How to Affect Synchronization of Multi-Threaded Applications by Altering Processors' States

Graduating
Luca Sannino
1542194

Advisors
Romolo Marotta
Christian Napoli

Faculty of Ingegneria dell'Informazione, Informatica e Statistica
Department of Ingegneria Informatica, Automatica e Gestionale
Master Degree in Engineering in Computer Science

SAPIENZA
UNIVERSITÀ DI ROMA

A/A 2020/2021

Index

1. Introduction

One of the key issues that the IT infrastructure is facing right now is how ICT is negatively impacting our environment. ICT stands for Information and Communication Technology: it is a term colloquially used to define all the technology that deals with sending, storing and retrieving digital information, which includes data centers and Cloud computing. The worldwide growth of internet users, helped by the diffusion of mobile phones and which is estimated to reach nearly two-thirds of the global population by 2023 [1], has brought an increasing interest in the use of digital services, leading as well to the rise of the ICT industry. Consequently, the number of data centers being built is continuously increasing to accommodate such high traffic [2].

These centers, often used by corporations and governments in form of expansive server farms like the one in Figure 1.1, need to store huge amounts of data, to work 24/7 and to resist to damage by supporting redundancy: the result is that they have to consume high amounts of energy in order to power and cool themselves. The consequences of these high levels of power consumption are inevitably felt on the environment: in fact, as of 2017, it is estimated that ICT accounts for 2% of worldwide CO^2 emissions, with data centers in particular estimated to have the fastest growing carbon footprint of the entire sector [3][4].

Fig 1.1 – Data centers used by Amazon like the one pictured above are so widespread that websites hosted on them are visited daily by one third of worldwide internet users. Image and stat taken from [5].

These centers generally contain multi-core architectures that support parallel computing, enabling them to run multi-threaded applications that rely on synchronization algorithms to commit to a sequence of operations that ensures the consistency of the data [6]. These algorithms can be quite expensive, namely in those phases of the execution where processors often waste their clocks at full power by spinning until the running threads are able to access the critical section.

Since servers' computation accounts for about 40% of the total energy consumption (as shown in Figure 1.2), it is critical for High Performance Computing, a branch of computer science that focalizes on supercomputer architectures and parallel algorithms, to find new ways to optimize the computation and reduce energy consumption [6]. With this context in mind, improving the synchronization process is certainly one area of interest, and because multi-core architectures can now be found in our daily life, from our low-end laptops to our mobile phones, this objective has never been this relevant.

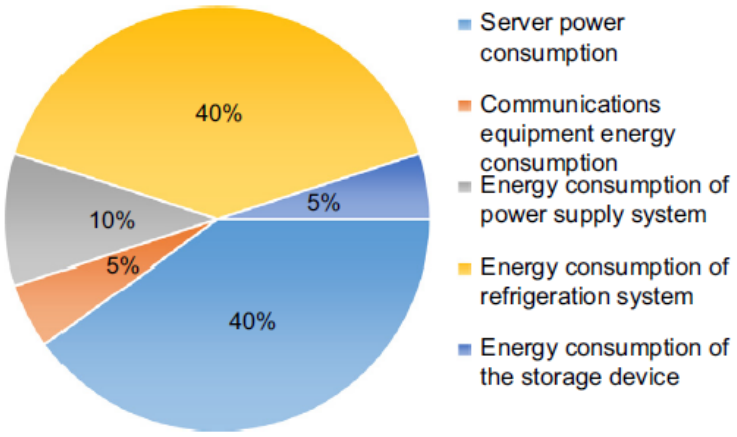

Fig 1.2 – A pie chart that shows how the various parts of a data center consume energy, and which underlines how a further 40% of energy consumption is caused by cooling systems, which has pushed engineers to also find new hardware-side optimazions. Pie chart taken from [6].

Since processors can target a variety of different hardware states that can drastically alter their usual behavior, it is in my interest to study what they bring to the tradeoff between energy consumption and performance in a running application: more precisely, if it is possible to save up on energy consumption by targeting specific hardware states during synchronization without downgrading the performance to a degree that the end user would consider unacceptable.

This research, which initially began by studying the related literature so I could learn about all the various types of states and how to trigger them, culminated in my contributions to Powerctl: a Linux kernel module that introduces a suite of system calls that allow user-space software to change hardware states by either writing on the model-specific registers (MSRs) that are responsible of such states' transitions or by using special privileged operations. Writing directly on MSRs allows to keep the latency as low as possible, making possible for user-space software to quickly request new states during synchronization. With these states at their disposal, threads have the ability to directly affect the processor's clock modulation (T-States), clock frequency (P-States) and its ability to enter a sleep state (C-States).

By using Powerctl to alter the behavior of processors while threads are waiting to acquire their lock for synchronization, we have the possibility to analyze which states

would help to jointly reduce contention and energy consumption the best. In particular, we focused on a specific aspect of lock-based synchronization, namely the implementation of back-off phases. In this context, we explored the opportunity given by such a hardware capability, evaluating which states would bring down the performance to an unreasonable level, how the size of the critical section and the number of active concurrent threads would affect the whole picture and, finally, which waiting policies would be the best suited to such changes.

Another feature of Powerctl is making sure that threads that didn't ask for a specific state are not subjected to performance profiles that were requested by previous threads: thanks to a probing system and to a thread-safe internal data structure that keeps track of all the various requests, Powerctl can always enforce the correct performance profile for each thread after a context switch while keeping latency at minimum.

Powerctl was ultimately tested as baseline software for benchmarking lock primitives implementations using a pair of tools, Litl and Lockbench, using various combinations of states and locking algorithms so to provide an expensive view on its capabilities. The objective of this thesis is to ultimately discuss the results of these benchmark tests and see if Powerctl's approach brought notable benefits or not on the synchronization process.

Lastly it must be noted that since all this work required an intensive study of low-level off-the-shelf hardware management, the x86 ISA has been targeted with a focus on Intel processors. This thesis is then structured as follows:

- Chapter 2 explains preliminary background notions so that readers can fully understand how power and thermal control works in Intel processors with an x86 ISA, as well as giving an overview on the most known spinlock synchronization algorithms so to comprehend how the ones used to test Powerctl work and how they were build trying to correct the disadvantages of their predecessors.
- Chapter 3 is a thorough explanation of the technical details behind Powerctl, from how its core components consist in a Linux kernel module to how it probes the system when running so it can assign the specific profile performances to the right threads.
- Chapter 4 is about the experimental study performed on the capabilities of Powerctl and the discussion about the results, while also going in detail about Litl and Lockbench, a pair of benchmark tools that were vital during the testing phase.
- The thesis will then conclude with a brief recap of the work done and some final remarks.

2. Preliminary Concepts

This chapter introduces the necessary preliminary notions to understand the contribution of this thesis: the first part (sections 2.1, 2.2, 2.3) deals with the main concepts behind the power management in x86 Intel processors, and the characteristics of the various CPU hardware states and how to change them. The second part (section 2.4) introduces some of the most known algorithms for lock-based synchronization in multi-threaded applications, some of which have been used in conjunction with Powerctl, so it was important to understand how they work and what are their advantages and disadvantages.

2.1 The Advanced Configuration and Power Interface

Before introducing the various states that can alter processors' behavior, it is important to clarify that these states are part of a standard called Advanced Configuration and Power Interface (ACPI): this standard defines a common specification of states that is used by the whole industry, and acts as an abstract interface (Figure 2.1) that links together hardware and software so that systems can perform power management through a specification called Operating System-directed configuration and Power Management (OSPM) [7].

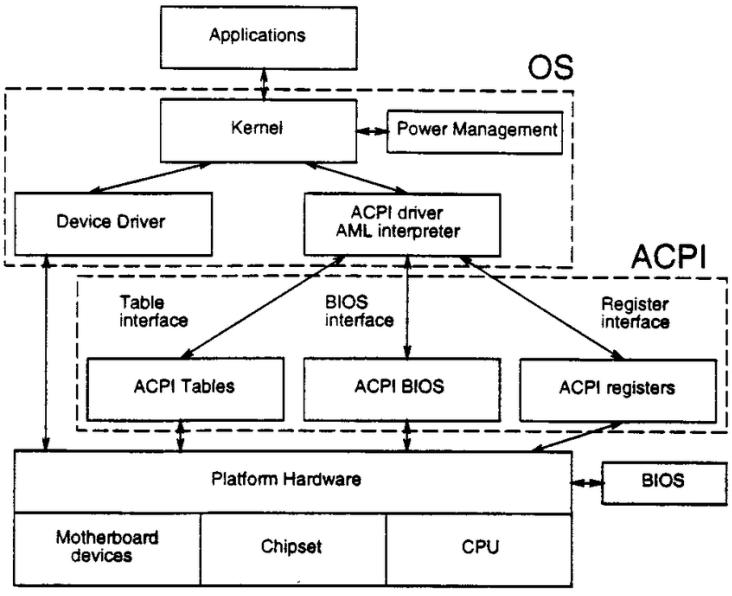

Fig 2.1 – This schema shows how ACPI acts as an interface between hardware and OS: its tables, loaded into memory by the ACPI BIOS, contain information about power management used by the OS to change processors' states, while its registers are used for hardware management. Schema taken from [8].

ACPI was conceived as a collaboration between Intel, Microsoft and Toshiba, and was first introduced in December 1996 [9]: it was initially only supported on Windows systems, but it also become available to Linux starting from version 2.4 [10].

Systems that are ACPI-compatible can recognize four Global States (G-States) and at max six Sleep States (S-States), which are mapped to G-States following this schema [7]:

- G0: Working.
  - S0. The system is fully running and computing.
- G1: Sleeping.
  - S1: Power on Suspend. The processor does not execute instructions anymore and its caches are flushed, but both still remain powered on.
  - S2: Power Off. Both CPU and its cache are turned off.
  - S3: Sleep. The system turns off its components after saving its state in the RAM, so that execution can be resumed at a later point.
  - S4: Hibernation. Similar to S3, but with the difference that the system saves its state to a non-volatile memory, like its hard drive, before turning off its components.
- G2: Soft Off.
  - S5: The system is turned off with the exception of those components that are able to turn it on, like the Power Button. Once turned on, the system resumes back from G0/S0, but all the context of the previous session is entirely lost.
- G3: Mechanical Off. The system is fully turned off.

ACPI also defines states whose only scope is to alter the power consumption of a processor while keeping it turned on and without modifying the rest of the system. These states, as shown in Figure 2.2, can be only enabled when the system finds itself in the G0/S0 state [7], and consist of:

- Throttling States (T-States), which directly affect processors' clock modulation.
- Performance States (P-States), which alter processors' voltage and frequency.
- CPU Power States (C-States), which are able to put to sleep various processors' components.

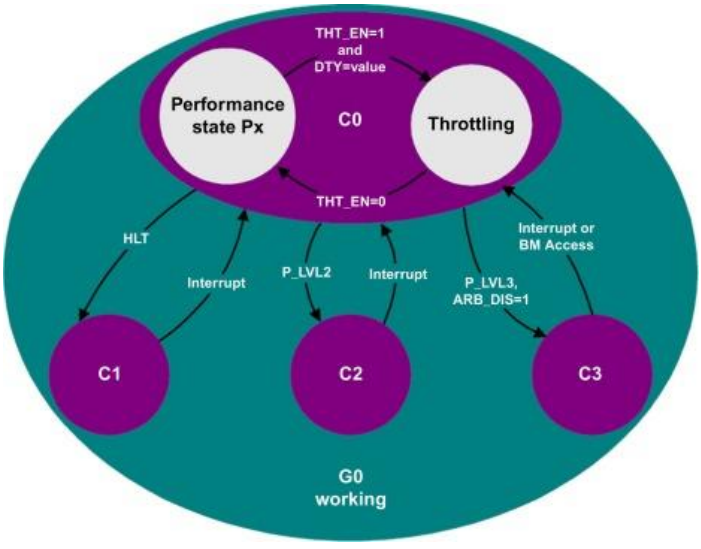

Fig 2.2 – A schema that shows how processor's states fit in regards to Global and Sleep States: CPU Power States (C-States) can only be enabled in G0/S0, while Power States (P-States) and Throttling States (T-States) can only be toggled in G0/S0/C0. Image taken from [11].

2.2 Power Consumption in a Processor

Since processors' energy consumption is a fundamental topic in this thesis, it's obligatory to establish prematurely how a CPU consumes power before looking at the various states, so to recognize those key areas that are the most responsible when it comes to power consumption and that are often the target of optimizations.

Even though each model of CPU has its own internal parts, there is a common schema that divides the total power consumption in two major components [12]:

$$p_{total} = p_{dynamic} + p_{static}$$

The dynamic component depends from the charging and discharging of the capacitors inside the switches of the logic gates during computation. These switches cause transistors to change their states: however, since these changes are not immediate, there is also a short-circuit power loss component that happens when multiple transistors are partially on, and that can be simply ignored or at worst approximated by adding 2-10% of the switching power to $p_{dynamic}$. This dynamic component can then be expressed as [12]:

$$p_{dynamic} = a \cdot CV^2 \cdot f + p_{short-circuit}$$

where V and f are respectively the voltage and frequency of the processor, C is the system capacitance and a is the activity factor that depends by often these switches occur.

The static component instead depends on the leakage power, which is a phenomenon that occurs when turned-off transistors still conduct small amounts of current that slowly discharge capacitors. This component can be expressed as [12]:

$$p_{static} = I_{leakage} \cdot V$$

where $I_{leakage}$ is the leakage current.

The ramifications of these formulas is that optimizations based on tuning down the frequency and the voltage of a processor lower both dynamic and static components, while others that are built on changing the applied clock signal only directly affect the dynamic one, since the only value that is altered is the activity factor a: the fewer are the applied clock signals, the lower is the number of switches inside logic gates [13].

2.3 Power and Thermal Control in Intel CPUs

As stated in the Introduction, Powerctl's primary goal is to offer an efficient user-friendly API that affect Intel x86 processors' behavior by changing their internal states with as low latency as possible.

We have already established that these states can be subdivided into different categories according to ACPI, but they all target the same goal: to reduce the power consumption of a processor. They can be changed either due to automated hardware mechanisms and control algorithms carried out by BIOS and OS, or as a consequence of user-made policies enforced by software, which is exactly the case of Powerctl.

Reasons for why a processor needs to run under a new state can vary, but generally include:

- The processor is reaching a critical temperature that can damage its internal components.
- The processor is inactive and it can afford to safely enter a sleep state so that it doesn't waste away its resources.
- The processor is consuming more energy than the user would like.

The rest of this section will then go in detail with some of these power and thermal control mechanisms.

2.3.1 On-Die Thermal Sensors

Intel processors include on-die sensors that allows thermal monitoring and protection. These sensors can't be manipulated by software so that trigger conditions can't be changed to unsafe ones [14], but they provide model-specific registers that software can read to know about their status, such as the one shown in Figure 2.3.

Such automated mechanisms include:

- A Catastrophic Shutdown Detector that halts execution of a processor until the next restart upon reaching a critical temperature that is factory calibrated. This sensor was introduced starting with the P6 micro-architecture, shipped in November 1995 [14] [15].
- A Thermal Monitor (TM1) that trips when the processor reaches a factory preset temperature which, while below the one set for the Catastrophic Shutdown Detector, is still enough to be considered dangerous. This monitor reacts by changing the duty cycle of the processor's clock until the temperature cools down enough to be considered safe once again [14] [16]. Altering the duty cycle of a processor does not mean slowing down the signal of the clock, but it is about enabling a modulation signal that, while active, inhibits the clock signal from being applied on the processor (Figure 2.4) [13]: to give one example, when a 50% duty cycle is enabled, the processor executes nearly half of the instructions in the same time-frame of when the duty cycle is at full capacity. This Thermal Monitor was introduced starting with Pentium 4 processors, which began shipping from November 2000 [16].
- A newer type of Thermal Monitor (referred as TM2) that trips with the same temperature threshold of TM1, with the difference that instead of altering the duty cycle of a processor, it reduces its operating frequency and voltage. This type was designed to offer a higher performance level than TM1, but not to replace it: following a power-up, the BIOS automatically enables one of them, while an application can choose TM2 as its preferred one by setting bit 16 of a model-specific

register called MSR_THERM2_CTL. It was introduced starting with Pentium M processors, which were launched in March 2003 [14] [17].

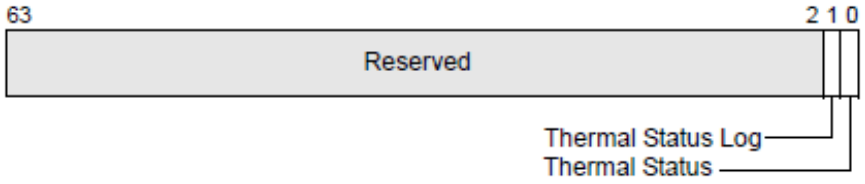

Fig 2.3 – The IA32_THERM_STATUS register. Even though thermal sensors can't be directly accessed by software, it's still possible to read this register to know if the sensors are being currently triggered ('Thermal Status' bit) or if they have been tripped since the last read ('Thermal Status' log bit) [14].

2.3.2 Software-Controlled Throttling States

Preceding the existence of both P-States and C-States, and nowadays seemingly overlooked by Intel itself [18], T-States (Throttling States) are essential in software-controlled power and thermal management since they allow the operating system to change a processor's clock modulation, a capacity that otherwise would be exclusive to the Thermal Monitor 1 [14].

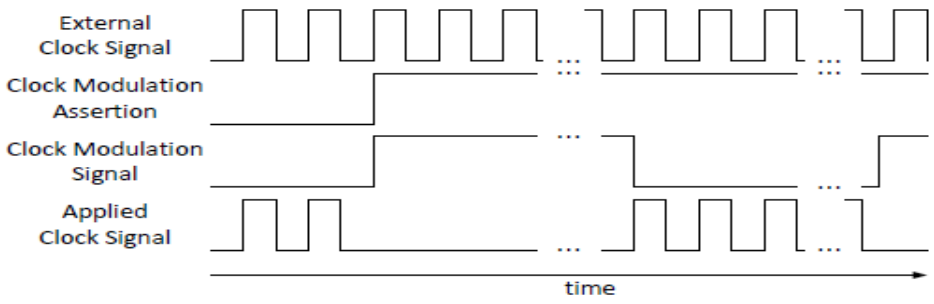

Fig 2.4 – How clock modulation works. Just like it happens with the Thermal Monitor, clock modulation doesn't affect the frequency of the signal, but only how much it is applied. Image taken from [13].

These states can be enabled by setting the 'On-Demand Clock Modulation Enable' bit on the IA32_CLOCK_MODULATION MSR (shown in Figure 2.5), while the preferred Duty Cycle setting can be set by writing its specific 3-bit control value in the 'On-Demand Clock Modulation Duty Cycle' field of the same register [14]: Powerctl follows this strategy to change T-States.

Each physical processor core has its own register, meaning that is possible to set different duty cycles for each of them, while the CPUs that support Hyper-Threading, a technology that allows two execution contexts for each physical core [19], should target the same duty cycle settings for all the logical cores mapped to the same physical one: otherwise all logical cores will either target the highest duty modulation setting between each other or the lowest one according to the processor's family [14].

Software can alter duty modulation by 12.5% intervals, but a more granular control that allows intervals of 6.25% is supported if bit 5 of EAX is enabled when calling CPUID with

EAX=6 [20]: CPUID is an instruction in the x86 architecture that allows software to check which capabilities are enabled on the processor, but to make this check easier on the user's behalf, Powerctl offers a function that directly returns the value of the relevant bit.

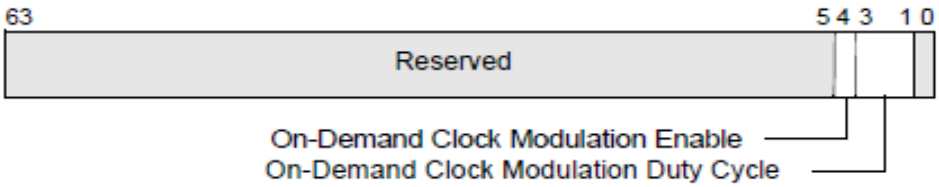

Fig 2.5 – The IA32_CLOCK_MODULATION register. Software uses 3-bit encodings on the 'On-Demand Clock Modulation Duty Cycle' field to change T-States. However if Extended Clock Modulation Duty is supported, bit 0 also become available, allowing 4-bit encodings and doubling the possible combinations [14].

The main advantages of T-States are that their control values are independent of the system used (Figure 2.6) and the fact they can be enabled with little to no latency: in some processors, the worst case scenario is a 43.5 μs delay that happens when a new duty modulation setting is requested right after the beginning of a new throttling event. This is the point when a processor checks if a new state was enabled, meaning that a new setting at worst must wait the period of this cycle before the request is acknowledged [13].

Conversely the main disadvantages are that, as stated in section 2.2, only the activity factor a in the dynamic part of the total power consumption's formula is affected by any optimization that is based on only altering T-States. Also, even though the various duty cycles settings promise to target precise percentages that indicate how much the clock should be modulated, the reality is that software should consider an additional 1.5-3% to the clock modulation settings they choose, since performance is downgraded slightly more than it should on paper [13].

They were introduced alongside the Thermal Monitor starting with Pentium 4 processors [16].

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

Fig 2.6 – Encodings used to set T-States. These are architecture-independent, making optimizations that rely on them more portable by default [14].

2.3.3 Software-Controlled Power-States

Mirroring the same concept of software-controlled T-States, which made possible to alter the system's behavior in the same fashion of a Thermal Monitor, software-controlled P-

States (Power States) were made available to manually alter the voltage and frequency of a processor just like Thermal Monitor 2 would automatically do under hot temperature situations.

