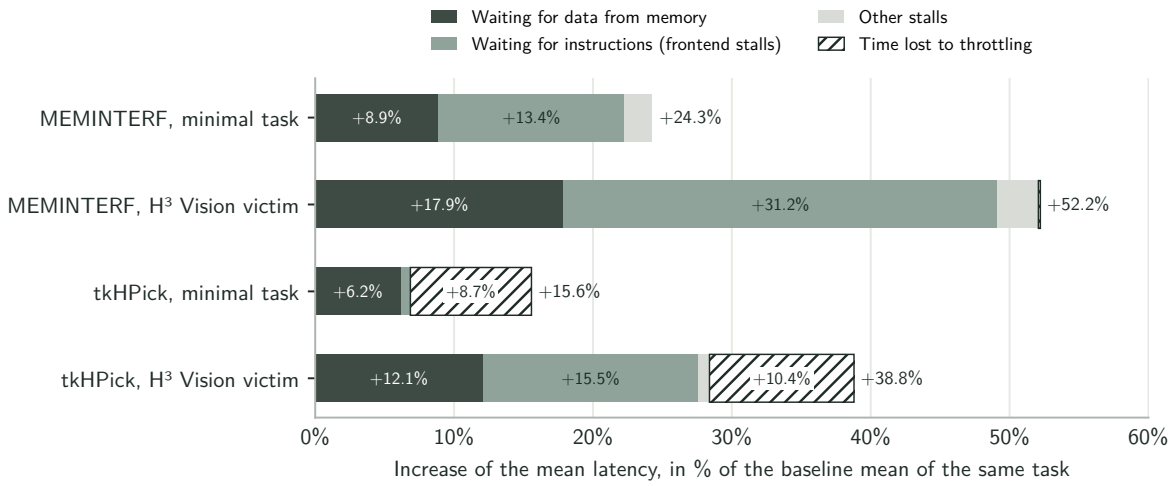


Figure 7.3: Increase of the mean latency on CPU3 with the cpuset partition, split into its components and expressed as a share of the mean latency of the baseline of the same task. Other stalls also include the cycles spent executing instructions.

Figure 7.3 splits the increase of the mean latency as in Section 6.6. With MEMINTERF the time lost to throttling is zero, and both memory components about double with the H^3 Vision footprint: waiting for data goes from 9 to 18 points, waiting for instructions from 13 to 31. With tkHPick the minimal task waits mostly for data and for the throttling, and almost never for instructions. With the H^3 Vision footprint, waiting for instructions appears, with 16 points, and becomes the largest component, while the time lost to throttling stays about the same, 10 points against 9. Together, the two memory components grow from 22 to 49 points with MEMINTERF and from 7 to 28 points with tkHPick.

7.3 The H³ Vision work evicts the kernel from the private caches

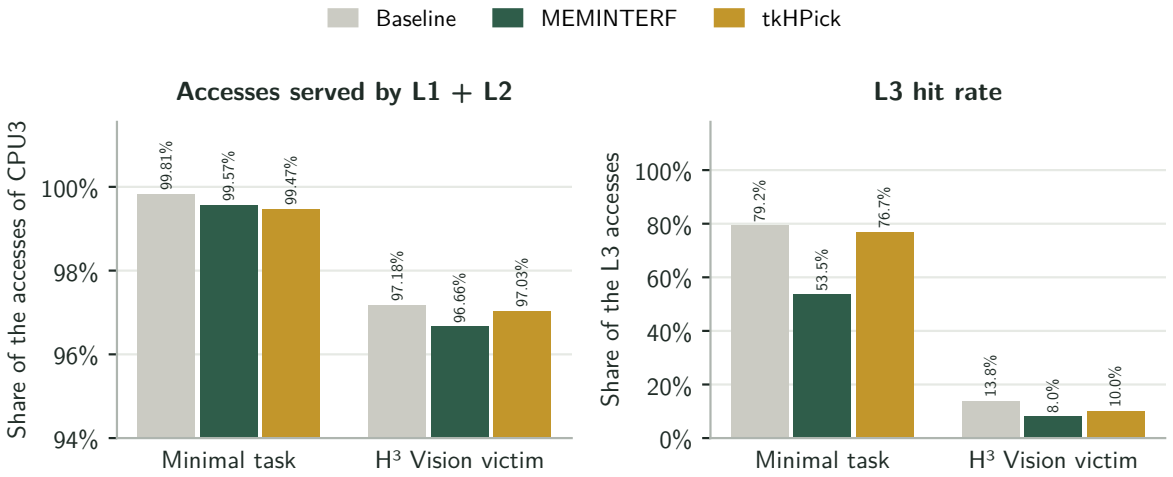


Figure 7.4: Memory accesses of CPU3 with the cpuset partition, counted for the whole CPU. Left: share of the accesses of the core, for data and instructions, served by the private L1 and L2 caches; lines brought into the L2 by the hardware prefetcher are not counted as L2 misses. Right: share of the accesses to the L3 cache of the cluster that hit in it.

For the minimal task, more than 99.4% of the accesses of CPU3 are served by its private caches in every scenario, and most of the few accesses that reach the L3 hit in it (Figure 7.4). With the H³ Vision footprint, about 3% of the accesses miss in the private caches on demand, but this share does not include the lines that the hardware prefetcher brings into the L2, which also come from beyond it. Almost all the accesses that reach the L3 miss in it, and its hit rate falls from about 79% to between 8 and 14%. This is what the work is built to do: every activation reads about 20,000 lines of 64 bytes that no task has touched for a long time, and the L3 refills, about 20,000 per wake-up, match them one by one. The L2 refill event does not count the refills caused by prefetches into the L2, while the L3 refill event counts every read that returns data from outside the cluster and excludes only the prefetches into the L3 [1, Section C2.3], so the L3 refills exceed the L2 ones.

The work runs outside the latency window, but its data replace the content of the private caches. At the next wake-up the kernel finds its code and data evicted: the L1 instruction cache misses a similar number of times, about 240–250 times per wake-up for the minimal task and 270–340 for the H³ Vision victim, but since the work has replaced the content of the L2, these lines are most likely served from beyond it, and often beyond the L3, where they compete with the traffic of the interferents. The counters cover the whole CPU and cannot show where each line is served, but this is the most likely origin of the extra waiting for instructions of Figure 7.3, and it explains why the number of instructions does not change. At this level the three scenarios look alike: the interferents change little how many accesses leave the private caches, but they change how long each of them takes.

7.4 tkHPick rivals MEMINTERF

With the minimal task, measured alongside the H^3 Vision victim, the memory contention caused by tkHPick is hard to see: waiting for data and instructions adds 7 points to the wake-up, against 22 with MEMINTERF, and most of the increase with tkHPick is throttling (Figure 7.3). The H^3 Vision victim, which uses the memory hierarchy as a real vision task does, changes this picture. The memory part of the increase grows four times with tkHPick, to 28 points, and more than doubles with MEMINTERF, to 49 points (Figure 7.3). Together with the time lost to throttling, the whole increase of the mean latency, 39% with tkHPick and 52% with MEMINTERF, is of the same order for the two interferents, and with tkHPick the tail grows even more (Figure 7.1). MEMINTERF is built to load the memory as much as possible, with a copy loop on each of the three cores, while tkHPick runs its neural-network inference on the GPU, and its four processes share the other three cores (Section 4.2.4). An industrial computer-vision application therefore lengthens the wake-up of a task with a realistic memory footprint by an amount comparable to a synthetic memory stressor, partly through memory contention and partly through throttling.

Three choices of the measurement bound how far this result can be carried. The split of Figure 7.3 applies the shares measured with the probes to the increase measured without them, so it assumes that the probes slow down every component in the same proportion. The work of the H^3 Vision victim reproduces only the amount of data that the vision system handles at every frame, not its image processing. Finally, only the wake-up latency is measured: how much the interferents lengthen the work of the H^3 Vision victim itself, which runs after the latency is recorded, is outside the scope of this thesis.