Software can indicate a new P-State by writing its 16-bit encoding in the 'Transition Target' field of the IA32_PERF_CTL register (shown in Figure 2.7): transitions to a new target, however, are not instant. To figure out the current P-State that is actually targeted by the system until the new change is applied, software can read another register called IA32_PERF_STATUS, which is updated dynamically when said change actually takes place [14].

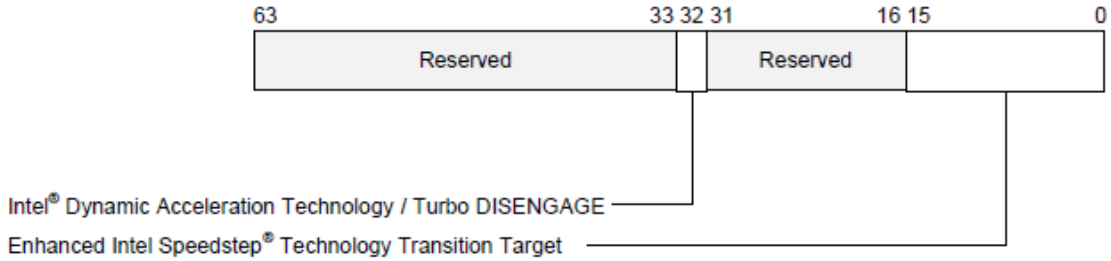

Fig 2.7 – The IA32_PERF_CTL register, which also contains a bit for enabling boost technologies: these will be explained in the next section [14].

In order to help software make more accurate guesses on the next P-State to target, two additional 64-bit counter registers are available to read: IA32_MPERF and IA32_APERF.

- IA32_MPERF increments proportionally to a fixed frequency, which is set when the processor is powered on.
- IA32_APERF increments in relation to the actual performance, meaning that it takes in consideration performance hits due to the activity of both Thermal Sensors and/or applied T-States.

Both registers start again from 0 when one of them overflows, and they offer no meaningful insight when taken singularly. However it is possible to calculate their ratio which, when multiplied to the percentage of time spent while the processor is not idling during a sample window, gives a percentage that expresses the true utilization of the processor in said window:

$$percentage_{used} = percentage_{not-idle} \cdot \left(\frac{APERF_{value}}{MPERF_{value}}\right)$$

Software that reads the result percentage can then target the next performance profile according to its established policies [14].

Powerctl offers system calls to change P-States using the approach described above and obtain the values of both APERF and MPERF.

P-States' main advantage is that by changing the voltage and frequency of a processor they can alter both dynamic and static components of the power consumption's formula, leading to more impactful optimizations than T-States [13]. On the contrary, there's a non-

negligible latency between a state transition completion and its request, which in more modern processors can reach up to 500 µs if the request falls right after the end of an update interval cycle [21] [22], as shown in Figure 2.8.

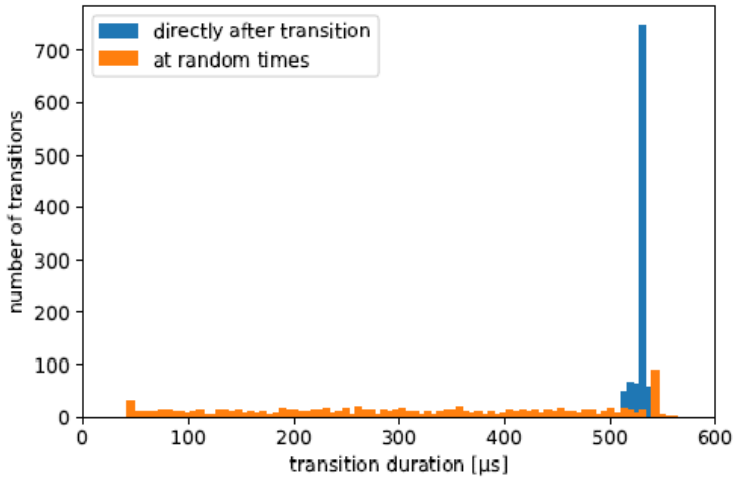

Fig 2.8 – A chart that shows the latencies when requesting a new P-State on more modern processors like Skylake-SP, which underlines how missing the update window can lead to considerable delays. Chart taken from [22].

Also, unlike what happens with T-States, values used to define the various performance configurations are different for each processor's architecture: software that wants to perform a transition to a designed P-State must then need a way to figure out its specific value inside the ACPI tables before writing it in the IA32_PERF_CTL register, meaning that without taking the right precautions strategies based on P-States are not as portable as those based on T-States. Regardless of the control values, it must be noted that P-States do follow a specific convention: starting from P0, which is the highest target for frequency and voltage, each subsequent state indicates a transition to a performance profile that is lower than the preceding one [23].

The last downside of P-States is that only recent generations of processors, such as the Intel Haswell, have introduced the capability to have a fine-grained control of the P-States, allowing to set them per CPU core independently from each other [21]. This is in contrast to T-States, whose control has been fine-grained from the beginning.

Software-controlled P-States are only available for processors that support Enhanced Intel Speed-Step Technology, which was introduced along Pentium M processors in 2003 [17]: support for this feature can be scanned by calling CPUID with EAX = 1 and checking bit 7 of ECX [20], but once again Powerctl has a function that directly returns this value.

2.3.4 Boost Technologies

Beyond Thermal Monitor 2 and manually writing on IA32_PERF_CTL, there are additional mechanisms that allow changes to P-States that user-space programs should be

aware of: these mechanisms, if not disabled, may take precedence in selecting a new P-State over what those programs choose.

Some Intel processors may contain boost features that allow them to set higher performance profiles when there's enough thermal headroom available, such as Intel Turbo Boost Technology or Intel Dynamic Acceleration Technology: the former works across both multi-threaded and single-threaded workloads, while the latter engages when only a single thread is active [14].

Both can disabled by clearing the 'Intel Dynamic Acceleration Technology/ Boost DISENGAGE' bit in the IA32_PERF_CTL register [20] or by calling a Powerctl's function that automatically does it for the user.

Processors can find out if they support this feature by calling CPUID with EAX = 6 and by checking bit 1 of the output EAX [20], or once again by letting Powerctl do the call. Intel Dynamic Acceleration Technology was introduced in Intel Core 2 Duo processors in 2007 [14], while Intel Turbo Boost Technology in Core i7 processors a year later. [24].

2.3.5 CPU Drivers and Governors

Another important factor to keep in mind in the field of software-controlled P-States is the influence of governors in Linux environments, which alter the performance of a processor according to their guidelines through policies enforced by OS and set by CPU drivers.

One of the most common Linux CPU drivers is 'acpi-cpufreq', which includes various OS governors such as [25]:

- 'performance', which always targets the highest performance in a given range defined by the user.
- 'powersave', which is the opposite of 'performance' by targeting the lowest performance of the range.
- 'ondemand', which dynamically targets a performance in the range accordingly to the activity of the processor.
- 'userspace', which directly gives to the user to ability to set P-States.

If user-space programs want to be the sole responsible of changing P-States then it is essential to set 'userspace' as the chosen CPU governor, which can be manually enabled by writing 'userspace' on /sys/devices/system/cpu/cpufreq/scaling_governor [26] or by calling a dedicated function in Powerctl.

However, there may be an additional step for some processors that deploy a different CPU driver, especially recent ones from Intel that enable by default a CPU driver called 'intel_pstate', which only support two governors, 'powersave' and 'performance', meaning that setting a custom P-State is not possible until it remains enabled [26]. To disable it, it is necessary to add an additional parameter to GRUB, the boot loader of Linux, by setting GRUB_CMDLINE_LINUX_DEFAULT="intel_pstate=disable" in

/etc/default/grub: by doing so, the boot loader will load 'acpi-cpufreq' by default on every subsequent restart [27]. Both 'intel p-state' and 'acpi-acpifreq' can be seen in the schema below.

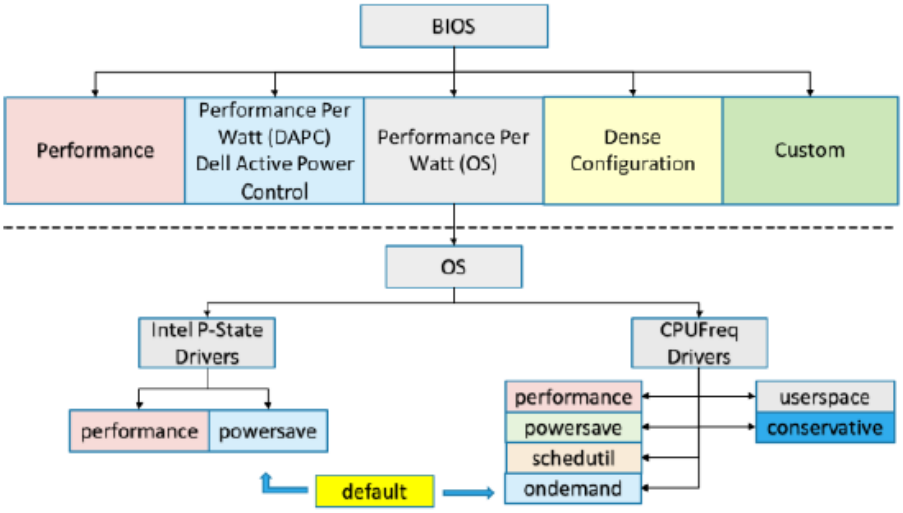

Fig 2.9 – A schema that show that, besides OS governors, there also BIOS drivers offered by hardware vendors that can control performance. Image from [28].

Once the 'userspace' governor is loaded in, another feature that becomes available to user-space programs is the possibility to alter P-States by manually inserting the desired processor's frequency in /sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed [29], an option that can also be taken advantage of by calling the corresponding Powerctl's function: of course such strategy introduces a higher latency than simply writing on the IA32_PERF_CTL MSR.

2.3.6 Hardware-Controlled Power States

The last mechanism user-space programs that want to set P-States should know about is HWP (Hardware-Controlled Power States). This feature, when enabled, allows the processor itself to choose its preferred performance profile according to its workload, which must be in accordance to hints and limits set by the OS.

The benefit of this approach is that a processor can choose a custom power state that is not tied to a specific P-State with its own fixed voltage and frequency, meaning that a more granular control is available to processors in a way that is simply not possible to both users and OS, which are only able to target a discrete number of states via the IA32_PERF_CTL register.

HWP is disabled by default when the system is powered on, but can be enabled by setting the bit 0 of IA32_PM_ENABLE [14]. Once enabled, OS can set its hints in the various fields of IA32_HWP_REQUEST (Figure 2.10), which include the desired performance and its maximum and minimum limits, the preferred bias between the

performance/energy efficiency spectrum, and the size of the activity window used by the processor to target its next power state. Each processor has its own copy of the register, meaning that it is possible to give each processor its own guidelines, while by writing on IA32_HWP_REQUEST_PKG it is possible to set a series of hints that are followed by all CPUs in a single package [14].

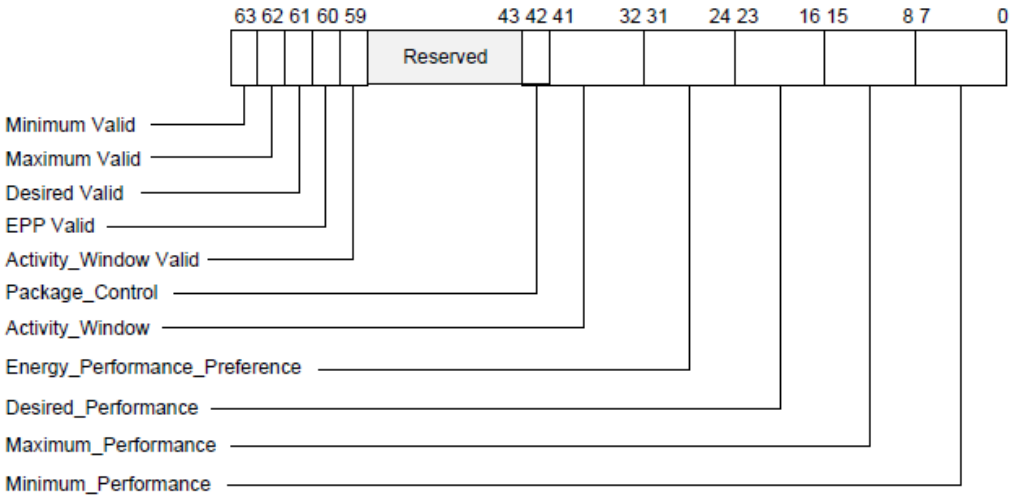

Fig 2.10 – The IA32_HWP_REQUEST register. Even though IA32_HWP_REQUEST_PKG allows to write hints that are followed by all the processors in a package, each IA32_HWP_REQUEST register also contains flags (bits 42, 59-63 as seen above) that allow single processors to ignore the directives imposed on the package [14].

In order to know which performance targets are possible for a processor, the OS can read the IA32_HWP_CAPABILITIES register, shown below.

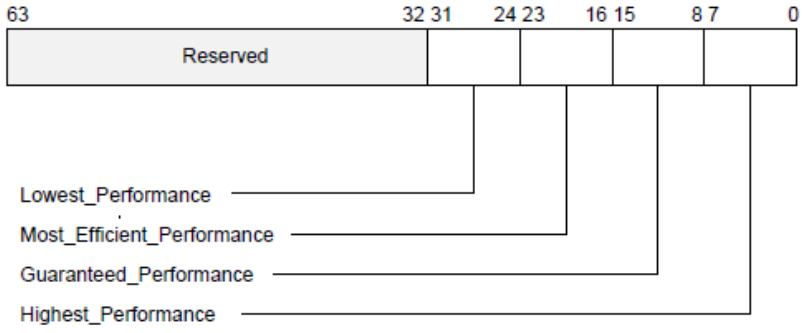

Fig 2.11 – The IA32_HWP_CAPABILITIES register, which enumerates its fixed capabilities that can be set by OS through IA32_HWP_REQUEST (lowest and highest performance), and its dynamic ones that change according to the current workload and external constraints (most efficient and guaranteed performance) [14].

Support for HWP can be checked by calling CPUID with EAX = 6. However, unlike previous features, it's not an all-or-nothing situation since some aspects of HWP may be available while other may not: in order to know which ones are accessible, users need to read from bit 7 through bit 11: more precisely, bit 7 indicates if HWP is supported, bit 9 and bit 10 if it possible to respectively set in IA32_HWP_REQUEST the activity window and the performance/energy bias, while bit 11 indicates if it possible to use IA32_HWP_REQUEST_PKG [20].

Powerctl does not currently support HWP but offers the usual function that makes the CPUID call on the user's behalf.

2.3.7 CPU Power States

Another way to alter a processor's power consumption is through CPU Power States (C-States), which dictate how much of a processor should be put to sleep when idle. There are various C-States, and what each of them precisely does may be different relative to the processor's architecture [30] [31], but generally:

- C0: Active. It's the default state where every component of the CPU is fully active. It's also the only C-State where a processor is able to change its P-State.
- C1: Halt. The processor halts its execution and disables its core clock. This state also offers a variant called C1E, which lowers the voltage and frequency of the processor as well.
- C2: Stop Clock. The processor disables both the core and bus clocks.
- C3: Deep Sleep. The processor directly stops the clock generator, and flushes its L1 and L2 caches.
- C4: Deeper Sleep. Same as C3, but the voltage and frequency is kept even lower.

As seen above, each subsequent C-State turns off more features than the previous one, but with a cost: the more is disabled in a C-State, the higher is the latency when waking up from that state. Also some processors may support C-States and sub C-States that go beyond C4 [31] [30].

C-States affect both components of the total power consumption formula to a higher degree than P-States and with lower latencies too, even though studies on recent Intel processors (Figure 2.12), such a Haswell and Skylake-SP, have shown that these latencies are slightly increasing over the years, and can go from 1.6-2.1 µs when waking up from C1, to ~40 µs when waking up from a deeper sleep state [21] [22].

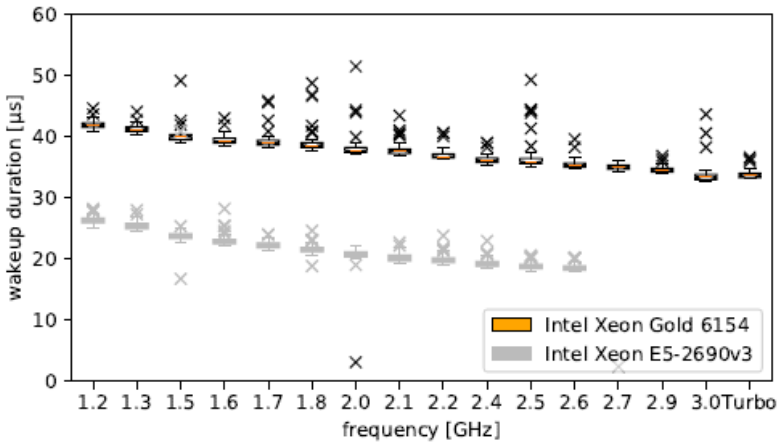

Fig 2.12 – Wakeup latencies from C6 in Skylake processors, which show latencies not only depend from how much deep is the sleep state, but also from the current frequency: lower ones cause longer delays. Image taken from [22].

2.3.8 MONITOR/MWAIT Instructions

Software can manually target a C-State by taking advantage of a pair of x86 instructions, MONITOR and MWAIT, which can be used only in kernel mode [32]:

- MONITOR: This instruction triggers the monitoring of an address that has been specified in input register EAX. If a store in that address range is detected, MONITOR is notified and wakes up any thread that is waiting with MWAIT, an instruction that each MONITOR needs to be paired with.

  This instruction also supports two additional inputs through registers ECX and EDX: the former is an optional extension, the latter an optional hint, but I genuinely couldn't find a case where a processor supports these two additional features, and what these optional extensions and hints may entail [32].
- MWAIT: If paired with MONITOR, any thread that encounters this instruction will put the processor in a optimized state until one of the three following events happen: MONITOR is triggered by a store in the monitored range address, or an external signal (like INIT# or RESET#) or a machine exception is detected.

  MWAIT supports two additional extensions through input registers EAX and ECX: the first allows to specify the C-State, and sub C-State within it, that needs to be targeted when entering the optimized state, while the second specifies if interruptions, masked ones included, should be allowed to interrupt the wait. If instead no MONITOR instruction is encountered beforehand, than MWAIT will work as a simple NOP, which means not doing any operation [32].

The support for this pair of instructions can be checked by calling CPUID with EAX=5 and seeing if bit 1 of ECX is set [20], or by using Powerctl, which actually changes C-States using this strategy.

2.3.9 Hardware Duty Cycling

Software that wishes to optimize performance in low active workloads can enable Hardware Duty Cycling (HDC), which allows processors to autonomously target a idle state, generally C3 or deeper [14].

Software can enable this feature by setting bit 0 in IA32_PKG_HDC_CTL, a copy of which is available for each physical package: it is then up to the processors inside to choose which components on the physical package need to be put to sleep (and with which severity) according to the intensity of the workload. By clearing that bit, instead, any component that was previously put to sleep due to HDC will awake: the idea is to keep enabling and disabling this register as software sees fit according to the obtained accumulated time that a package has spent during a HDC-forced sleep state [14].

This value can be read from register MSR_PKG_HDC_DEEP_RESIDENCY, which only keeps track of HDC residency time for one particular C-State at a time, and which is updated every time the package wakes from a HDC-forced sleep state: to change this tracked C-State, software need to indicate the C-State they want to track in the first two bits of MSR_PKG_HDC_CONFIG [14].

Each logical processor in a physical package also possess a IA32_PM_CTL1 register, whose first bit can be set in order to give that processor the ability to ignore the command to enable HDC: this ensures that HDC can be controlled in a granular fashion, despite that fact that it can only be enabled and disabled on a physical package level.

Support for HDC can be checked by calling CPUID with EAX=6 and observing if bit 13 of EAX is set [20]: Powerctl does not currently support HDC but as usual it offers a function that checks its availability.

2.4 Synchronization in Multi-Threaded Applications

As anticipated in the Introduction, Powerctl was created as a tool that allows to quickly request new processors' states during the synchronization process in multi-threaded applications, which have become common in the industry due to the rise of multi-core architectures, from expensive server farms to our mobile phones.

Synchronization is a necessary process that ensures data consistency when manipulating shared data in concurrency and parallelism. This is typically achieved by using a locking mechanism that allows only a thread at a time (mutual exclusion) to enter a specific portion of code (critical section) that modifies memory structures that are shared between multiple threads. But it is also an activity that always comes with a cost, since multiple threads can either spin on shared variables that indicate if they acquired the lock, leading to a waste of CPU cycles and possibly even to a large amount of memory contention, or simply go to sleep until it is their turn to access the critical section, which however leads to drops in performance as threads need time before they fully wake up and resume their operations [33].

As the scale of parallel machines steadily increases, trying to optimize this phase has never been so crucial for modern applications. This section will then introduce some of the most common spin-locking algorithms, including ones that were used to test Powerctl.

2.4.1 Test-and-set

Considered the most simple locking algorithm, Test-and-set (Figure 2.13) is based on using the homonymous Read-Modify-Write instruction 'test_and_set' to acquire the lock, which allows threads to atomically write to a memory location and get in return the old value stored there.

This value can be 0 when the lock is free to acquire, or 1 when it is in use by some thread. If a thread tries to acquire the lock, meaning that is trying to write 1 to this memory location, then it will only acquire it when the value that gets returned is 0, which is a situation that happens right after it has been released by the thread that previously held it.