8. Conclusions and future work

This thesis set out to measure how much an industrial computer-vision application, running next to a periodic real-time task on the same embedded system-on-chip, lengthens the mean wake-up latency of that task, and through which channels this interference travels. Perception and control increasingly share one platform, while the interference studied so far on NVIDIA Tegra came mostly from synthetic workloads that load the memory as much as possible and do not say whether an industrial application produces effects of the same size. On a Jetson Orin Nano, we therefore compared an industrial perception application, tkHPick, with the synthetic interferent MEMINTERF, and measured their effect on `cyclictest` in two forms: a minimal task, whose own work is negligible, and the H^3 Vision victim, which after every wake-up handles as much data as a high-speed vision system does at every frame.

With the minimal task, the interference came through the channels around the task. With CPU affinity only, both interferents raised its mean latency to about 1.5–1.6 times that of the baseline. With tkHPick the victim also executed more kernel instructions: when it went to sleep, the scheduler searched the cores of tkHPick for a task to pull, because the victim CPU shared its root domain with them. A cpuset partition removed this coupling and lowered the latency in every scenario, but it did not change where the peripheral interrupts were handled, and with tkHPick CPU0 kept receiving many more of them, mostly from the GPU. On the isolated CPU3 only an increase of 0.4 to 0.6 μs remained. Memory contention added only a fraction of a microsecond: the task is so small that more than 99.4% of its accesses were served by the private caches of its core. With tkHPick most of the remainder was over-current throttling, which slows down all the CPUs when the module draws too much power and which no partition of the CPUs can prevent.

With the H^3 Vision victim, memory became a main channel. On CPU0 with CPU affinity only, the scheduler and the interrupts probably hid the effect of its footprint, but on the isolated CPU3 the increase of its mean latency was about twice that of the minimal task: 52% with MEMINTERF and 39% with tkHPick. The kernel executed the same instructions as before, but the work of the previous activation had pushed them, together with the data they use, out of the private caches, so the kernel waited longer for memory, where it competed with the traffic of the interferents. Once a real-time task accesses memory beyond its private caches, an industrial computer-vision application thus lengthens its wake-up by an amount comparable to that of a synthetic interferent built to disturb as much as possible, partly through memory contention and partly through throttling.

Besides these results, the thesis contributes a way of carrying out the analysis. Keeping the victim fixed and changing only what runs on the other cores allows an industrial application to be compared directly with a synthetic reference. Using the same task in two forms separates the channels that act around a task, namely the scheduler, the interrupts, and the power supply, from the one that depends on the memory the task itself uses. Splitting each wake-up into phases with eBPF probes and hardware counters then attributes the remaining increase to waiting for data, waiting for instructions, and time lost to throttling.

These conclusions hold within the limits of the study. Only the mean latency was analysed: the high percentiles and the maxima were measured and reported, but not explained. Only the wake-up was measured, not the time the H^3 Vision victim needs to complete its own work. The experiments involve one board and one application, and the H^3 Vision victim reproduces the amount of data handled by the vision system, not its image processing. Finally, the split of the latency increase into its components applies the shares measured with the probes to the increase measured without them, so it assumes that the probes slow down every component in the same proportion.

Future work follows from these limits, starting with the tail. With CPU affinity only, the maximum latency reached about half of the 1 ms period, and although such values were rare, a single one can make a task miss its deadline. Understanding what happens in these outliers requires measurements that follow individual activations, since the counters used here accumulate over a whole run and cannot link a single outlier to a specific event. A second direction is the scheduler under CPU affinity. Newly-idle balancing on the victim CPU searched the cores of tkHPick for a task to pull, although the affinity of almost every candidate forbade the move, and the kernel detected this only at the end of the search, when it examined each candidate. Whether this search can be avoided, or cut short when no candidate may move, deserves a closer study of the load-balancing code.

The measurements themselves can be extended. The work of the H^3 Vision victim runs after the latency is recorded, and how much the interferents lengthen it, and with it the response time of the task, was left outside this thesis. The size of the footprint can be varied between that of the minimal task and the 1.3 MB used here, below and above the 256 KiB private L2 cache and the 2 MiB L3 cache of the cluster, to find from which size memory contention starts to weigh on the wake-up. Repeating the analysis on other boards and with other applications, heavier or more complete than tkHPick, would show whether industrial and synthetic interference remain comparable beyond this case.

The configurations studied in this thesis can also be combined with mitigations. On the memory side, cache coloring and the other partitioning and regulation techniques of Section 3.3 could protect a task like the H^3 Vision victim, whose wake-up depends on the memory hierarchy it shares with the interferents. On the kernel side, the `nohz_full` boot parameter could further reduce the interrupts that still reach the isolated CPU, such as the scheduler tick. Over-current throttling needs a different approach, because it comes from the power budget of the module and not from what the other cores do with memory: its effect could be studied under other power modes, or with a lower GPU frequency that keeps the module below the over-current limit.

For one platform and one application, this thesis has shown that an industrial perception application disturbs a real-time task through channels that CPU placement alone does not close, and that, once the task uses memory as a real task does, the disturbance becomes comparable to that of a synthetic workload built for the purpose. The next steps aim to extend this answer from the mean latency to the whole behaviour of the task.

9. References

[1] Arm Limited. Arm Cortex-A78AE Core Technical Reference Manual. https://documentation-service.arm.com/static/5fb7c9cdca04df4095c1d5e4, November 2020. Revision r0p1, document 101779_0001_06_en. Sections C2.1, C2.3, and C3.4. Accessed October 4, 2026.

[2] Michael Bechtel and Heechul Yun. Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study. *IEEE Transactions on Computers*, 73(7):1753–1766, 2024.

[3] Alessio Igor Bogani and John Kacur. cyclictest(8): High Resolution Test Program. https://manpages.debian.org/unstable/rt-tests/cyclictest.8.en.html. Manual served from Debian rt-tests 2.10-1, Options. Accessed October 4, 2026.

[4] Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. Demystifying the real-time Linux scheduling latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[5] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, 3 edition, 2011.

[6] Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko Bertogna. Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms. In *2020 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2020.

[7] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *2017 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10. IEEE, 2017.

[8] Eclipse iceoryx contributors. iceoryx: Overview. https://iceoryx.io/classic/v2.0.5/getting-started/overview/. Documentation version 2.0.5, RouDi. Accessed October 4, 2026.

[9] Brendan Gregg. CPU Flame Graphs. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html. Author's documentation, Interpretation and Instructions. Accessed October 4, 2026.

[10] Brendan Gregg. FlameGraph: Stack Trace Visualizer. https://github.com/brendangregg/FlameGraph. Project source and README, stackcollapse-perf.pl and flamegraph.pl. Accessed October 4, 2026.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6 edition, 2017.

[12] Tejun Heo. Control Group v2. `https://www.kernel.org/doc/html/v5.15/admin-guide/cgroup-v2.html`. Linux 5.15 documentation, Cpuset. Accessed October 4, 2026.

[13] HeSoC-mark: meminterf Source Code. `https://git.hipert.unimore.it/mem-prof/hesoc-mark/-/blob/2432bf23f22cc4a2c06666c255bcfe7c41342146/membench/meminterf.cpp`. mem-prof project repository, revision 2432bf23f22cc4a2c06666c255bcfe7c41342146, membench/meminterf.cpp. Accessed October 4, 2026.

[14] Masami Hiramatsu. Kprobe-based Event Tracing. `https://www.kernel.org/doc/html/v5.15/trace/kprobetrace.html`. Linux 5.15 documentation. Accessed October 4, 2026.

[15] Idaku Ishii, Taku Taniguchi, Ryo Sukenobe, and Kenkichi Yamamoto. Development of high-speed and real-time vision platform, H^3 vision. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3671–3678, 2009.