This algorithm suffers from two setbacks: it may lead to starvation since threads acquire threads in no particular order, and there is no mechanism in place that ensures that a thread eventually acquires a lock, even though such scenario is highly unlikely. The other issue is memory contention, since threads keep saturating the bus by writing on the same shared variable [33]: in order to minimize the latter issue, Test-and-set can be deployed with a backoff delay that becomes exponentially bigger when a thread tries to unsuccessfully acquire the lock [34].

```
int lock = 0   // 0: Free, 1: Acquired

ACQUIRE-LOCK(*lock)
int backoff = 1
while (test_and_set(lock, 1) == 1)   // test_and_set returns old value: if this value is 1, lock is taken
    pause(backoff)                   // Thread waits so that it doesn't write again on the bus
    backoff = backoff*2              // Delay gets exponentially bigger with each failed attempt!

RELEASE-LOCK(*lock)
*lock = 0

// Space cost: O(l) where l is the number of locks
```

Fig 2.13 – Pseudocode of Test-and-set with exponential backoff.

2.4.2 Test-test-and-set

Another way to reduce the memory contention seen in previous algorithm is to ensure that threads request test_and_set instructions only when there is a concrete possibility they are able to acquire the lock, as shown in the pseudocode in Figure 2.14.

Each thread keeps reading a copy of the shared variable stored in their local cache, so to keep the bus free of activity regarding the lock contention. When a thread releases a lock by writing 0 in the variable, the bus is then used to overwrite the new value in all the local copies: only when threads detect this change there will be a situation similar to Test-and-set, where multiple threads try to write on the same variable at the same time, but it will be short-lived since one of them will quickly acquire the lock while the others will go back to reading their local copies, leading to a stop of traffic in the bus [35].

```
int lock = 0   // 0: Free, 1: Acquired

ACQUIRE-LOCK(*lock)
while (test_and_set(lock, 1) == 1)   // test_and_set returns old value: if this value is 1, lock is taken
    while (*lock == 1)       // Threads keep reading the value of the lock without trying to acquire it…
        continue             // … ensuring the bus is not saturated by parallel writing instructions!

RELEASE-LOCK(*lock)
*lock = 0

// Space cost: O(l) where l is the number of locks
```

Fig 2.14 – Pseudocode of Test-and-test-and-set.

2.4.3 The Ticket Lock

While Test-and-set with exponential backoff and Test-and-test-and-set try to fix the memory contention issue of the original algorithm they are both based on, they don't do anything to reduce the starvation problem, since there is no mechanism in place that ensures that eventually a thread will acquire the lock after requesting it. The Ticket Lock (Figure 2.15) aims to fix this issue by ensuring a FIFO service, where the lock is acquired by threads following the same order as when they requested it.

The lock is composed by two counters, one that keeps track of how many times the lock was released and a ticket that counts how many times it was requested: threads that want to acquire the lock increase this last counter by 1 through a fetch_and_add instruction, which atomically increments the value of the register and returns its old value. Threads then wait until this returned value is the same as the released locks counter before they can finally acquire the lock.

```
struct lock
    int ticket = 0         // Ticket system for threads that request the lock that ensures a FIFO policy
    int released = 0       // Keeps track of the latest released lock

ACQUIRE-LOCK(*lock)
int my_ticket = fetch_and_add(lock->ticket, 1)   // Thread requests its ticket, entering the FIFO list
while (my_ticket != lock->released)         // Thread keeps checking if it is its turn to acquire the lock
    delay(BASE_DELAY * (my_ticket - lock->released))      // The closer to the head of the FIFO list…
                                                         // … the smaller the delay!

RELEASE-LOCK(*lock)
lock->released = lock->released + 1         // Refreshes the number of released locks

// Space cost: O(l) where l is the number of locks
```

Fig 2.15 – Pseudocode of the Ticket Lock.

This algorithm also reduces the amount of write atomic instructions seen when a lock is released in Test-and-test-and-set, which could be still large enough in case of numerous contending threads. However, since threads keep reading the same counter to get access to the latest number of released locks, there is still the potential problem of memory contention, which cannot be resolved by introducing a exponential backoff delay like the one seen in Test-and-set, since a thread that gets delayed would also delay as well all the following ones due to the FIFO order. To minimize this issue, threads suffer a backoff delay that is proportional to the difference between the ticket number and the released locks counter [33].

2.4.4 Array-Based Queuing Locks

To counter the issue of memory contention, while regaining the FIFO policy, two array-based locking algorithms for cache-coherent systems were devised:

- In Anderson's Lock (Figure 2.16), a thread acquires a ticket by executing a fetch_and_add instruction where the counter is increased by 1, similarly to the Ticket Lock. The thread then uses the old returned value as an index to a shared array in order to access a unique cache line where it can spin until it acquires the lock. This portion of memory can either contain 1, which means the lock is already acquired, or 0, meaning that the lock is now free for the thread to grab it. When a thread releases the lock it writes 0 in the portion of memory indicated by its index plus 1, so that the next thread in line is notified that the lock is now free, ensuring a FIFO policy. The thread that acquires the lock will then write 1 in its cache line, so that future threads that will obtain access to that location will work correctly even when the ticket counter will start again from 0 in order to avoid overflow [34].

```
struct lock
    int ticket = 0    // Ticket system for threads that request the lock that ensures a FIFO policy
    int array[MAX_THREADS] = {0, 1, 1, …, 1}       // Each value is stored in a different cache line

ACQUIRE-LOCK(*lock)
int my_ticket = fetch_and_add(lock->ticket, 1)  // Thread requests its ticket, entering the FIFO list
while (lock->array[my_ticket] == 1)      // Thread keeps checking if it is its turn to acquire the lock…
    continue                             // … by reading the variable on their own dedicated cache line!
lock->array[my_ticket] = 1;              // Ensures lock will work correctly in the next cycle

RELEASE-LOCK(*lock)
lock->array[(my_ticket+1) mod MAX_THREADS] = 0      // Frees lock for the next thread…
                                                    // … and avoids overflow!

// Space cost: O(t*l) where t is the number of threads and  l is the number of locks
```

Fig 2.16 – Pseudocode of Anderson's Lock.

- Graunke and Thakkar's Lock (Figure 2.17) is a variant of Anderson's Lock that uses a 'proto' linked list instead of a ticket system. Threads that request the lock dynamically update the tail of the list by inserting a pointer to their own unique portion of memory, which is obtained from the address of the element of the shared array identified by using their virtual processor id as index.

  Threads update the list through fetch_and_store atomic operations, which give in return the old tail of the list: once updated the list, they spin on the location indicated by the previous tail. When a thread releases the lock, it writes 0 on its portion of memory, which will allow the next thread that is spinning on that location to acquire it, ensuring a FIFO policy [36].

The disadvantage of both approaches is that they have to allocate a fixed-sized array that inevitably increases the space needed to implement the locks since for *t* threads and *l* locks we have a cost of *O(t*l)*, whereas the previous algorithms were just *O(l)* [33].

```
struct lock
    int array[N_PROCESSORS] = {1, 1, 1, ..., 1}     // Each value is stored in a different cache line
    int *last_element = 0           // Pointer to the cache line of the last element of the list

ACQUIRE-LOCK(*lock)
// Thread adds a pointer to its slot as last element of the list, and gets the previous pointer
int* element_ahead = fetch_and_store(lock->last_element, &array[vpid])
while (*element_ahead == 1)    // Thread spins on the previous tail until the lock is released
    continue

RELEASE-LOCK(*lock)
lock->array[vpid] =  0              // Frees lock in its slot, so that next thread in line will be notified

// Space cost: O(t*l) where t is the number of threads and  l is the number of locks
```

Fig 2.17 – Pseudocode of Graunke and Thakkar's Lock.

2.4.5 The MCS List-Based Queuing Lock

MCS is a list-based lock algorithm (Figure 2.18) created with the intention of preserving all the advantages of the previous array-based algorithms, such as the FIFO queue and the lack of memory contention, while reducing the cost of the space that they needed to implement the lock.

This algorithm implements a shared explicit linked list, initially empty. When a thread requests the lock, it creates its own node and adds it to the list through a 'fetch_and_store' instruction, gaining in return a pointer to the previous last element of the list. At this point one of the following two cases can happen:

- There's no previous element because the list is empty, so the thread can simply go ahead and acquire the lock.

- A previous element exists, so the thread enters the FIFO line by adding a pointer to its node in the previous element and begins to spin in its node, whose value is set to 1.

```
struct node
    node* next          // Pointer to the next node
    int locked          // 0: Free, 1: Acquired

node* lock = null       // The list is initially empty

ACQUIRE-LOCK(*lock, *current)
// 'current' is a node that the thread initializes in its local memory
current->next = null                    // This new node will be the last element of the list
node* predecessor = fetch_and_store(lock, current)    // Adds node and gets the previous  one
if predecessor == null                  // There's no previous node: the thread acquires directly the lock
    current->locked = 0
    return
current->locked = 1                     // There's a previous node: the thread must wait until it is freed
predecessor->next = current             // The previous node now points to the current one
while (current->locked == 1)            // The thread spins on its own node until the lock is free
    continue

RELEASE-LOCK(*lock, *current)
if (current->next == null)              // Seems there's no next node in line
    if (compare_and_swap(lock, current, null) == 1)      // Checks if this is the only node in the list:
        return                                           // CAS returns 1 if so!
    while (current->next == null)       // There are other nodes, but not visible now due to delays:
        continue                        // thread spins until pointer to the next node is refreshed
current->next->locked = 0               // Thread releases thread on the next node, ensuring FIFO policy

// Space cost: O(t+l) where t is the number of threads and  l is the number of locks
```

Fig 2.18 – Pseudocode of the MCS Lock.

When instead a thread releases the lock, it first checks if there is another thread in line by checking the 'next' pointer of its node:

- If there is another node in line, then the thread releases the lock by writing 0 on the next node, so that the thread which it belongs to can be notified that the lock is now free.
- If there is no next node in line, then the thread might be the only element left and the linked list can return to be empty. To be sure, the thread tries to empty the list by executing a 'compare_and_swap' instruction, which deletes the list only if it is truly the last element in it. If this write fails, then it means another thread has actually requested the lock, but it is currently 'invisible' to the current one because

there is a latency between adding a node to the list and refreshing the 'next' field of the previous node. The current thread then waits until its 'next' field points to a valid node, and then releases the lock there by writing 0 when it has finally refreshed.

Since a requesting thread only need to spin in its own node that it dynamically allocates in its local memory, a location that then is shared to the previous node in the queue so that it knows where to release the lock, another advantage of MCS is that it works correctly on both cache-coherent and cache-incoherent systems [33].

2.4.5 The CLH List-Based Queuing Lock

CLH is another list-based algorithm (Figure 2.19) where the core difference to MCS concerns where the threads spin and release the lock.

The linked list is initially composed by a single 'neutral' node with a value of 0 that grants immediate access to the lock to the first thread that enters the queue. When a thread requests the lock, it allocates a new node in its local memory with value 1 and adds it to the list through a fetch_and_store operation, gaining a pointer to the previous last element of the list, like it happens in MCS, but with the key difference that instead of refreshing the previous node with a pointer to the current one, the vice versa happens: the current node points to the previous element. The thread then spins on the previous node instead of the one that it has just placed.

Threads that release the lock then simply write 0 in their node, notifying the next thread that is spinning on it that the lock is now free, once again ensuring the FIFO policy.

```
struct node
    node* prev              // Pointer to the previous node
    int locked              // 0: Free, 1: Acquired

node* lock = {null, 0}      // The list has initially a free lock

ACQUIRE-LOCK(*lock, *current)
// 'current' is a node that the thread initializes in its local memory
current->locked = 1                     // This new node will be the last element of the list
current->prev = fetch_and_store(lock, current)      // Adds its node and links to it the previous  one
while (current->prev->locked = 1)       // Spins on the previous node
    continue

RELEASE-LOCK(*current)
current->locked = 0                     // Thread releases in its own node

// Space cost: O(t+l) where t is the number of threads and  l is the number of locks
```

Fig 2.19 – Pseudocode of the CLH Lock.

CLH is less complex then MCS while retaining its same space cost, but since threads spin on locations that were dynamically allocated by other threads in their own local memories, it is an algorithm that can only work on cache-coherent machine [37] [38].

2.4.6 Waiting Policies

Spinlock algorithms, including both MCS and CLH, can be generally altered to support various waiting policies, which dictate further actions that a thread should follow when it fails to acquire the lock:

- Spin with pause. The thread notifies a processor that is has entered a spin loop through a PAUSE instruction, which has two benefits: the first is that it reduces power consumption by introducing a finite delay (or a series of NOP instructions in older processors) between each attempt to acquire the lock, which slows down power consumption (without changing any states) since otherwise threads would be able to spin very quickly leading to a waste of resources. The second benefit is that it improves performance duo to a series of optimizations made by the processor to reduce memory order violation issues, which occur when threads read values that then become outdated due to another thread's store while their operations are still pending, meaning that they need to be reverted since they cannot be committed anymore [32].
- Park. The thread simply goes to sleep until it is notified that the lock is free to acquire. In Linux systems this can be implemented through the 'futex' system call, which allows a thread to point out a memory location that needs to be changed to a desired value (in this specific case a free lock) before it can be awoken, in a mechanism that is very reminiscent of MWAIT/MONITOR but that does not alter the state of the processor, even though it still leads to a reduction of power consumption [39].
- Spin-then-park. A hybrid strategy that combines together 'spin with pause' and 'park'. A thread initially spins with finite delays until it reaches a certain threshold of tries, after which it goes to sleep. This mechanism ensures that a thread goes to sleep only if it is truly necessary, which is the case when it is better to deal with delays due to the awakening process then having the thread continuously fail to acquire the lock spin after spin.
- MWAIT/MONITOR. A waiting policy the uses the homonymous instructions, where the address of the lock is actually the location that needs to be monitored. With this strategy a thread needs to read the value of the lock twice in a loop: the first time to command the processor to enter a sleep state if the lock is not free, while the second is upon waking up to make sure MONITOR ended the sleep phase because it has correctly detected that the lock is free, which is not a situation

that can be given for granted since, as illustrated before, other external events can trigger it [40].

Powerctl was tested using MCS with the waiting policies described above, using different configurations of C, P and T-States.

3. Powerctl

This chapter properly introduces Powerctl, the kernel module that allows user-space software to tweak hardware states in a CPU, going in detail with all the technical and design choices that made possible to turn its features from paper to reality.

3.1 The Necessity of Programming a Linux Kernel Module

The first thing that must be noted is that some of the functions that Powerctl needs to provide cannot be implemented from a user-space point of view, since they often rely on writing and reading CPU model-specific registers (which is very important since we want to minimize latency as much as possible) or on executing privileged x86 instructions such as MONITOR and MWAIT. In order to have complete freedom to execute privileged instructions like those above and to reference any memory address, the running software needs to have unrestricted access to the hardware, and the only way to achieve that is to make it run under kernel mode.

Linux is a open-source OS, where the functionalities of its kernel are provided by multiple modules that can be freely loaded and unloaded at any moment's notice according to the current necessities of the system (Figure 3.1). The advantage of this approach is that it is possible to create new modules and dynamically load them in without the necessity of rebuilding the whole kernel [41].

By implementing the core functionalities of Powerctl through a kernel module written in C, it is then possible to provide them without those user-space restrictions. From a programming point of view instead, the disadvantages of being kernel-side are that the programmer can't rely anymore on those user-space libraries they are accustomed to, and that uncaught exceptions in the code can lead to issues such as memory corruption or fatal crashes of the system instead of a safe termination of the program.

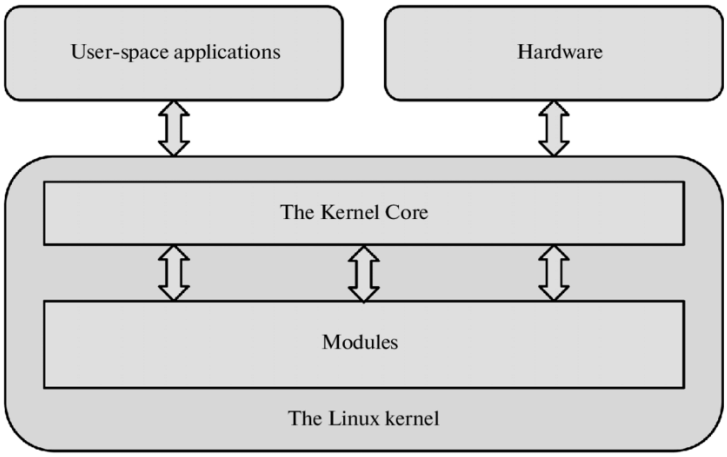

Fig 3.1 – The Linux Kernel. Despite its modular nature, there is still a monolithic core component behind the kernel that guarantees basic functions such as memory and processes management. Schema taken from [42].

Users themselves can directly check the current loaded modules by typing 'lsmod' on their terminal (as shown in Figure 3.2): the returned information consists in the name of the modules, their size in bytes and the number of programs that are currently utilizing their services. They can also add new ones with the 'insmod [module name]' command and remove them using 'rmmod [module name]' [41].

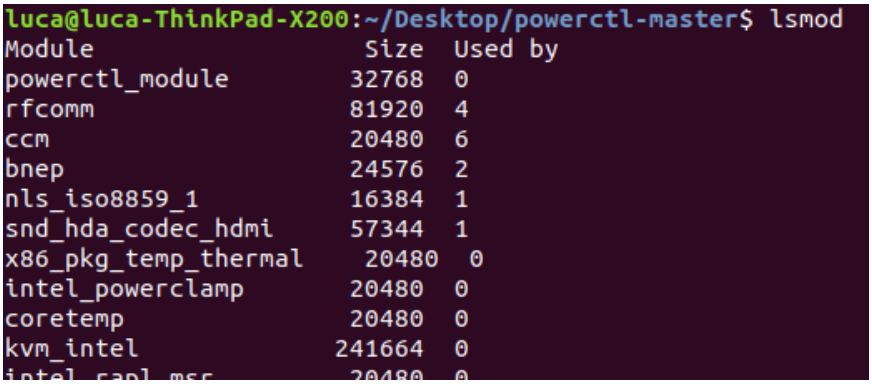

Fig 3.2 – 'lsmod' in action; Powerctl can be seen at the top of the list.

Regardless of its contents, when writing a new module it is mandatory to include <linux/kernel.h> and <linux/module.h> in the headers, and define a pair of functions that dictate what the module has to do when it is loaded and unloaded, respectively called 'int init_module' and 'void cleanup_module' (Figure 3.3). Alternatively, it is possible to use custom names for these functions preceded by the suffixes "__init" and "__exit", and then later indicate their role by calling 'module_init([function name])' and 'module_exit([function name]) and using their names as parameters. Once the module is ready, it is also necessary to create a Makefile that can correctly install and remove the module in lib/modules, which is the directory used by the kernel to load its modules. The result of this process is a file with extension .ko that can then be loaded as module through the command 'insmod' (Figure 3.3) [41].

```
#include <linux/kernel.h>
#include <linux/module.h>

static __init load_module(void) {
    // Do stuff when the module is loaded
}

static __exit unload_module(void) {
    // Do stuff when the module is loaded
}

module_init(load_module)
module_exit(unload_module)
```

```
obj-m += [module name].o

all:

    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) modules

clean:

    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) clean
```

Fig 3.3 – Skeleton of a Linux kernel module, both in its code and Makefile.

3.2 Implementing an Interface for User-Space Software

Simply developing Powerctl as a kernel module is not enough to provide its services to the user: there is still the need of an interface that exposes its functionalities so that they can be requested on-demand by user-space software.

Since there is already a mechanism in place that allows users to request services from kernel through special invocations called system calls, a possible way to implement this interface is ensuring that Powerctl adds new system calls when it is loaded in as a module, which directly write on the MSRs responsible of the state transitions.

This approach allows to target new states with a lesser degree of latency compared to other solutions based on the utilization of the pseudo-filesystem (Figure 3.4), which require the manipulation of special files through the 'ioctl' system call [43]. This is a key benefit when dealing with situations where higher levels of overhead are not affordable, such as when requesting new states on the fly during the back-off phase of a spinlock synchronization algorithm.

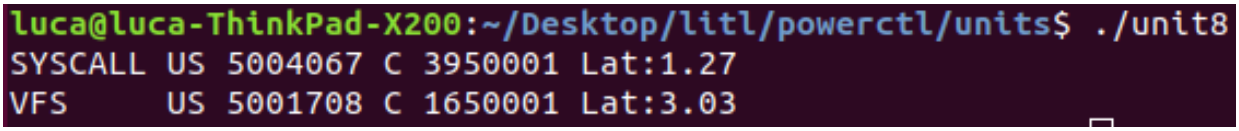

Fig 3.4 – Changing P-States through a system call that directly writes on the responsible MSR allows to halve the latency (expressed above in microseconds) compared to changing it through pseudo-filesystem.

3.2.1 System Calls in Linux

Before we explain how these system calls are added, it is first necessary to understand how the whole mechanism works. In Linux each system call is identified by its number, which generally can go from 0 to 254: these ids are then used by threads to call the desired system calls, either explicitly through functions like 'syscall' in C alongside other possible additional parameters, or implicitly by using functions that act as system calls 'wrappers' that do the calls on the user's behalf, and that can be found in libraries such as 'glibc', which provides the core C libraries for systems such as Linux ones [41] [44].

No matter the approach, the result is that on an Assembly level programs store the id of the requested system call in the CPU EAX register, while other registers such as EBX and ECX are reserved for additional parameters if needed. Programs then notify the kernel thanks to a specific interrupt (int 0x80) that causes a switch from user to kernel mode. The kernel, thanks to a dispatcher, will then use the input id as an index to access a slot inside the 'system call table', an array stored in kernel space with pointers to all the system calls currently registered (Figure 3.6). After retrieving the address of the desired system call, the dispatcher will execute it using as parameters those stored previously in the registers by user-space programs, while EAX will be used for the returned value. Obviously the context and data structures of the calling threads are preserved in a stack so that they can

be restored once control returns to them, making possible to resume their operations from where they were left off [41]. The life cycle of a system call is shown in the picture below.