[16] Linux Foundation Real-Time Linux project. Cyclictest. `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`. Real-Time Linux documentation, Introduction. Indexed excerpt consulted October 4, 2026.

[17] Linux kernel contributors. drivers/irqchip/irq-gic-v3.c. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/irqchip/irq-gic-v3.c?h=v5.15.148`. Linux 5.15.148 source code, function `gic_set_affinity`. Accessed October 14, 2026.

[18] Linux kernel contributors. Event Tracing. `https://www.kernel.org/doc/html/v5.15/trace/events.html`. Linux 5.15 documentation, Event Filtering. Accessed October 4, 2026.

[19] Linux kernel contributors. hrtimers: Subsystem for High-Resolution Kernel Timers. `https://www.kernel.org/doc/html/v5.15/timers/hrtimers.html`. Linux 5.15 documentation. Accessed October 13, 2026.

[20] Linux kernel contributors. kernel/time/hrtimer.c. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/time/hrtimer.c?h=v5.15.148`. Linux 5.15.148 source code, function `__hrtimer_init_sleeper`. Accessed October 14, 2026.

[21] Linux kernel contributors. Linux eBPF Helper Interface. `https://raw.githubusercontent.com/torvalds/linux/v5.15/include/uapi/linux/bpf.h`. Linux v5.15 source, include/uapi/linux/bpf.h, bpf_perf_event_read_value. Accessed October 4, 2026.

[22] Linux kernel contributors. Scheduler Domains. https://www.kernel.org/doc/html/v5.15/scheduler/sched-domains.html. Linux 5.15 documentation. Accessed October 4, 2026.

[23] Linux kernel contributors. The Kernel's Command-Line Parameters. https://www.kernel.org/doc/html/v5.15/admin-guide/kernel-parameters.html. Linux 5.15 documentation. Accessed October 13, 2026.

[24] Linux kernel contributors. The /proc Filesystem. https://www.kernel.org/doc/html/v5.15/filesystems/proc.html. Linux 5.15 documentation, Section 1.3. Accessed October 4, 2026.

[25] Linux man-pages project. mlock(2): Lock and Unlock Memory. https://man7.org/linux/man-pages/man2/mlock.2.html. Description. Accessed October 4, 2026.

[26] Linux man-pages project. perf_event_open(2): Set Up Performance Monitoring. https://man7.org/linux/man-pages/man2/perf_event_open.2.html. Linux manual page, Description and perf_event_attr. Accessed October 4, 2026.

[27] Linux man-pages project. sched(7): Overview of CPU Scheduling. https://man7.org/linux/man-pages/man7/sched.7.html. Accessed October 13, 2026.

[28] Linux man-pages project. sched_setaffinity(2): Set and Get a Thread's CPU Affinity Mask. https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html. Description. Accessed October 4, 2026.

[29] Linux perf contributors. perf-list: List All Symbolic Event Types. https://raw.githubusercontent.com/torvalds/linux/v5.15/tools/perf/Documentation/perf-list.txt. Linux v5.15 source documentation, Event Modifiers. Accessed October 4, 2026.

[30] Linux perf contributors. perf-record: Run a Command and Record Its Profile into perf.data. https://raw.githubusercontent.com/torvalds/linux/v5.15/tools/perf/Documentation/perf-record.txt. Linux v5.15 source documentation, Options. Accessed October 4, 2026.

[31] Linux perf contributors. perf-report: Read perf.data and Display the Profile. https://raw.githubusercontent.com/torvalds/linux/v5.15/tools/perf/Documentation/perf-report.txt. Linux v5.15 source documentation, Children Overhead. Accessed October 4, 2026.

[32] Linux perf contributors. perf-stat: Run a Command and Gather Performance Counter Statistics. https://raw.githubusercontent.com/torvalds/linux/v5.15/tools/perf/Documentation/perf-stat.txt. Linux v5.15 source documentation, Options and CSV Format. Accessed October 4, 2026.

[33] Alessio Masola, Nicola Capodieci, Benjamin Rouxel, Giorgia Franchini, and Roberto Cavicchioli. Machine learning techniques for understanding and predicting memory interference in CPU-GPU embedded systems. In *2023 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 147–156. IEEE, 2023.

[34] MCAP contributors. MCAP Format Specification. `https://mcap.dev/spec`. Channel, Message, and Statistics records; online specification consulted October 4, 2026.

[35] NVIDIA. Jetson Linux Developer Guide: Kernel Customization. `https://docs.nvidia.com/jetson/archives/r36.4.3/DeveloperGuide/SD/Kernel/KernelCustomization.html`. Release R36.4.3, Building the Real-Time Kernel. Accessed October 4, 2026.

[36] NVIDIA. Jetson Linux Developer Guide: Platform Power and Performance—Jetson Orin Nano, Orin NX, and AGX Orin Series. `https://docs.nvidia.com/jetson/archives/r36.4.3/DeveloperGuide/SD/PlatformPowerAndPerformance/JetsonOrinNanoSeriesJetsonOrinNxSeriesAndJetsonAgxOrinSeries.html`. Release R36.4.3, Power Management and Maximizing Jetson Orin Performance. Accessed October 4, 2026.

[37] NVIDIA. Jetson Linux Developer Guide: Quick Start. `https://docs.nvidia.com/jetson/archives/r36.4.3/DeveloperGuide/IN/QuickStart.html`. Release R36.4.3, Jetson Modules and Configurations. Accessed October 4, 2026.

[38] NVIDIA. NVIDIA Jetson Orin Nano Series Data Sheet. `https://cdck-file-uploads-global.s3.dualstack.us-west-2.amazonaws.com/nvidia/original/4X/9/a/f/9afa72566a229ed11beabeb0750b52c07afb1569.pdf`, February 2023. Document DS-11105-001, version 1.0, February 22, 2023. Pages 1 and 6–8. NVIDIA document hosted on the NVIDIA Developer Forums. Accessed October 4, 2026.

[39] NVIDIA. NVIDIA Jetson Orin Nano Super Developer Kit. `https://dam-cdn.nvd.orangelogic.com/AssetLink/2ug686w80406gxf6q8uv7r3l7265m2n8.pdf`, December 2024. Developer kit datasheet, publication 3575392, December 2024. Pages 1–2. Accessed October 4, 2026.

[40] NVIDIA. Jetson Linux Release Notes. `https://docs.nvidia.com/jetson/archives/r36.4.3/ReleaseNotes/Jetson_Linux_Release_Notes_r36.4.3.pdf`, January 2025. Release R36.4.3, document RN_10698-r36.4.3. Section 1.1. Accessed October 4, 2026.

[41] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364. IEEE, 2017.

[42] Ashutosh Pradhan, Daniele Ottaviano, Yi Jiang, Haozheng Huang, Alexander Zuepke, Andrea Bastoni, and Marco Caccamo. Arm DynamIQ Shared Unit and real-time: An empirical evaluation. In *2025 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 241–254. IEEE, 2025.

[43] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Computing Surveys*, 52(1):18:1–18:36, 2019.

[44] Steven Rostedt. ftrace: Function Tracer. `https://www.kernel.org/doc/html/v5.15/trace/ftrace.html`. Linux 5.15 documentation, The Files and The Tracers. Accessed October 4, 2026.

[45] Seeed Studio. Flash JetPack OS to A603 Carrier Board. `https://wiki.seeedstudio.com/reComputer_A603_Flash_System/`. A603 Carrier Board documentation, introduction and Supported Module. Accessed October 4, 2026.

[46] Michael Williams. Statistical Profiling Extension for ARMv8-A. `https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/statistical-profiling-extension-for-arm` 2017. Arm Community, discussion of interrupt-based event sampling. Accessed October 4, 2026.

[47] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.

[48] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.