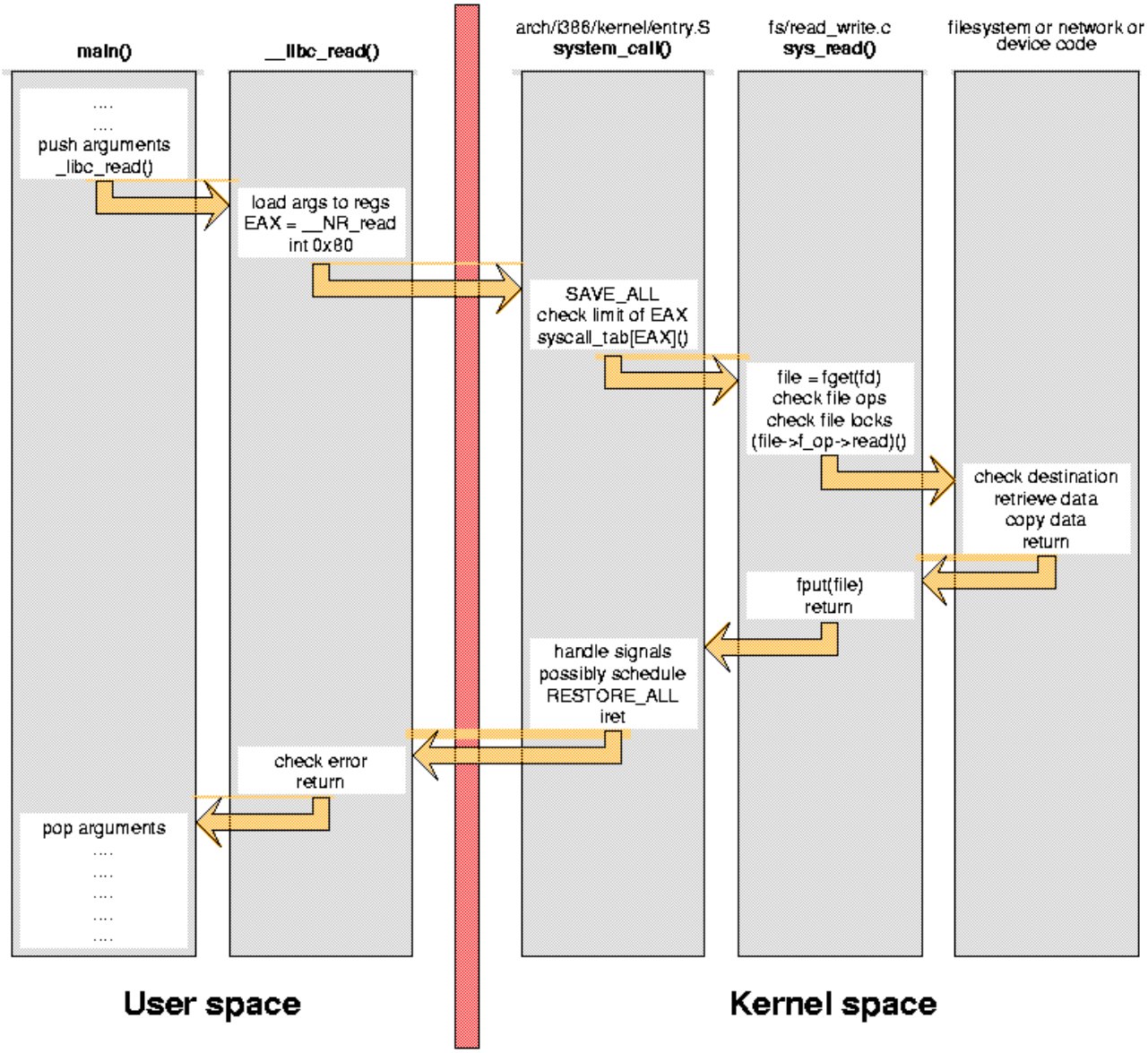

Fig 3.5 – The complex life cycle of a system call. In this specific case, the user requested to read a file. Image from [45].

3.2.2 Injecting New System Calls

The implication of the mechanism described in the previous paragraph is that new system calls not only need to be defined, but their addresses also have to be added to the system call table under an id, otherwise the kernel won't know how to access them when they are needed. But since the length of arrays cannot be dynamically changed, an easier solution is simply swapping the addresses of unneeded system calls with those of the newest ones: in particular there is a system call called 'sys_ni_syscall' that acts as a placeholder for the implementation of future system calls [46], and that is referenced multiple time across the

table, making it the ideal candidate for a substitution. That is the approach chosen for Powerctl.

Id	Addr	Name
0	0000ffff81093aa0	sys_restart_syscall
1	0000ffff810844b0	sys_exit
2	0000ffff815604c9	stub32_fork
3	0000ffff8119d9d0	sys_read
4	0000ffff8119da80	sys_write
5	0000ffff811f2600	compat_sys_open
6	0000ffff81199bc0	sys_close
7	0000ffff8105df00	sys32_waitpid
8	0000ffff81199e40	sys_creat
9	0000ffff811afdf0	sys_link

Fig 3.6 – The first few system calls pointed by the system call table. Adding Powerctl's custom ones is just a matter of finding and replacing the addresses that point to 'sy_ni_syscall'. Image taken from [47].

However, another (and final) step is required: CPUs are forbidden from writing on read-only locations such as the system call table, even when operating under kernel mode, if bit 16 of register CR0 is set; this register is responsible of enabling and disabling CPU features such as caching and paging through its flag bits, so it is necessary to clear that bit before modifying the table [20].

The system calls themselves can be implemented using the macro __SYSCALL_DEFINEx(x, sname, __VA_ARGS__) defined in <linux/syscalls.h> [48], whose parameters are respectively the number of additional arguments, the actual name of the system call and then the series of arguments, all separated by commas between types and variables' names. After the system calls are defined, it is possible to store their addresses in variables of type 'unsigned long' so that they can be later inserted in the 'system call table' when the module is loaded.

For what concerns the table and 'sys_ni_syscall', getting their addresses can be tricky since they change at each boot for security reasons due to the Address Space Layout Randomization (ASLR) [49]. However, their addresses can still be recovered using their symbol names in /proc/kallsyms, which is a file that exposes the list of symbols alongside their addresses that are recognized by the kernel, and that is generated at each boot of the system, meaning its contents are always up to date [41]. Users can then find their locations by typing 'sudo cat /proc/kallsyms | grep sys_call_table' and 'sudo cat /proc/kallsyms | grep sys_ni_syscall' on the command line, and then manually inserting their addresses in unsigned long variables in the program. Powerctl however performs automatically these commands upon executing the Makefile, and stores these addresses with their associated symbol names in the system.map, another symbol table used by the kernel but static unlike /proc/kallsyms [50], so that the addresses can be later retrieved by the module at run time during the loading phase.

Once all these addresses are properly stored in the program, the swapping can take place in the module initialization function following this algorithm (Figure 3.7): the program cycles through all the pointers of the table; when it finds one that points to 'sys_ni_syscall', it saves that index and proceeds, terminating the search when there have been enough occurrences of 'sys_ni_syscall' as the number of the system calls that need to be added. The module then clears bit 16 of CR0 so that it is able to write on the table, and swaps all the occurrences of 'sys_ni_syscall' in the locations corresponding to the saved indexes: it is also important to make sure that these indexes are stored in a global variable, so to restore the table to its previous form when unloading the module. It is always good practice to revert every long-term change that a module applies upon removing it, especially in this situation where there may be other modules that want to introduce new system calls by following this same strategy. Once modified the table, the module restores bit 16 of CR0, enabling once again the protection of read-only locations.

```
// Gets addresses from the system.map
unsigned long _syscall_table = SYSCALL_TABLE;
unsigned long _ni_syscall = NI_SYSCALL;

int restore[N_SYSCALLS];          // Contains the indexes of the table that point to sys_ni_syscalls

// New System Calls
__SYSCALL_DEFINEx(2, _set_modulation, …)        {   // Do stuff   }
__SYSCALL_DEFINEx(2, _set_pstate, …)            {   // Do stuff   }
// Etc.

// Create pointers to the addresses of the new System Calls
unsigned long sys_set_modulation_ptr  = (unsigned long) __x64_sys_set_modulation;
unsigned long sys_set_pstate_ptr  = (unsigned long) __x64_sys_set_pstate;
// Etc.
// NOTE: The names on the left are automatically generated once the syscalls are defined!

__init load_module(void) {
    int swaps = 0;
    for (int i = 0; i < 255; i++) {                 // Iterates through the whole table
        if (syscall_table[i] == _ni_syscall) {      // We have found one occurrence of sys _ni_syscall
            restore[swaps] = i;                     // We remember the index for later
            swaps += 1;
            if (swaps == N_SYSCALLS) {              // We have covered all the new system calls
                break;
            }
        }
    }

    // Clears bit 16 of CR0 and enables writes on the read-only table
    // We swap the entries!
    syscall_table[restore[0]] = sys_change_modulation_ptr;
    syscall_table[restore[1]] = sys_set_pstate_ptr;
    // Etc.
    // Sets bit 16 of CR0 and re-enables read protection on the table
}

__exit unload_module(void) {
    // We restore the table like it used to be
    // Clears bit 16 of CR0
    syscall_table[restore[0]] = _ni_syscall;
    syscall_table[restore[1]] = _ni_syscall;
    // Etc.
    // Sets bit 16 of CR0
}
```

Fig 3.7 – How Powerctl adds and removes its system calls as it is respectively loaded and unloaded as a module. For brevity only two system calls are shown.

When it is loaded, Powerctl adds a suite of system calls that include:

- _set_modulation: it changes the clock modulation of the processor according to the T-State chosen by the user.
- _set_pstate: it changes the frequency and voltage of the processor according to the P-State chosen by the user. It also accepts as input a user-space memory location where it is stored the delay that occurs between the request of a new state and it actually being implemented by the system.
- _read_hwp_fbck: it returns the values of the registers IA32_MPERF and IA32_APERF to an user-space memory location given in input, so that programs can implement the feedback mechanism described in section 2.3.3.
- _set_cstate: it puts the processor to sleep according to the C-State chosen by the user through a locking waiting policy based on the MONITOR/MWAIT pair of instructions. Users also need to give in input the address location that will be used by MONITOR to trigger MWAIT upon a new store.

3.2.3 Retrieving the New System Calls' Numbers

Even though the last paragraph has illustrated how Powerctl adds its suite of system calls, another problem still remains: user-space software need to know their syscall numbers to access them, which may not be as trivial as it seems. These ids dynamically depend from where the table stores the addresses of 'sys_ni_syscall', which may change depending on the kernel version and which may be influenced by previous modules that have inserted their own suite of system calls.

To ensure these new system calls can be reached correctly and dynamically at run time, Powerctl imports <linux/proc_fs.h> and takes advantage of /proc, a file system used by the kernel to access information and stats about its running processes [41]. During the loading phase and after modifying the system call table, Powerctl creates a new directory in /proc where it stores a number of entries, each one with the purpose of returning the id of a system call when they are read. When creating a new entry, it is necessary to specify its name, the parent directory, the file access permission value and a data structure that describes which operations need to performed when the entry is read or written [51]: in our case, we only need the read operation, which returns the number of the assigned system call (which the entry can suppose by its own name) by retrieving it from kernel space and copying it to user-space. Once again, in order to undo any long-term change in the system, these entries and the directory where they reside are removed when the module is unloaded. This algorithm is shown in Figure 3.8.

The last step then is simply defining a series of macros in user-space, where each one returns the number of a system call by reading the assigned entry in /proc.

```
// Function that is executed when an entry is read
static ssize_t read_proc(struct file *filp, char *buf, … ) {
    int res = -1;
    // The entry retrieves the position of the assigned system call's id using its name
    for(int i =0; i < NUM_SYSCALLS; i++)
        if (!strcmp(filp->f_path.dentry->d_iname, subdirs[i]))
            res = i;        // This value will be used to retrieve the id of the assigned system call
    }
    if (res != -1) {
        char pbuf;                                          // Buffer that will contain the id of syscall
        int len = sprintf(pbuf, "%d\n", restore[res]);      // Stores the id in the buffer
        copy_to_user(buf, pbuf, len);                       // Returns the id to user-space
    }
}

// Structure that indicates what to do when reading and writing on an entry
struct proc_ops  proc_fops = {
    proc_read: read_proc,
    //proc_write
}

char* subdirs[NUM_SYSCALLS];                                // Contains all the names of the entries
struct proc_dir_entry *parent;                              // Points to the parent directory

__init load_module(void) {
    // Creates new directory in /proc
    parent = proc_mkdir("syscall_table_patcher", NULL);
    // Creates entries inside new directory and stores their names
    proc_create("sys_set_modulation",    0444, parent,  &proc_fops);
    subdirs[0] =  "sys_set_modulation";
    proc_create("sys_set_pstates",       0444, parent,  &proc_fops);
    subdirs[1] =   "sys_set_pstates" ;
    // Etc.
}

__exit unload_module(void) {
    // Removes entries inside the new directory
    remove_proc_entry("sys_set_modulation",  parent);
    remove_proc_entry("sys_set_pstates",    parent);
    // Etc.
    // Removes new directory in /proc
    remove_proc_entry("syscall_table_patcher",    parent);
}
```

Fig 3.8 – How Powerctl ensures the ids of the new system calls can be recovered from user-space software. The instructions in both loading and unloading functions are performed right after changing the system call table.

3.3 Interacting with Model-Specific Registers

Until now, we have established various times that it is required to interact with model-specific registers located on the CPU in order to change states or even to be able to modify the system call table, but without never going in detail in how this procedure works.

3.3.1 Reading and Writing

These 64-bits registers, which not only allow to toggle certain CPU features but can also be important for debugging, can be read and written through two privileged instructions of the x86 instruction set, 'rdmsr' and 'wrsmr':

- 'rdmsr' allows to read the MSR register whose 32-bit address is specified in the input register ECX. Its contents will then be stored in registers EDX and EAX: the first will contain the high-order 32 bits of the register, while the second its 32 low-order ones [32].
- 'wrmsr' allows to write the contents that are stored in input registers EDX and EAX on a MSR register whose 32-bit address is specified in ECX: exactly like 'rdmsr', EDX contains the high-order bits, EAX the low ones [52].

In the C language, it is possible to implement those two instructions through 'asm', which is an inline assembler that allows the insertion of low-level code in the program. 'asm' also allows user to specify a list of clobbers, memory addresses beyond those specified as the output whose data may be changed during the execution of the instruction: the inclusion of such list prevents compilers to perform optimizations on the memory that may mess with the results of the computation. Further optimizations can also be forbidden by extending 'asm' with the expression 'volatile' [53].

Powerctl offers functions to read and write on a MSR thanks to a pair of system calls (shown in Figure 3.9), that are loaded alongside the ones that deal with power control following the same procedure seen in the previous paragraph:

- _read_msr: Allows users to read a MSR. It takes as input two values: an integer that indicates the address of the register, and an unsigned long pointer to a memory address in user-space where the contents of the register can be stored so that users can read it. Before reading the register, this system call creates an array of two integers, which will be used to store the contents of EDX and EAX. After reading, the system call transfers their contents in the user-space input address through a particular macro called __put_user.
- _write_msr: Allows users to write on a MSR. It takes as input two values: an integer that indicates the address of the register, and an unsigned long pointer to a memory address in user-space that contains the data the user wants to store. Before writing, this system call creates an array of two integers on which it is stored the user-space

input data, which is accessed thanks to a macro called __get_user. The data on this array will then used to initialize the contents of EDX and EAX.

```
// Reads data from a register
static void mu_rdmsr(unsigned value, unsigned *hi, unsigned *lo) {
    asm volatile (
        "rdmsr"                                 // Name of the command
        : "=a" (*lo), "=d" (*hi)                // Output locations
        : "c" (value)                           // Id of the register we want to read
        : "memory"                              // Clobbers
    );
}

// Writes data inside a register
static void mu_wrmsr(unsigned value, unsigned *hi, unsigned *lo) {
    asm volatile (
        "wrmsr"                                 // Name of the command
        :                                       // Output locations: empty in this case
        : "a" (*lo), "d" (*hi), "c" (value)     // Input locations
        : "memory"                              // Clobbers
    );
}

__SYSCALL_DEFINEx(2, _read_msr, unsigned int, value, unsigned long*, address) {
    unsigned int read[2];                       // read[0] = lo, read[1] = hi
    mu_rdmsr(value, read+1, read);              // Reads register
    unsigned long *result = (unsigned long*) read;  // Creates a 64-bit pointer to 'read'
    __put_user(*result, address);               // Writes the result back in user space
}

__SYSCALL_DEFINEx(2, _write_msr, unsigned int, value, unsigned long*, address) {
    unsigned int write[2];                      // write[0] = lo, write[1] = hi
    unsigned int *input = (unsigned int*) address;  // Converts address from unsigned long to uint
    get_user(write[0], input);                  // Extracts data from user-space
    get_user(write[1], input+1);
    mu_wrmsr(value, write+1, write);            // Writes on register

}
```

Fig 3.9 – How Powerctl reads and writes on MSRs.

3.3.2 Accessing Single Bits and Fields

Now that we have established how to access model-specific registers, the next problem becomes how to interact with only certain sections of bits, which is especially important during a store since we don't want to accidentally overwrite other bits that are used for

enabling or disabling features beyond the desired ones. In order to avoid potential unforeseen consequences, Powerctl employs these two strategies:

- One solution is to create a data struct that allows to parse the contents of a MSR by defining unsigned integer variables inside it that are directly mapped to the various fields of a MSR's 64-bit memory space, following the same order as they appear, from the lowest bits to the highest ones. Since it is possible to specify the size in bits of these variables by adding '*:x*' after their declaration, where *x* is the bit size, they can target individual bits or a continuous succession of them. After reading a MSR, it possible to access the fields of interest by casting to the appropriate struct the integer array that contains the data of EDX and EAX. Before writing on a MSR, instead, it is possible to write on single bits or fields after casting to the appropriate struct the integer array that is used to initialize the input registers EDX and EAX.

```
// Structure of IA32_CLOCK_MODULATION
typedef struct parsing {
    // LOW REGISTER (EAX)
    unsigned int duty_cycle     :4;     // Clock Modulation Settings
    unsigned int enable         :1;     // Enables Clock Modulation
    unsigned int reserved1      :27;    // Reserved bits

    // HIGH REGISTER (EDX)
    unsigned int reserved2;             // Reserved bits
} parsing;
```

Fig 3.10 – An example of a parsing struct; this one parses the IA32_CLOCK_MODULATION MSR.

- The other solution for targeting precise bits during a store is bit masking. A bit mask allows to define which bits need to be set or cleared after it is applied to the contents of a register through a bitwise operation [54]. To give an example, we have established that is mandatory to clear bit 16 of CR0 in order to write on the system call table: to avoid clearing other bits as well, it is possible to create a mask with a hexadecimal value of 0xFFFEFFFF, which in binary translates to the value 11111111111111101111111111111111, where the only 0 belongs to bit 16. After reading CR0, this mask can then be applied to its contents through a 'AND' operation, and the resulting data will be written back: features that are already enabled won't be disabled because '1 AND 1' still returns 1, unlike bit 16 that will be cleared because '1 AND 0' will result in a 0. The same strategy can be followed for the other direction: when all the system calls are added, bit 16 is enabled once again from applying a mask with value 0x0001000 through a 'OR' operation, since '0 OR 1' gives 1 while all the other features remain unchanged regardless of their status.

3.4 Power Control

Now that we have introduced all the moving pieces that allow Powerctl to offer an interface to users so they can change processors' hardware states, we can finally analyze the functions that implement such changes.

3.4.1 Changing T-States

As we have seen previously, users can target a throttling state by calling the _set_modulation system call, which relies on the function described below.

```
void change_state(int mode) {
    unsigned int regs_w[2];
    parsing *prs;

    // Restores register
    regs_w[0] = 0;
    regs_w[1] = 0;

    // Checks if the user wants to enable Clock Modulation, otherwise it continues to stay disabled
    if (mode != DISABLE_CLOCK_MODULATION) {
        prs = (parsing*) regs_w;
        prs->enable = 1;

        // Selects the right setting
        if (mode == REQUEST_DUTY_CYCLE_25)
            prs->duty_cycle = DUTY_CYCLE_25;          // DUTY_CYCLE_25    =  0x0100b
        else if (mode == REQUEST_DUTY_CYCLE_37_5)
            prs->duty_cycle = DUTY_CYCLE_37_5;        // DUTY_CYCLE_37_5  =  0x0110b
         else if (mode == REQUEST_DUTY_CYCLE_50)
            prs->duty_cycle = DUTY_CYCLE_50;          // DUTY_CYCLE_50    =  0x1000b
        else if (mode == REQUEST_DUTY_CYCLE_62_5)
            prs->duty_cycle = DUTY_CYCLE_62_5;        // DUTY_CYCLE_62_5  =  0x1010b
        else if (mode == REQUEST_DUTY_CYCLE_75)
            prs->duty_cycle = DUTY_CYCLE_75;          // DUTY_CYCLE_75    =  0x1100b
        else if (mode == REQUEST_DUTY_CYCLE_87_5)
            prs->duty_cycle = DUTY_CYCLE_87_5;        // DUTY_CYCLE_87_5  =  0x1110b
        else
            prs->duty_cycle = DUTY_CYCLE_12_5;        //DUTY_CYCLE_12_5   =  0x0010b

    }

    // Writes the current settings to the register that handles Clock Modulation
    mu_wrmsr(IA32_CLOCK_MODULATION, regs_w+1, regs_w);
}
```

Fig 3.11 – How Powerctl handles clock modulation. For brevity, the states are only shown in increments of 12.5%.

This function follows the same mechanism described in paragraph 2.3.2, meaning that it enables or disables clock modulation by setting or clearing the 'On-Demand Clock Modulation Enable' bit on the IA32_CLOCK_MODULATION MSR, while setting the value belonging to the chosen T-State in the 'On-Demand Clock Modulation Duty Cycle' field. In order to accomplish this, the function relies on a parsing struct to write on the correct fields which was described in the previous paragraph.

It also must be noted that this function clears all reserved bits on the register when choosing a new clock modulation profile: the reason is that at the moment of writing this relation, no additional feature relies on those bits to be enabled or disabled, so they can be cleared without involuntary repercussions on the processor. Should this change in the future, the function should be modified to include reading the register before writing on it, so to retain the values of those reserved bits while changing only the 'Enable' and ' Duty Cycle' fields: of course this would inevitably increase the latency time.

3.4.2 Changing P-States

Upon a request for a new P-State, the _set_pstate system call activates an auxiliary function (Figure 3.13) whose first action is to retrieve the 16-bit control value from the input id of the targeted P-State. This action alone is trickier than it seems since, as we have seen in paragraph 2.4.3, the control values for the P-States depend from the CPU's architecture, so the module first needs to extract the correct ones from the ACPI tables stored in the main memory before being able to set new performance configurations.

This is why the Powerctl's Makefile, upon execution, calls 'acpidump', a command that extracts ACPI tables stored in physical memory to dump them in a file in the ASCII format: these tables are internally used by the OS in order to gain information about the features of hardware and processors, and include the _PSS tables (also known as TPSS or LPSS tables) which contain control values for the first 16 available P-States, alongside their targeted frequency and voltage [55]. Once dumped these tables, the Makefile extracts the wanted ones into a raw binary file by using the command 'acpixtract' and converts them to a human-readable format by using the command 'iasl' (as shown in Figure 3.12) [56]. A Python script is then launched to parse the information about the P-States and store it in an auxiliary file, so that the module can later retrieve the P-States' control values through their respective ids.

However, not always those tables can be found on the system: in order to avoid this potential obstacle, a fallback script is launched that initially sets 'userspace' as the OS governor and then reads all the available frequencies that the processors can target in /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies. Once retrieved this list, the script sets those frequencies simultaneously in all the processors one by one by writing them in /sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed, exactly like

described in paragraph 2.3.4. Once a new global frequency is set, its respective control value can then be read from any PERF_STATUS MSR and stored in an auxiliary file, so that it can be recovered later by the module.

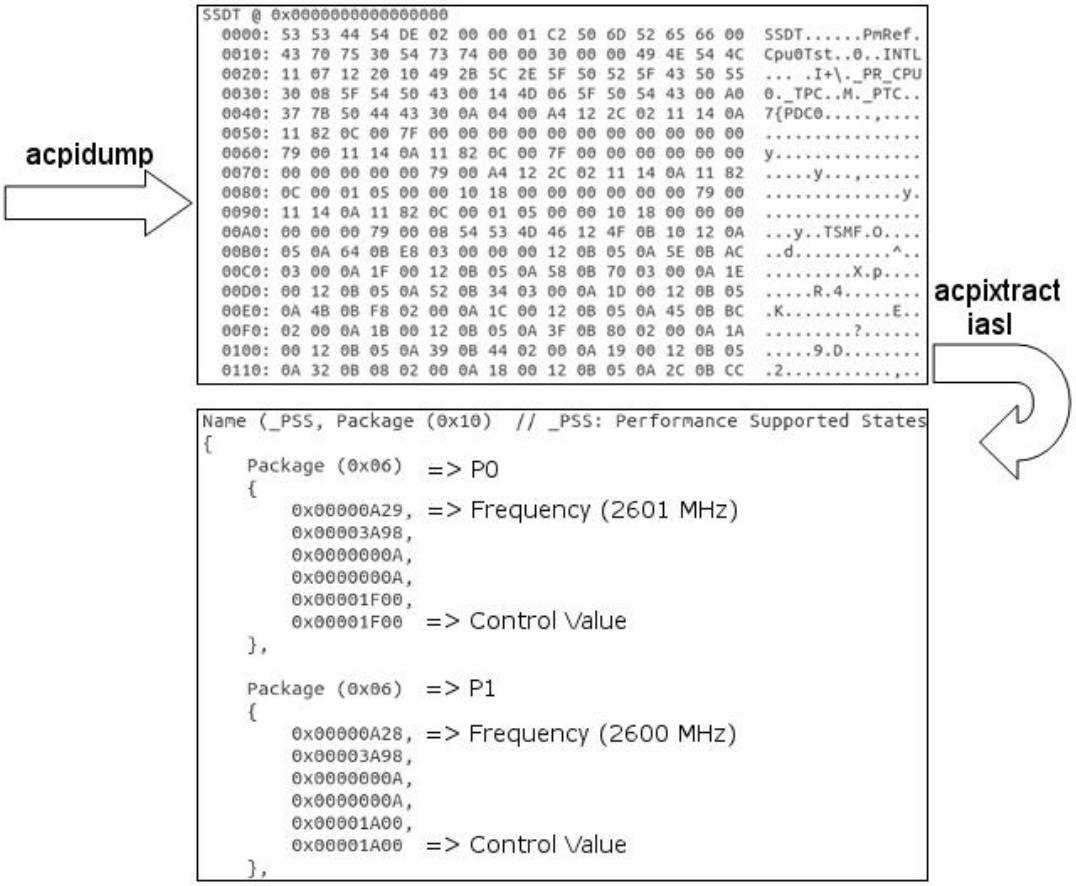

Fig 3.12 – An image that show the process necessary to get the list of P-States supported by the system. After executing 'iasl' on the binary file output by 'acpixtract', it is possible to obtain a human-readable version of the tables where P-States can be parsed from.

Once the control value is retrieved, the following actions depend from if the user indicated as input an unsigned long address where the latency of the transition will be stored:

- If no address is specified, then the control value is directly written in the first 16 bits of IA32_PERF_CONTROL and the function returns.
- If an address is specified, then the function saves the current timestamp and reads IA32_PERF_STATUS in order to gain the last P-State acknowledged by the system. By using a mask to recover the first 16 bits, the function compares its control value with the one of the new requested state. If they are different, then the function writes the new state in IA32_PERF_CONTROL and keeps reading IA32_PERF_STATUS until the change is recognized by the system. When it finally does, it returns the latency by subtracting the new current timestamp with the previous saved one.

Similarly to the function described in the previous paragraph and following the same motivation, the preserved bits of IA32_PERF_CONTROL are not maintained.

```
void change_pstate(int value, unsigned long* address) {
    unsigned int regs_r[2], int regs_w[2], old_state, new_state;
    unsigned long long ts;

    // Gets the control value of the P-State that the user wants to set, and saves it in regs_w
    get_pstate_control(value, regs_w+1, regs_w);
    regs_w[1] = 0x0;                                        // Clears the high part of the register

    // The user is not interested in the latency: directly writes the new P-State and returns
    if (address == NULL) {
        mu_wrmsr(PERF_CONTROL,  regs_w+1, regs_w);
        return;
    }

    ts = rdtsc();  // Reads the current value of the processor's time-stamp counter
    mu_rdmsr(PERF_STATUS, regs_r+1, regs_r);  // Reads the current P-State and saves it in regs_r
    old_state= regs_r[0] & PERF_CONTROL_MASK;  // Obtains the control value of the old P-State
    new_state = regs_w[0] & PERF_CONTROL_MASK;  // Obtains the control value of the new state

    // Checks if new requested state is different from the previous one
    if (old_state != new_state) {
        mu_wrmsr(PERF_CONTROL, regs_w+1,  regs_w);    // Writes the new state

        // Cycles until PERF_STATUS acknowledges the new state
        while (old_state == (regs_r[0] & PERF_CONTROL_MASK))
            mu_rdmsr(PERF_STATUS, regs_r+1, regs_r)
    }

    ts = rdtsc() – ts                                       // Gets the latency
    __put_user(ts, address)                                 // Writes the latency in user-space
}
```

Fig 3.13 – How Powerctl handles P-State's transitions and calculates the their latency.

3.4.3 Changing C-States

Upon the request of a new C-State, the _change_cstate system call (shown in Figure 3.14) initially maps the MWAIT input hints and wakeup modes to their control values (which are fixed for x86 processors), and then implements the waiting policy based on the utilization of MWAIT and MONITOR exactly like it is described in paragraph 2.4.7, where the address of the lock is given as input by the user.

In Powerctl, MWAIT and MONITOR are implemented by importing <asm/mwait.h> and respectively calling the functions __mwait and __monitor, which make use of the inline assembler 'asm' to execute this pair of privileged instructions [57].

```
__SYSCALL_DEFINEx(4, _change_cstate, unsigned long * volatile, lock, unsigned long, hint,
unsigned long, wakeup_mode, int, thread_id) {
    unsigned long HINT, WAKEUP_MODE

    // Sets target C-State
    if (hint == 1)
        HINT = TARGET_C2_STATE;          // C2: Stop Clock.
    else if (hint == 2)
        HINT = TARGET_C3_STATE;          // C3: Sleep.
    else if (hint == 3)
        HINT = TARGET_C4_STATE;          // C4: Deeper Sleep.
    else
        HINT = TARGET_C1_STATE;          // C1:  Halt.

    // Sets wake-up mode
     if (wakeup_mode == 1)               // Treat interrupts as break events even if masked.
            WAKEUP_MODE = ALLOW_INTERRUPTS_AS_BREAKS;
    else
        WAKEUP_MODE = DISABLE_INTERRUPTS_AS_BREAKS;

    while (1) {
        __monitor(lock, 0, 0);           // MONITORS the input address

        if (*lock)                       // Checks if the lock is free: if so, it exits from the loop.
            break;
        else
            __mwait(HINT, WAKEUP_MODE);      // Enters in the target T-State

        if (*lock)                       // Checks if the thread has awaken for the right reason.
            break;
    }
}
```

Fig 3.14 – How Powerctl targets C-States.

3.5 The Probing System

Since threads request new states that end up changing the behavior of the whole processor, we inevitably have situations where other threads that do not target alternative performance profiles are nonetheless subjected to such changes if assigned to a depowered CPU, leading to a downgrade to their throughput. To give an example, a scenario where a thread requests a 12.5% clock modulation setting and then terminates without restoring the clock modulation to its 100% capacity can have a drastic impact on the whole system, since all the following threads will be forced to run with that target state: a situation that is permanent until another thread scheduled in that very same CPU requests a new T-State.

That is why Powerctl implements a probing system that, by inspecting the threads' identities after a context switch, it is able to assign the correct performance profiles to the threads that actually requested them, and to restore a processor to its default settings if it turns out a requesting thread has died without reverting the changes itself.

This section will then go in detail in how the probe works and how it remembers the various requests for new settings thanks to a read-mostly hashtable.

3.5.1 The Read-Mostly Hashtable

Powerctl keeps track of the states' requests through a hashtable, whose entries contain the following values: the PID of the thread requesting the changes (which acts as the key), and the targeted T and P-States. There's no need to also store C-States since the system call that implements the waiting policy based on the pair MONITOR/MWAIT has only a temporary effect on the system that goes no further than the synchronization process.

However, since Powerctl needs to be used in the context of multi-threaded applications, the hashtable has to be a thread-safe object, so that it can be safely read and written concurrently by multiple threads without introducing unintended behaviors. At the same time, making so that only a thread at a time can access the hashtable via the utilization of locks can have dire consequences in terms of latencies, especially since this is an operation that is supposed to be executed by the probing system every time a new thread is scheduled in order to load the correct performance profile. The solution then lies in the utilization of a read-mostly hashtable, which relies in a read-copy-update (RCU) synchronization strategy that ensures quick reads that avoid locks of any kind [58].

In order to explain this lock-less synchronization strategy, we can take as an example a linked list, which is an apt comparison since in the read-mostly hashtable defined inside the Linux kernel each bucket consists in a series of entries where each node points to the next one, starting from the head [58]:

- When adding a new entry inside a specific position, the list simply allocates a new node that points to the next element of the list, and then makes the previous one point to the new current one.
- When modifying an already existing entry, the list allocates a copy of the entry, changes its value, makes it point to the same next element of the original one, and then refreshes the previous entry to point to the new copy. Once no reader is currently reading the list, the original entry is de-allocated.
- When deleting an existing entry, the list makes the previous element point to the next element of the entry that needs to be deleted. Once no reader is currently reading the list, the entry is de-allocated.

The consequence of these strategies is ensuring that a reader always reads a correct version of the list due to the fact that it is impossible to deference a null pointer since

entries are de-allocated only when no reader is detected. The downside, however, is that the entries encountered inside the list itself may depend on when the list is being read, if before or after the pointers are refreshed due to a change, which may be a problem for threads that can't afford to read outdated information: a case that does not concern Powerctl since each thread is only interested in reading and writing their own entry, meaning that a thread can't read an entry that is being currently updated by another one.

It also must be noted that if a read-mostly hashtable can guarantee lock-less synchronization between a single writer and multiple readers, it can't promise the same between multiple writers that are updating the table at the same time, hence why it is called 'read-mostly' [58]. To fix this shortcoming, Powerctl deploys a read-mostly hashtable that uses locks for each bucket that must be acquired by a thread in order to modify them: this a decision that inevitably raises the latency time, but only to a certain and limited degree since in our case read requests are way more frequent then write requests, which explains the key objective to not make reading rely on locks.

Linux defines a read-mostly hashtable that can be implemented kernel-side by importing <linux/hashtable.h>, which is initialized by Powerctl upon loading: it also must be noted that each entry needs to include a hlist_node variable, which represents the node inside the bucket where the entry will actually be stored and that is mapped automatically upon insertion [59]. The initialization of the table is shown below.

```
// Declaration of the hashtable. The second value means we have 2^BUCKET_BITS buckets.
DEFINE_READ_MOSTLY_HASHTABLE(threads, BUCKETS_BITS);

// Mutex locks used in order to modify the buckets of the hashtable.
struct mutex etx_mutex[BUCKETS];

// Struct of an entry inside the hashtable
struct entry {
    unsigned int pid;
    int t_state;
    int p_state;
    int padding;                // Padding so that each entry fits inside a 32 byte cache line
    struct hlist_node node;     // Each entry is automatically mapped to a node inside a list/bucket
};

__init load_module(void) {
    // This is right after modifying the system call table
    hash_init(threads);                 // Initialization of the hashtable
    for (x = 0; x < BUCKETS; x++)       // Initialization of the locks
        mutex_init(&etx_mutex[x]);
}
```

Fig 3.15 – How Powerctl initializes the read-mostly hashtable

3.5.2 Updating the Hashtable

New entries are added or modified by Powerctl's system calls every time a thread requests new throttling and power states, ensuring the hashtable is always up to date.

```
// Adds a new entry to the hashtable or modify an already existing one using the thread's pid
int add_entry(unsigned int pid, int t_state, int p_state) {
    struct entry *e, *cur;
    int modified = 0;

    // Checks if this thread already requested a previous T-State/P-State
    u32 key = hash_32(pid, 3);                          // Obtains the bucket's key from the pid
    hash_for_each_possible_rcu(threads, cur, node, key) {  // Cycles through each bucket's element
        if (pid == cur->pid) {
            if (t_state != DONT_CHANGE && t_state != cur->t_state) {    // Needs to refresh T-State!
                cur->t_state = t_state;
                modified = 1;
            }
            if (p_state != DONT_CHANGE && p_state != cur->p_state) {  // Needs to refresh P-State!
                cur->p_state = p_state;
                modified = 1;
            }
            if (modified)
                return CHANGED_ENTRY;
            return NO_CHANGES;
        }
    }

    // First time this thread requests a T-State/P-State. Adds new entry to hashtable!
    e = kmalloc(sizeof(struct entry), GFP_KERNEL);      // Allocates space for a new entry
    e->pid = pid;
    if (t_state == DONT_CHANGE)
        e->t_state = NO_VALUE;
    else
        e->t_state = t_state;
    if (p_state == DONT_CHANGE)
        e->p_state = NO_VALUE;
    else
        e->p_state = p_state;
    mutex_lock(&etx_mutex[key]);                        // Acquires the lock before adding the entry
    hash_add_rcu(threads, &e->node, key);               // Adds the entry to the hashtable
    mutex_unlock(&etx_mutex[key]);                      // Releases the lock
    return ADDED_ENTRY;
}
```

Fig 3.16 – How Powerctl adds new entries or modifies existing ones inside the read-mostly hashtable.

```
__SYSCALL_DEFINEx(2, _change_modulation, int, t_state, unsigned long*, address) {
    // Gets the pid_t of the current thread
    unsigned int pid = (unsigned int) current->pid;
    // Adds the thread to the hashtable alongside its request or alter an already existing node
    add_entry(pid, t_state, DONT_CHANGE);
    // Sets the T-State wanted by the thread
    change_tstate(t_state);
}

__SYSCALL_DEFINEx(2, _set_pstate, int, p_state, unsigned long*, address) {
    // Gets the pid_t of the current thread
    unsigned int pid = (unsigned int) current->pid;
    // Adds the thread to the hashtable alongside its request or alter an already existing node
    add_entry(pid, DONT_CHANGE, p_state);
    // Sets the T-State wanted by the thread
    change_pstate(p_state);
}
```

Fig 3.17 – How Powerctl's system calls update the hashtable upon receiving a new request. Both change_tstate and change_pstate functions were described respectively in paragraphs 3.4.1 and 3.4.2.

Lastly, there are two additional functions used by the probing system: one to check if a particular entry exists so it can recover its selected states, and the other to delete an existing entry upon detecting its corresponding thread has been killed.

```
// Checks if an entry is present inside the hashtable and if so returns its values.
int * is_present(unsigned int pid, int *results) {
    struct entry *cur;
    // Checks if the entry is present inside the bucket associated to the key
    u32 key = hash_32(pid, 3);
    hash_for_each_possible_rcu(threads, cur, node, key) {
        if (pid == cur->pid) {
            results[0] = cur->t_state;
            results[1] = cur->p_state;
            return results;
        }
    }
    // Nothing was found
    results[0] = NO_ENTRY;
    results[1] = NO_ENTRY;
    return results;
}
```

Fig 3.18 – How Powerctl checks if an entry is present inside the hashtable.

```
// Deletes directly an entry inside the hashtable regardless of its values
int delete_entry(unsigned int pid) {
    struct entry *cur;
    // Checks if the entry is present inside the bucket associated to the key
    u32 key = hash_32(pid, 3);
    mutex_lock(&etx_mutex[key]);
    hash_for_each_possible_rcu(threads, cur, node, key) {
        if (pid == cur->pid) {
            hash_del_rcu(&cur->node);
            kfree(cur);                                  // Frees memory
            mutex_unlock(&etx_mutex[key]);
            return DELETED_ENTRY;
        }
    }
    // Nothing was found
    mutex_unlock(&etx_mutex[key]);
    return NO_DELETION;
}
```

Fig 3.19 – How Powerctl deletes an already existing entry inside the hashtable.

3.4.3 The Kretprobe

Now that have introduced the hashtable, we can consider the other side of the coin: the probing itself. Probes are tools that allow to monitor kernel routines, and which are triggered when certain conditions are met: once triggered, they are able to run their own custom routines which can very useful for debugging or performance monitoring [60].

In Linux systems, two types of probes are available:

- Kprobes, which monitor single lines of codes by inserting, upon registration, a "breakpoint" instruction in their first bytes. When this "breakpoint" instruction is encountered, control switches to the probe, which then calls a pre-handler function which is executed right before the probed line, and a post-handler function right after that line is executed. In both handler functions, the probe is able to recover the contents saved inside the CPU registers: a feature that makes very clear why probes are used for debugging, since they are able to inspect registers before and after an instruction is executed [60].
- Kretprobes, which are a variant of kprobes that monitor functions instead of single lines of codes. Upon registration it possible to specify the name of the function that needs to probed (as long as its symbol name can be recognized by the kernel), while its pre-handler is executed at the entry point of the function and its post-handler when it returns. Kretprobes are also able to recover the contents of the CPU registers in its handler functions [60].

Since Powerctl needs to inspect the identities of threads after a context switch, the most sensible option is to use a kretprobe that monitors 'finish_task_switch': a function that executes clean up actions right after a context switch, and from which it is possible to extract from the CPU registers the identity of the previous running thread before the switch [61].

Kretprobes can be implemented by including <linux/kprobes.h>; they are then initialized with four parameters: the name of the probed function, the names of the pre-handler and post-handler functions, and the max number of instances of the same function that can be probed at the same time. In Powerctl's case, we only need to specify a pre-handler function, since 'finish_task_switch' overwrites the identity of the previous running thread in the CPU register during its execution, making the implementation of a post-handler function absolutely useless in this situation [62].

Once initialized and after the name of the function that needs to be monitored is specified, the kretprobe can be safely registered when the kernel module is loaded. Likewise, the probe is unregistered when the module is unloaded.

```
static struct kretprobe krp_fts = {
    // Pre-handler that is invoked at the entry of a function
    .entry_handler = pre_handler_finish_task_switch,
    /* Post-handler that is invoked when a function returns
    .handler
    Max instances of the function that can be probed
    .maxactive   */
};

__init load_module(void) {
    // This is right after modifying the system call table
    krp_fts.kp.symbol_name = "finish_task_switch";    // Assigns the name of the probed function
    ret_fts = register_kretprobe(&krp_fts);           // Registers the probe
}

__exit unload_module(void) {
    // This is right after modifying the system call table
    unregister_kretprobe(&krp_fts);                   // Unregisters the probe
}
```

Fig 3.20 – How Powerctl registers and unregisters the probe upon respectively loading and unloading.

3.5.4 The Pre-Handler Function

Now that we know how a kretprobe is triggered, it is time to examine its pre-handler function (shown in Figure 3.21) that is executed at the entry point of 'finish_task_switch', whose first step is to recover the PIDs of both the current and previous thread.

The current thread's one is easy enough to obtain since that thread is the one actually running the pre-handler function, so it is just a matter of extracting it from 'current', which is a pointer that always leads to the task_struct of the current running thread: this is a data structure that contains info and stats about a certain thread, including its PID and life-cycle status [63].

Getting the PID of the previous thread is instead a more complex task since it requires extracting it from the CPU registers, which the kretprobe is able to access to. When 'finish_task_switch' is called after a context switch, it accepts only one input parameter, the task_struct of the thread that was just de-scheduled [61]. When passing these input parameters in the CPU registers, the x86 architecture always follows the same convention, which is storing the first parameter in register %rdi [54]: this means that by extracting the data stored inside it, and by casting it to a task_struct, it is possible to retrieve the PID of the previous thread.

Now that the pre-handler knows the identities of both threads, it proceeds as follows:

- If both threads never asked for a T or P-State, meaning the no entry belonging to them is found in the hashtable, then the pre-handler function does not change anything, since it assumes the previous thread has already took the correct precautions when it was its time to run the pre-handler.
- If, instead, either of them or both asked for T or P-State, which again can be checked from their entries in the hashtable, then it means a new performance profile is necessary. It may be because we are in a possible situation where the current thread that did not ask for a particular state is succeeding a thread that did ask for a new state, so it needs to revert to the default configuration, either by restoring 100% clock modulation or by targeting P0 (which we consider the default P-State) or even both, so it can work with downgrading its output. Or we may be in the opposite situation, where the current thread is the one who did ask for a state, while the previous didn't, so it needs to restore its targeted states in order to not waste power during synchronization. Or, finally, we may be in the situation where both threads asked for specific states, so the current thread simply needs to enforce its preferred settings.

Once the correct configuration is loaded, the pre-handler needs to execute one last step before returning: if it detects that the previous thread has died by checking its task_struct and there is an entry inside the hashtable belonging to it, then it deletes that entry. This is a very important decision, because PIDs are recyclable in the Linux OS after a threshold value is reached: this implies that, without precautions, there could be a situation where a newly created thread is forced to a configuration that it never asked for simply because it shares the PID of a previous dead thread that, instead, asked for those settings [64].

```
// Pre-execution function handler: this executes at the entry of finish_task_switch
static int pre_handler_finish_task_switch(struct kretprobe_instance *ri, struct pt_regs *regs) {
    struct task_struct *prev;          /* Pointer to the task_struct that was used to call
                                       finish_task_switch: it contains the ID of the previous thread! /*
    pid_t prev_pid, current_pid;                // Will contain PID of previous and current thread
    int prev_pid_states[2]; current_pid_states[2]    // Will contain both states for both threads
    int prev_pid_state, current_pid_state;     // Will contain one of the two states for both threads

    prev = (struct task_struct*) regs->di;        // Gets back from *regs the parameter *prev
    prev_pid = prev->pid;                       // Gets the ID of the previous thread
    current_pid = current->pid;                 /* Gets the ID of the current thread ('current' is the
                                                task_struct of the current thread) */
    // Gets the states of both previous and current thread
    is_present((unsigned int) prev_pid, prev_pid_states);
    is_present((unsigned int) current_pid, current_pid_states);
    for (int i = 0; i < 2; i++) {
        // If (i == 0) we are taking T-States, else (i == 1) we are taking P-States
        prev_pid_state = prev_pid_states[i];
        current_pid_state = current_pid_states[i];
        // Skip to the next step if both threads didn't previously request any T-State or P-State
        if ((prev_pid_state == NO_VALUE || prev_pid_state == NO_ENTRY) && (current_pid_state ==
NO_VALUE || current_pid_state == NO_ENTRY))
            continue;
        // If we are here, a new change is required: the current thread sets its requested T /P-State
        if (i == 0) {
            if (current_pid_state == NO_VALUE || current_pid_state == NO_ENTRY)
                change_tstate(DISABLE_CLOCK_MODULATION);
            else
                change_tstate(current_pid_state);

        }
        if (i == 1) {
            if (current_pid_state == NO_VALUE || current_pid_state == NO_ENTRY)
                change_pstate(DEFAULT_PSTATE, NULL);
            else
                change_pstate(current_pid_state, NULL);
        }
    }
    // Checks if a previous thread that requested a T-State or P-State is now dead
    if (prev_pid_states[0] != NO_ENTRY && !pid_alive(prev))
        // It's important to remove dead threads because old PIDs are 'recyclable'
        delete_entry(prev_pid);
    return 1;
}
```

Fig 3.21 – How the pre-handler function of the kretprobe ensures the right states are always ensured for each thread.

4. Experimental Study

In this chapter, I investigate the capabilities of Powerctl by utilizing its suite of functions in benchmark tests that will help us assess its effectiveness of tuning CPU states during spin-lock phases on the throughput and energy consumption of multi-threaded applications. First I'll introduce the pair of tools that together made conducting these benchmarks tests possible, then I will explain my methodology and finally I will present the results while discussing and reviewing them.

4.1 Benchmark Tools

In order to accurately test the potentiality of Powerctl, I made use of this pair of benchmarks tools:

- Standing for Library for Transparent Lock interposition, LiTL is a tool that allows executing programs that rely on Pthread Mutex locks where those locks are substituted with other locking algorithms chosen by the user at run time. A key characteristic of LiTL is that even though the base distribution of the library already includes the most common locking algorithms, the program is built in such a way that users themselves can add their own locking algorithms and waiting policies with relative ease. Thanks to this it was possible to add custom waiting policies for MCS that rely on Powerctl's system calls to dynamically change processors' hardware states during synchronization. In this way, programs that use the Mutex lock could easily be tested though LiTL without changing their code to make them aware of Powerctl [65].
- Lockbench is a tool that offers a benchmark suite for evaluating lock implementations. It allows to generate benchmark tests where the user can tweak many key factors, such as the execution sizes of both the critical and non-critical sections of code in terms of microseconds, how many cores should be tested, how long should the tests last, how many times should they be repeated inside a batch of tests and so on. Another feature of Lockbench is its integration with LiTL, making possible to execute benchmarks tests that use lock implementations defined in LiTL, including Powerctl's custom ones. Lockbench also includes a series of scripts that allow users to aggregate the results of a batch of benchmarks tests and to visualize them into a series of charts that make understanding the data more accessible and intuitive [66].

4.2 Methodology

Now that that all the relevant tools have been properly introduced, I can explain my methodology. From the beginning it was already clear that in order to accurately test

Powerctl, the tool had to be evaluated under a variety of scenarios that differ in the number of active cores, in the μs length of the critical and non-critical sections of code, and in the waiting policies involved. By examining all these variations, it is possible to highlight the conditions where Powerctl falls short and where it reaches its full potentiality in terms of how it affects power consumption and throughput relative to the default scenarios that do not involve the selection of new CPU hardware states during synchronization.

To meet this objective, I have added in LiTL variations for the existing Spin-and-pause, Park and Spin-then-park waiting policies, where each new variation targets a specific T or P-State while a thread is waiting to acquire the lock. When the lock is finally acquired, the thread will then restore the processor by disabling Clock Modulation or by targeting P0. I also added brand new waiting policies that target different C-States by relying on the MWAIT/MONITOR operations following the same strategy described in paragraph 3.4.3. This selection of waiting policies was chosen because, even though my priority was to test a pure spinlock implementation like Spin-and-pause, I still wanted to study the effects of parking, since such strategy does not actually change any processor's state by itself. These waiting policies were then made available for MCS since, as we have seen in paragraph 2.4.5, it can be considered the state-of-art in terms of spinlock algorithms

Each waiting policy was then tested under four benchmark scenarios in Lockbench: one where both critical and non-critical sections are very short (max 3.7 μs for each), one where the critical is longer than the non-critical (max 366 μs for the first, max 3.7 for the second), one where the opposite happens and, lastly, one where both sections are relatively long (max 366 μs for each). To be more precise, each scenario was tested with MCS using the default version of the waiting policies at P0/T0, so to obtain performance data that acts as reference points for all the following tests, and then with the custom waiting policies, so to target all the possible configurations of states: with T-States, it meant starting with a Duty Cycle set at 93.5% and finishing with one at 6.25%, scaling down by steps of 6.25, while with P-States it meant going from P0 to P15, exhausting most, if not all, the frequency and voltage configurations that a processor can target. Lastly, with C-States it meant going from C0 to C4. Each scenario was also executed under a variety of max active cores, and was repeated for 5 times in order to minimize the possible effects of outlier executions, each one lasting for 10 seconds.

For each benchmark test, I looked mainly at three factors:

- The throughput, expressed in terms of the average of lines of codes executed during the critical section per second.
- The energy consumption, expressed in Joules and calculated by difference of the values obtained from /sys/class/powercap/intel-rapl/intel-rapl:x/energy_uj, which are read twice: before and after the test.

- The overall efficiency, expressed by the ratio between throughput and energy consumption: the higher is the value, the more efficient is the execution [67].

Lastly, the system I used to perform these benchmark tests was equipped with an Intel Xeon Silver 4210 processor at 2.20 GHz, with 20 physical cores equally divided on two NUMA nodes. In order to properly calculate the energy consumption of a benchmark test, I summed the extracted energy consumption's metrics from both nodes. The CPU drivers and OS governor were respectively put to 'acpi-cpufreq' and 'userspace', and any form of boosting was disabled, so to leave full control of the P-States to Powerctl. Also hyper-threading was turned off as well, so to avoid inconsistencies that arise when having two logical cores mapped to the same physical one that target two different T or P-States: in the case of T-States it means that both cores target their lowest or highest setting between their chosen ones, while for P-States it means they do not trigger the new state at all [14].

4.3 Results and Discussion

This section contains the results I have obtained from the benchmark tests, and that will be shown here in a series of charts while providing my comments. However, I first wanted to preface that, in order to make these charts readable in this format, I had to make two concessions. The first was to visualize only a selection of T or P-States, so to not 'flood' the charts with a overwhelming number of plots that are barely distinguishable from each other. The second was to restrict the y-axis to the min and max values of the dataset they are representing. Starting the y-axis from 0 for all the charts would have been certainly the more correct choice, especially in order to give the reader a uniform sense of scale over metrics such as the throughput and the energy consumption; my hope is that the plots that visualize the efficiency will help alleviate this issue.

4.3.1 Results – T-States

Beginning with MCS-Spinlock, the benchmark scenario where both critical and non-critical sections are short (max 3.7 μs each) showed that enabling duty modulation can cause significant drops in throughput relative to the default spinlock: a performance drop that is higher the more severe is the duty modulation applied, culminating in the spinlock with a 12.5% duty cycle setting that sees a ~80% drop compared to the default one. This drop can also be perceived in the energy consumption, especially when a higher number of active cores is involved, but to a lesser degree that makes the default spinlock still the most efficient option. My first guess was that, since the lock is handed over very quickly due to the short length of both sections, there is a lot of overhead and slowdowns that come from continuously writing the IA32_CLOCK_MODULATION MSR, both before and after acquiring the lock, which makes this strategy simply not feasible. After adding a new waiting policy that does not alter the duty modulation, but still writes on the register as

often as the other waiting policies, I could confirm that writing on the register does actually cause tangible impacts on performance. An impact that obviously also depends from the duty modulation applied since these frequent write operations become increasingly slower as we go through these settings. Here are the relevant charts:

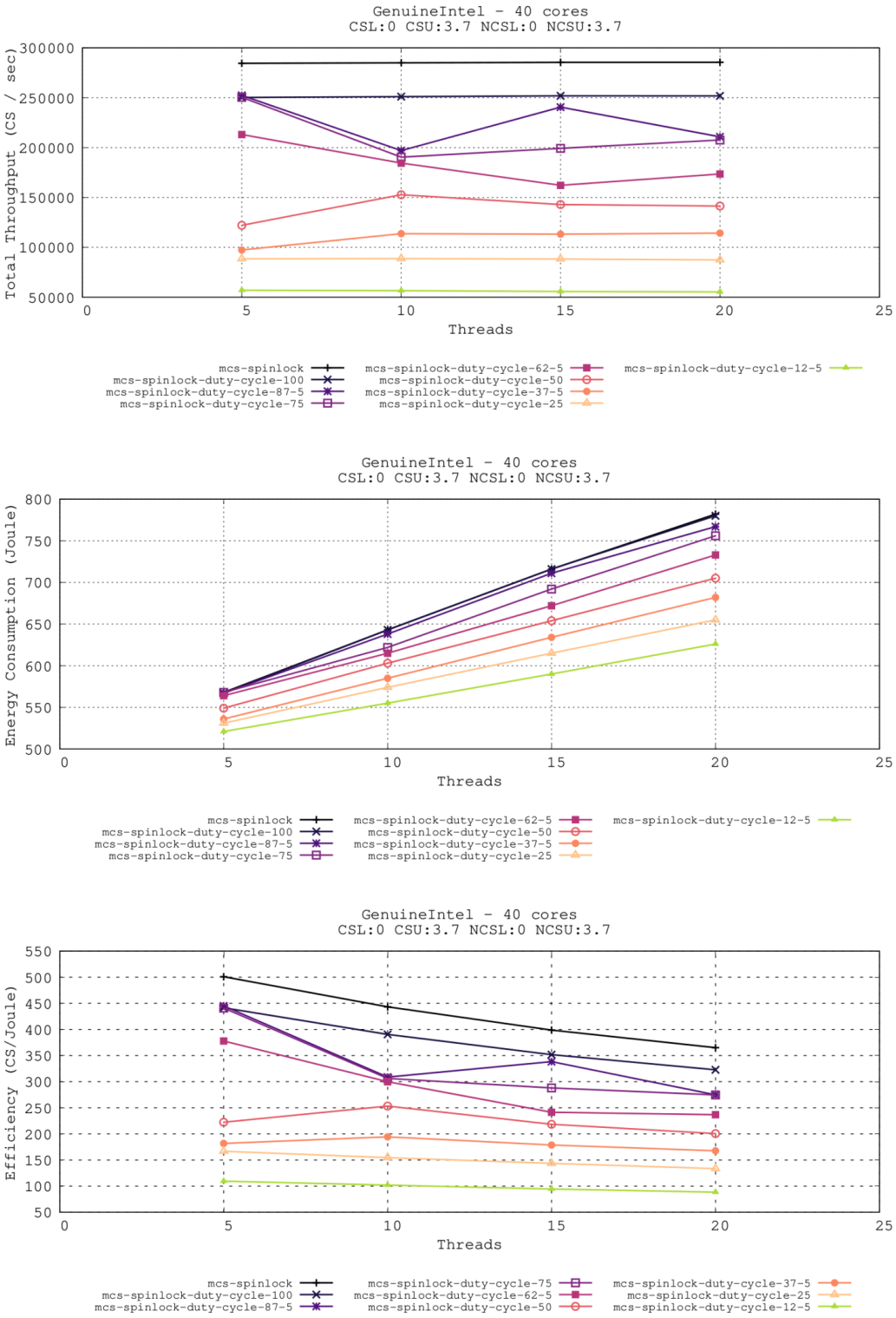

Fig 4.1 – Results for Spinlock MCS in benchmark with CS:0-3.7, NCS:0-3.7 for T-States.

Increasing the non-critical section of code to 366 μs helps reducing the drops in performance, since in this scenario write operations on the MSR are less frequent due to the fact that the synchronization occupies only a small portion of the computation. However the small size of the critical section prevents significant gains in the energy consumption as well: in terms of efficiency it means that our custom policies are still ineffective, even though not as harmful as in the previous scenario. The relevant charts:

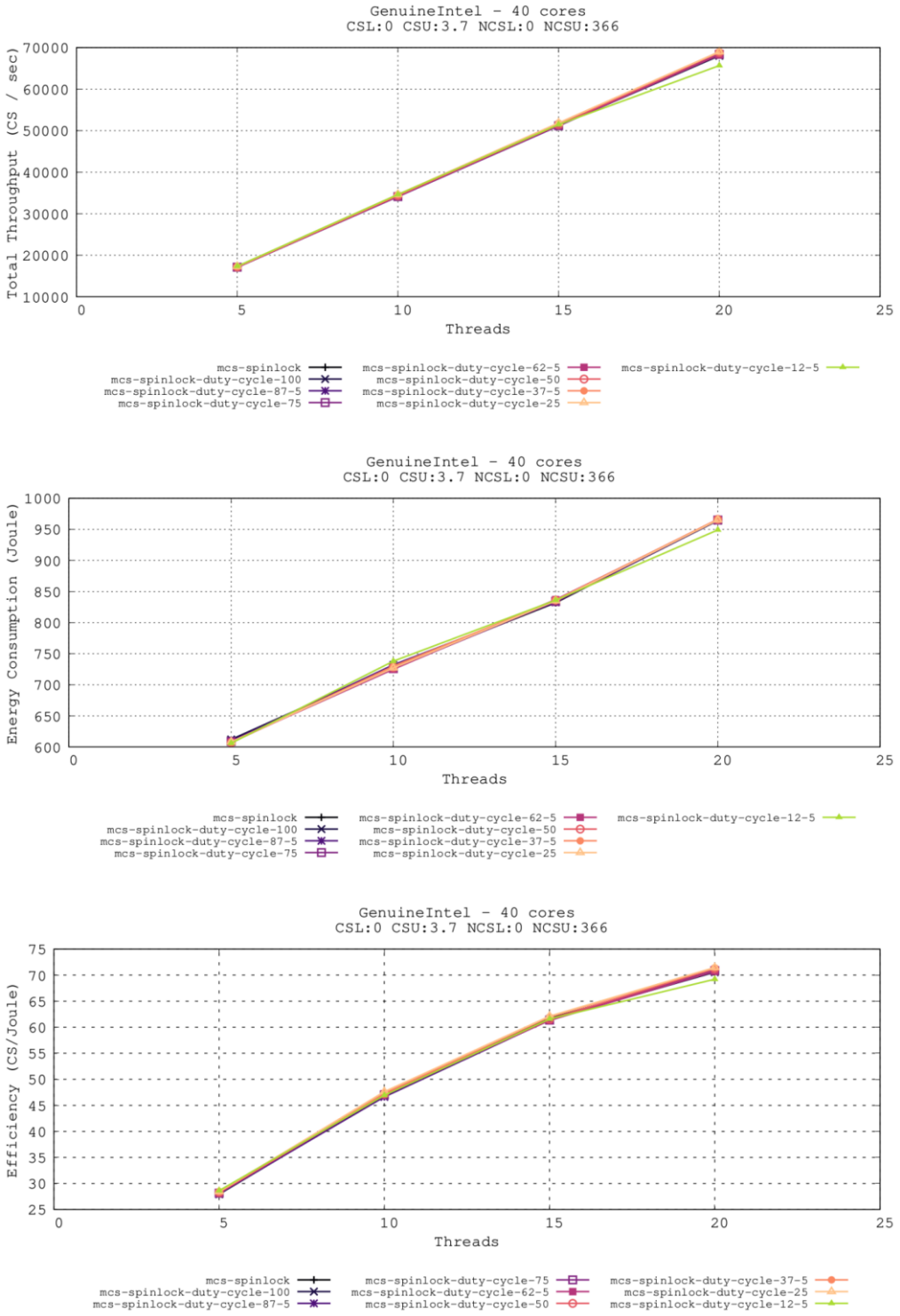

Fig 4.2 – Results for Spinlock MCS in benchmark with CS:0-3.7, NCS:0-366 for T-States.

The first benefits of our policies can be seen when we increase the length of the critical section to 366 μs. If we keep the non-critical section to 3.7 μs, then the majority of the execution time is spent during the synchronization activity, which allows for visible gains in energy consumption without damaging too much the throughput, since a lengthier critical section means less frequent write operations on the MSR as well. In terms of efficiency, we can ultimately see a ~16% increase when using all cores with the spinlock that targets 12.5% duty cycle over the default version. The relevant charts:

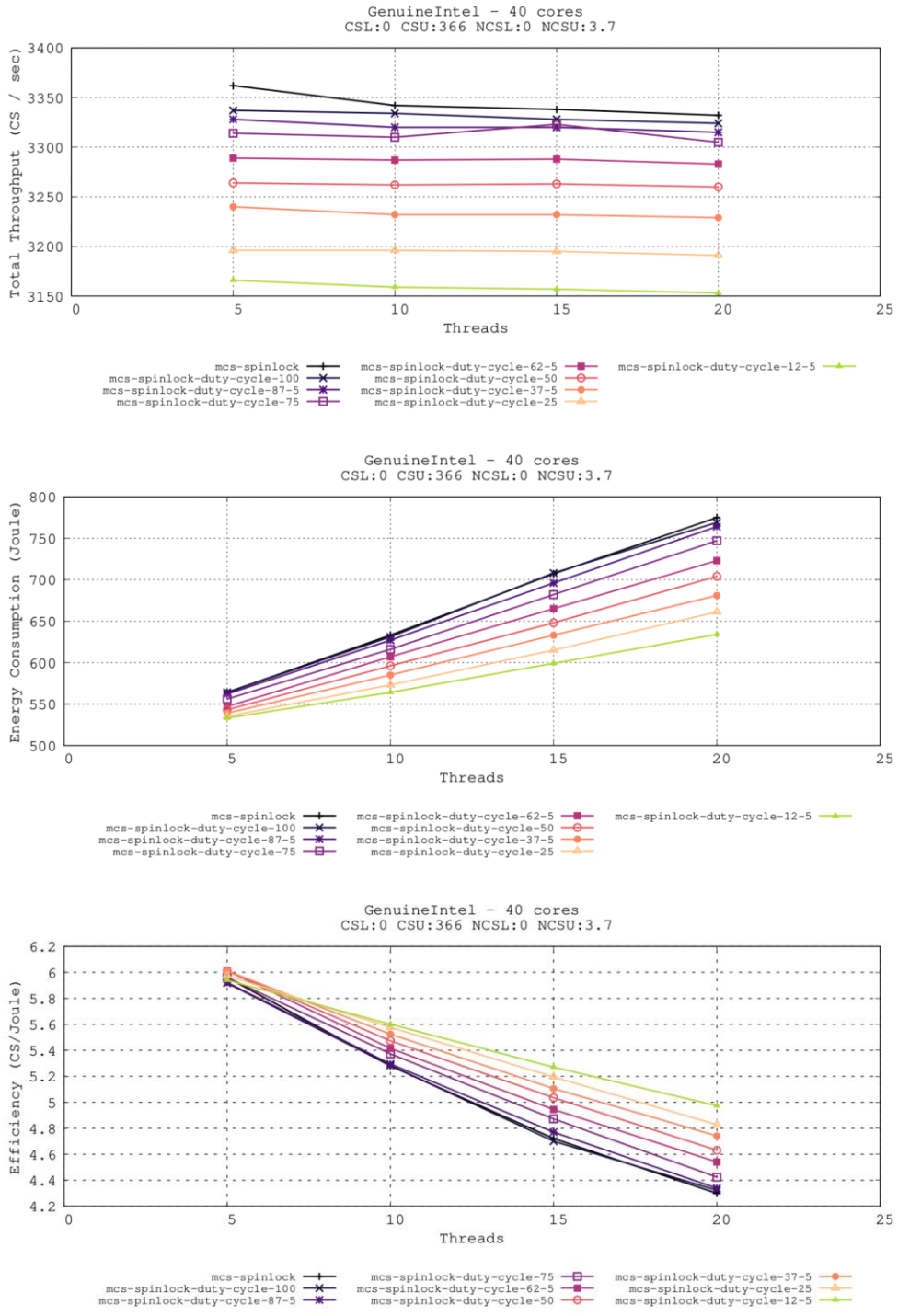

Fig 4.3 – Results for Spinlock MCS in benchmark with CS:0-366, NCS:0-3.7 for T-States.

Our policies are still efficient even when both sections are 366 µs long, but to a slightly lesser degree: since in this case we have increased the operations that a thread executes that are not synchronization-related, the system spends less time with clock modulation enabled, reducing the gains in energy consumption. Consequently, the efficiency's raise when using spinlock with a 12.5% duty cycle with all cores reduces to a ~14% over the default one, which is nonetheless still more efficient than I had expected. The charts:

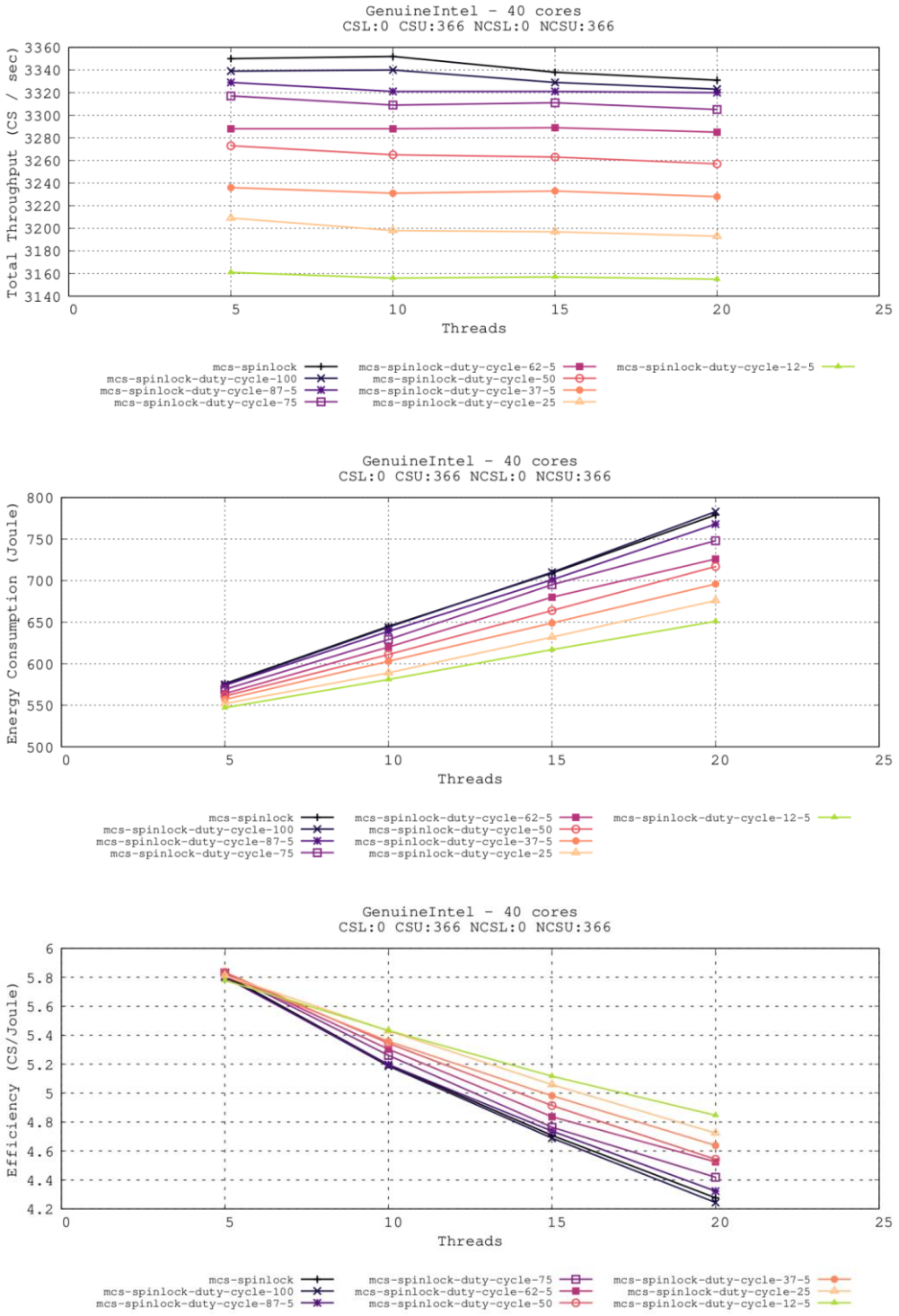

Fig 4.4 – Results for Spinlock MCS in benchmark with CS:0-366, NCS:0-366 for T-States.

Regarding the Spin-Then-Park policy, the first two benchmarks (the ones with CS:0-3.6) showed that is still better using the default version of the waiting policy for the same reasons I have described with Spinlock, so I will focus on the other two scenarios (the ones with CS:0-366) where our strategy was proven to be efficient. Since both cases were nearly identical, here are the charts for only the scenario with CS:0-366, NCS:0-3.7:

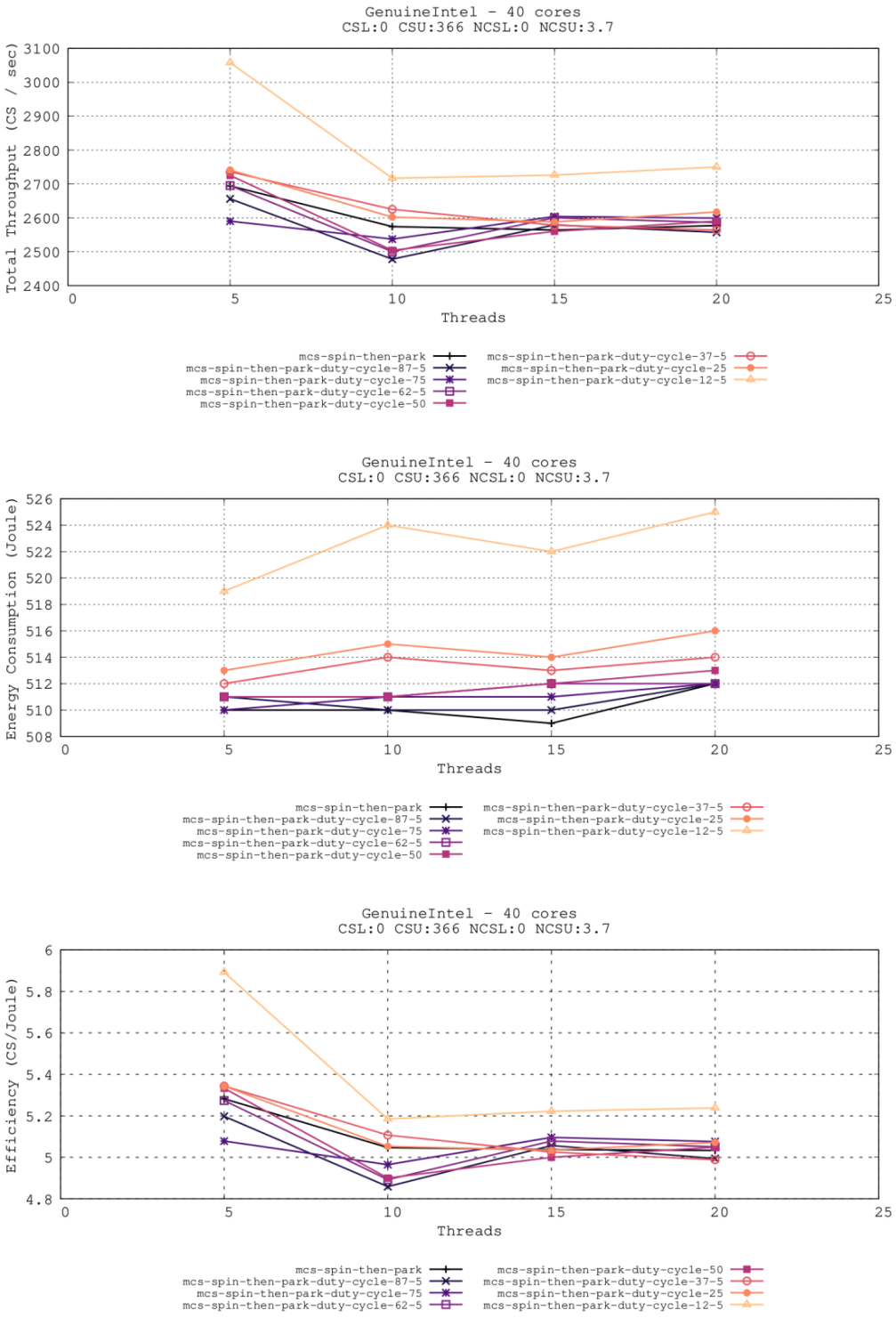

Fig 4.5 – Results for Spin-Then-Park MCS in benchmark with CS:0-366, NCS:0-3.7 for T-States.

This time, however, we can observe two unexpected behaviors, even though the best efficiency still goes to the policy with a 12.5% duty modulation: the first is that the throughput with 12.5% duty modulation visibly drops after more than 5 cores are active, the second is that the throughput and energy consumption generally increase as the duty modulation setting rises, in what is the total opposite of our previous results. My guess is that by enabling duty modulation, the counter that parks a thread when it reaches a certain threshold takes longer to reach its limit due to the slower execution times. With less instances of parking, the throughput rises because there is less latency caused by wake-up times. The downside of more spinning, however, is that the energy consumption is increased as well. In both scenarios, the most efficient case was using Spin-Then-Park at 12% duty modulation with 5 cores, a test case that likely decreases the instances of parking even more, which caused a ~%11 increase over the default waiting policy, but with the downside of having a higher energy consumption as well.

Concerning Park, no benchmark test resulted in a scenario where using a custom waiting policy brought benefits over Spin-Then-Park or explainable behaviors. Even though parking does not change any processor state by its own, the way the plots intersect each other in all the obtained graphs in what look like random patterns suggests that full-time parking is not reliable and compatible with our strategy, even though it still can produce positive impacts to efficiency at the cost of a higher energy consumption, exactly like it happens for the Spin-Then-Park policy. Here is a chart as an example:

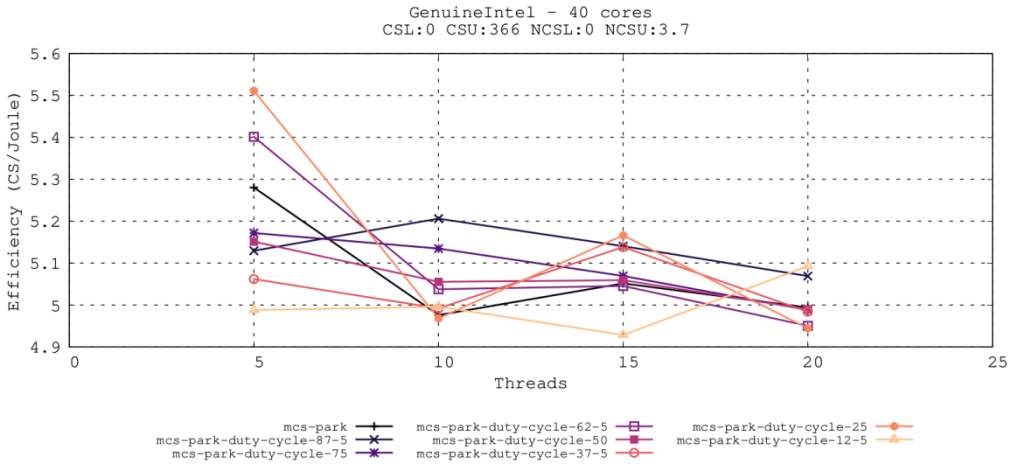

Fig 4.6 – Results for Park MCS in benchmark with CS:0-366, NCS:0-3.7 for T-States.

4.3.2 Results – P-States

Regarding Spinlock-MCS, the first benchmark scenario (the one with CS:0-3.7, NCS:0-3.7) has produced unfavorable results for our custom policies, but it was an expected situation due to our insights gained from the corresponding test with T-States. What was surprising, however, was the sudden 40% drop in performance between the default policy and the one that targets P1, which is considerably more evident than any following drop

between a pair of custom policies. Since writing on the IA32_PERF_CONTROL MSR introduces the same amount of latency as writing on the IA32_CLOCK_MODULATION MSR, the reason of such drop could be found in that series of actions that are exclusive to when setting a new P-State, such as parsing its control value. These supplementary actions could also explain why the custom policy that targets P1 consumes more energy than the default one (P0), despite their difference in throughput and despite targeting the same frequency. Here are the relevant charts:

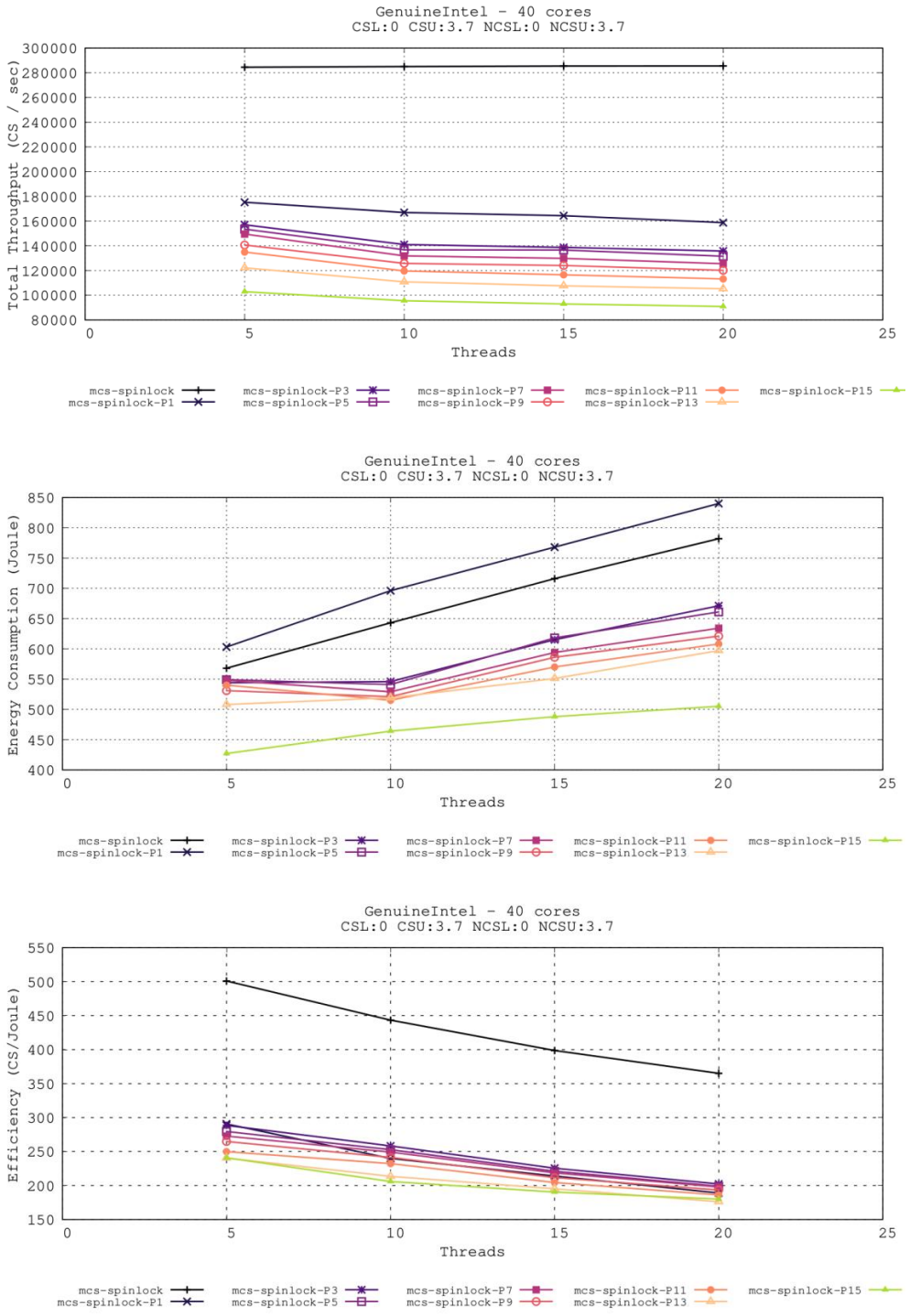

Fig 4. 7 – Results for Spinlock MCS in benchmark with CS:0-3.7, NCS:0-3.7 for P-States.

Even the 'CS:0-3.7, NCS:0-366' scenario produced surprising results, with our custom waiting policies causing higher throughputs and energy consumptions than the default case, even when they are supposed to target lower frequencies and voltages. I have no explanation for these results, other than the possibility that boost mode may have somehow engaged itself even though it was disabled prior to the test. Here are the charts:

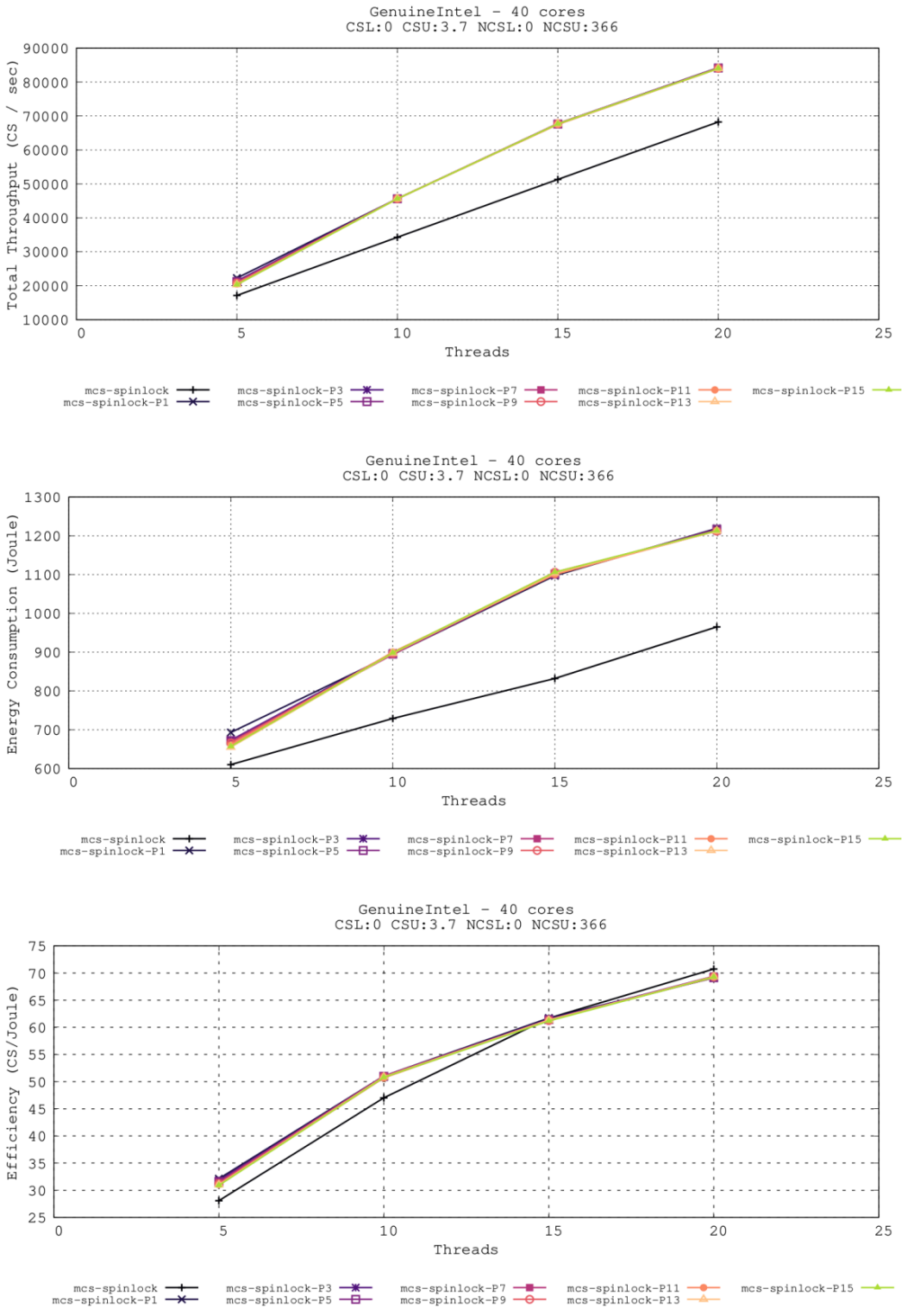

Fig 4.8 – Results for Spinlock MCS in benchmark with CS:0-3.7, NCS:0-366 with P-States.

The ‘CS:0-366, NCS:0-3.7’ scenario gave us the first notable positives results for our custom policies, which is expected since a longer critical section only makes the utilization of P-States even more effective. We can notice two key differences with the previous equivalent test with T-States: the first is that the most efficient P-State is P3, unlike with T-States where it was always the lowest duty modulation setting that gave us the best results; the second is that the most efficient case can be found when using 5 cores, with a ~13% increase over the default setting, and not when all of them are active. The charts:

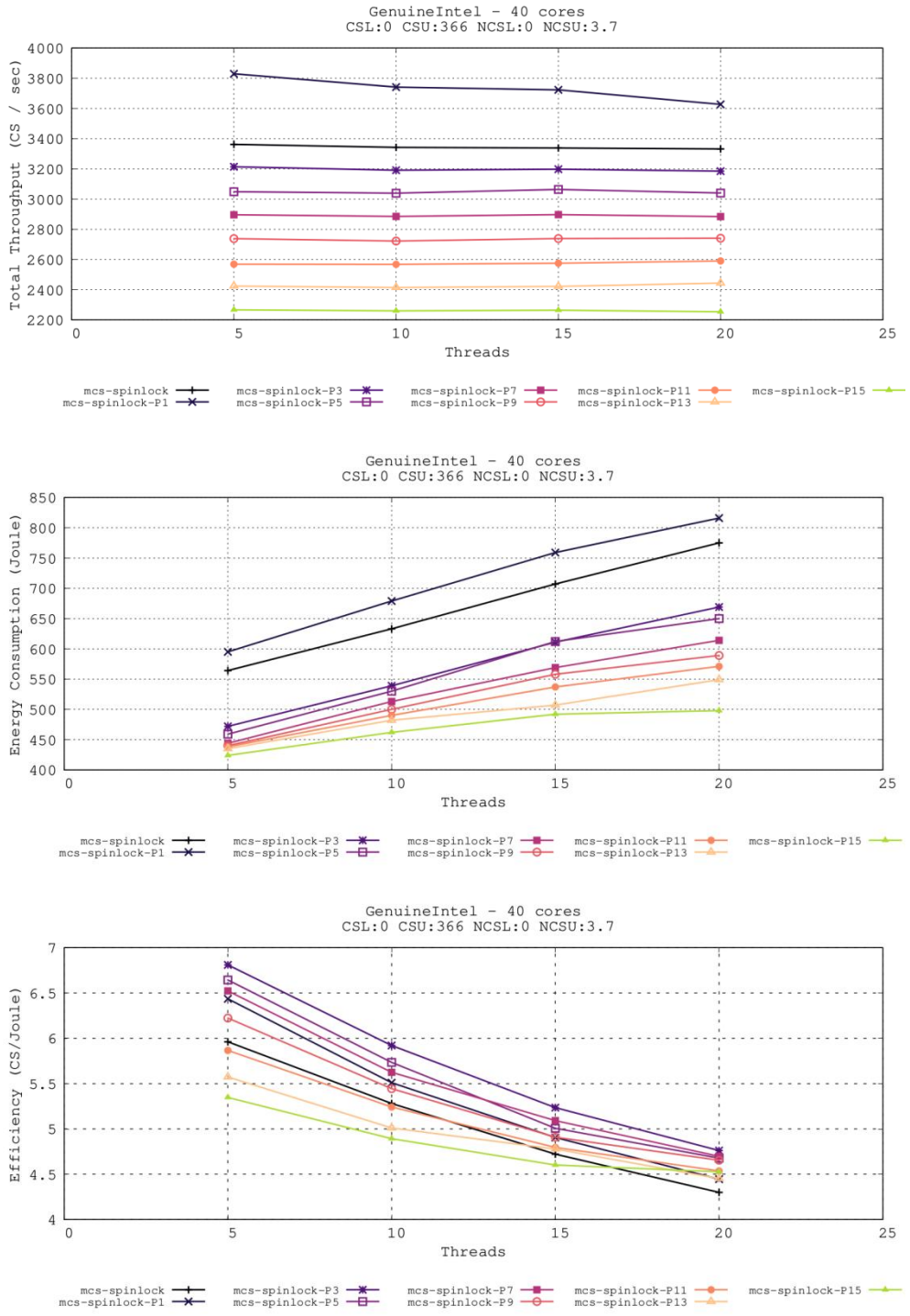

Fig 4.9 – Results for Spinlock MCS in benchmark with CS:0-366, NCS:0-3.7 with P-States.

The ‘CS:0-366, NCS:0-366’ scenario had similar results as the previous case, with the same efficiency increase between the custom waiting policy that targets P3 and the default one. However, with the lengthier non-critical section and when all cores are active, we have finally reached a test case where the drop in throughput between the default policy and the one that targets P1 is as uniform as all the following drops between consecutive pairs of policies, P13 and P15 excluded. The charts:

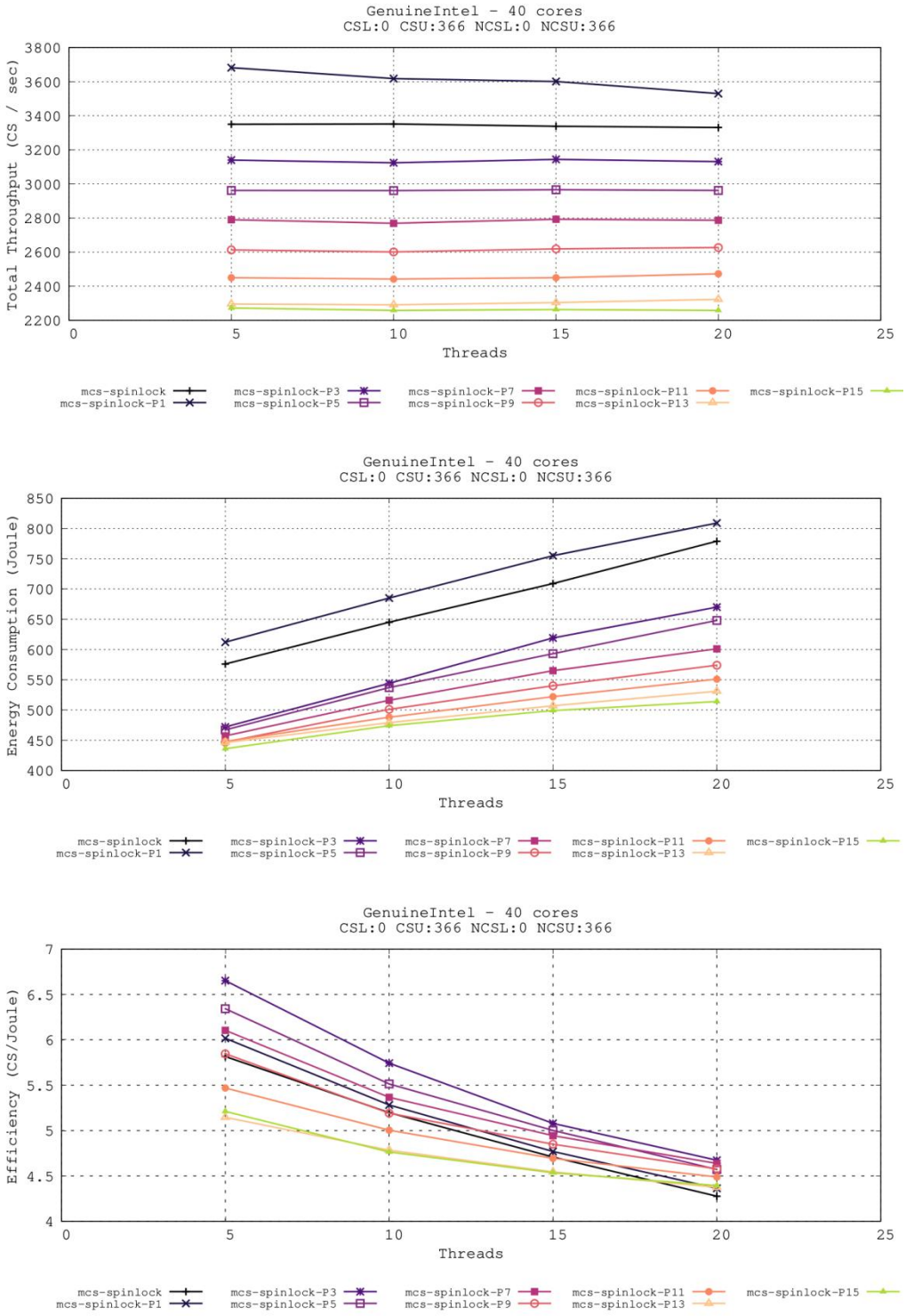

Fig 4.10 – Results for Spinlock MCS in benchmark with CS:0-366, NCS:0-366 with P-States.

Regarding Spin-Then-Park, the 'CS:0-366' scenarios unsurprisingly gave us the best results, but with the same caveats of the respective tests with T-States: that a higher efficiency also comes at the cost of a higher energy consumption, since it is likely that by lowering the voltage and frequency of a processor, we also lower the possibilities of a thread to park. What was surprising, however, is that even this time P3 is the best state to target, with a ~16% increase over the default policy with 5 cores. For brevity, I will only show the charts of the 'CS:0-366, NCS:0-366' scenario:

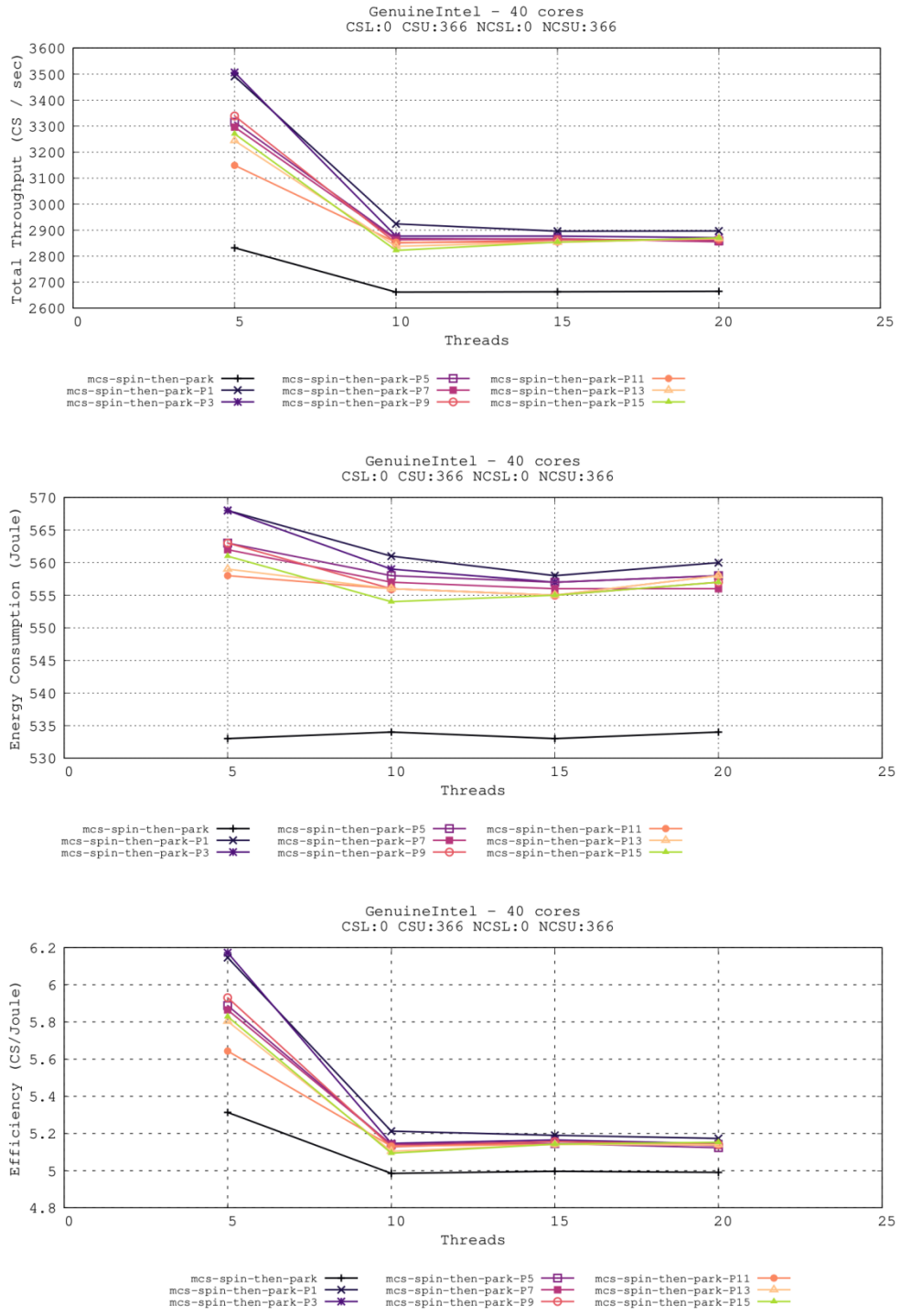

Fig 4.11 – Results for Spin-Then-Park MCS in benchmark with CS:0-366, NCS:0-366 with P-States.

Concerning Park, that policy proved to be unpredictable as it did with T-States, with all the plots related to our custom policies intersecting each other as if the chosen P-State has no relevance at all on the behavior of the program, despite producing occasional positive results that are not worth pursuing over Spinlock and Spin-Then-Park. Here is an example:

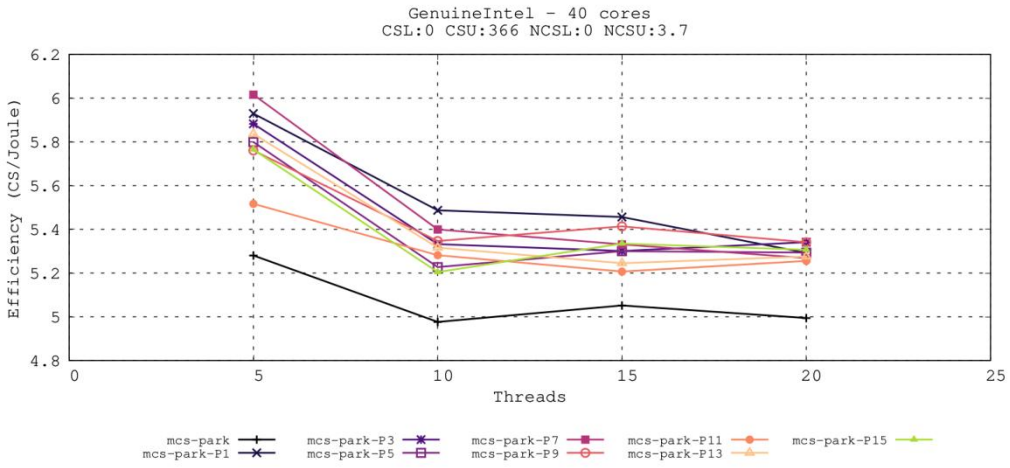

Fig 4.12 – Results for Park MCS in benchmark with CS:0-366, NCS:0-3.7 for P-States.

3.4.3 Results – C-States

Regarding the utilization of C-States through the waiting policy based on the MONITOR/MWAIT pair of operations, our first surprise came when all our results reported no throughput at all, as if no thread was able to enter the critical section due to the fact that all of them were sleeping, which is impossible since one of them had to acquire the lock by design and, consequently, remain awake.

Sadly, this was an unfortunate issue because these custom policies work correctly on my own working laptop without any code changes, and I could not find the root issue on the testing machine due to my very limited time I could spend with it, especially when CPUID reports that it is supposed to support MWAIT, although with a limited degree since only C1 and C3 could be targeted.

This is why all the following charts will represent tests that I had to execute on my own laptop, which has a Intel Core i7-4510U CPU @ 2.00 GHz with two physical cores. Since the presence of logical cores do not introduce complications when targeting C-States, I could safely enable hyper-threading so I could double the available cores.

The bright side is that these custom policies seem to bring a positive impact already in the first benchmark scenario, the 'CS:0-3.6, NCS:0-3.6' one, when all cores are active: even if there is a 44-47% drop in throughput when using the custom waiting policies, the drop in energy consumption is slightly more important when targeting C1 and C2, which makes them more efficient than the default spinlock by 6-8%. Since the tests with T-States and P-States with this scenario were not successful on my machine as well, I remain

hopeful that these positive results would have carried over on more powerful hardware, assuming there would not be noticeable differences in latencies during the states' transitions. The charts:

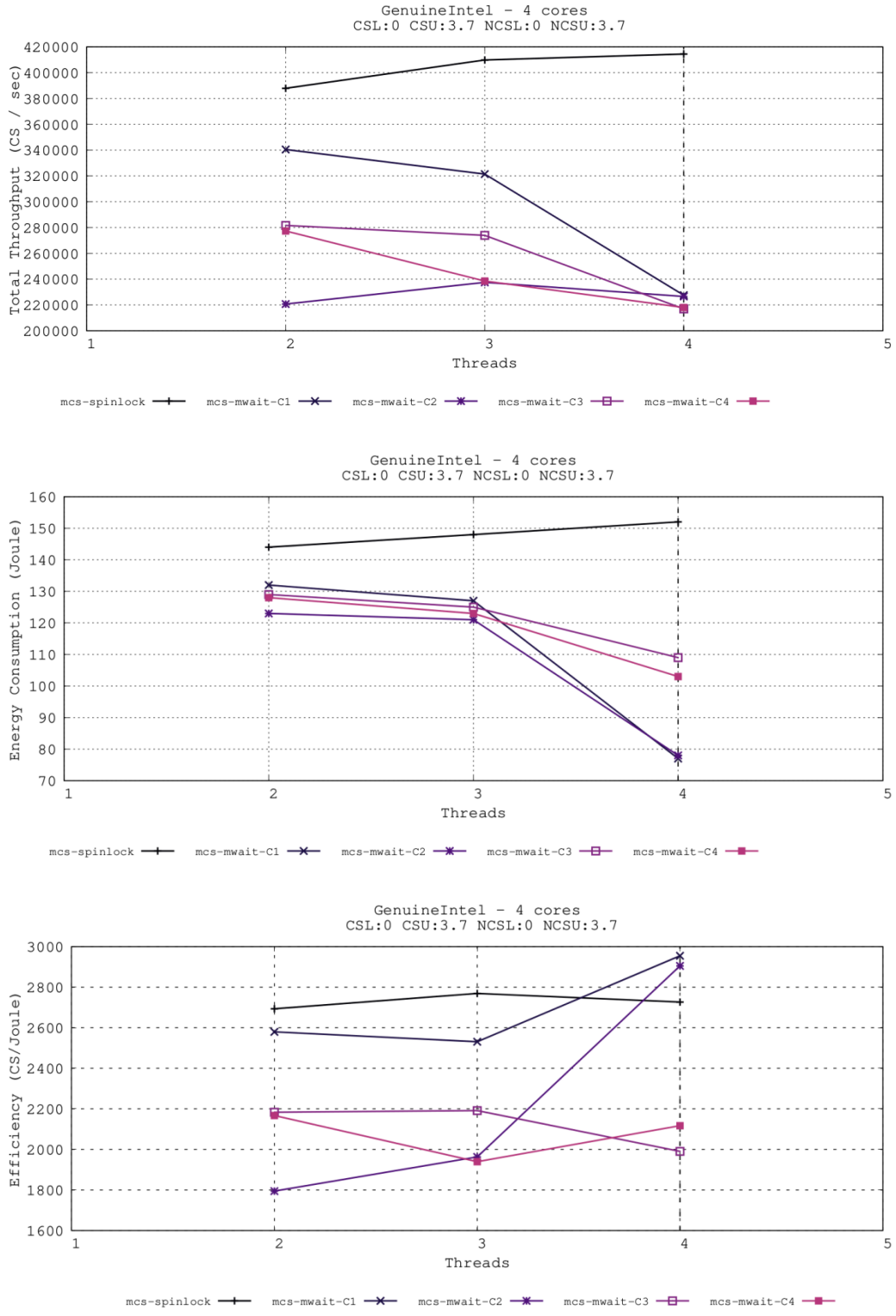

Fig 4.13 – Results for Spinlock MCS and MWAIT policies in benchmark with CS:0-3.7, NCS:0-3.7.

The ‘CS:0-3.6, NCS:0-366’ scenario brought instead expected results: exactly like it happened with T-States, the critical section is too small compared to the non-critical one, which nullifies any eventual effectiveness of the custom policies. The charts:

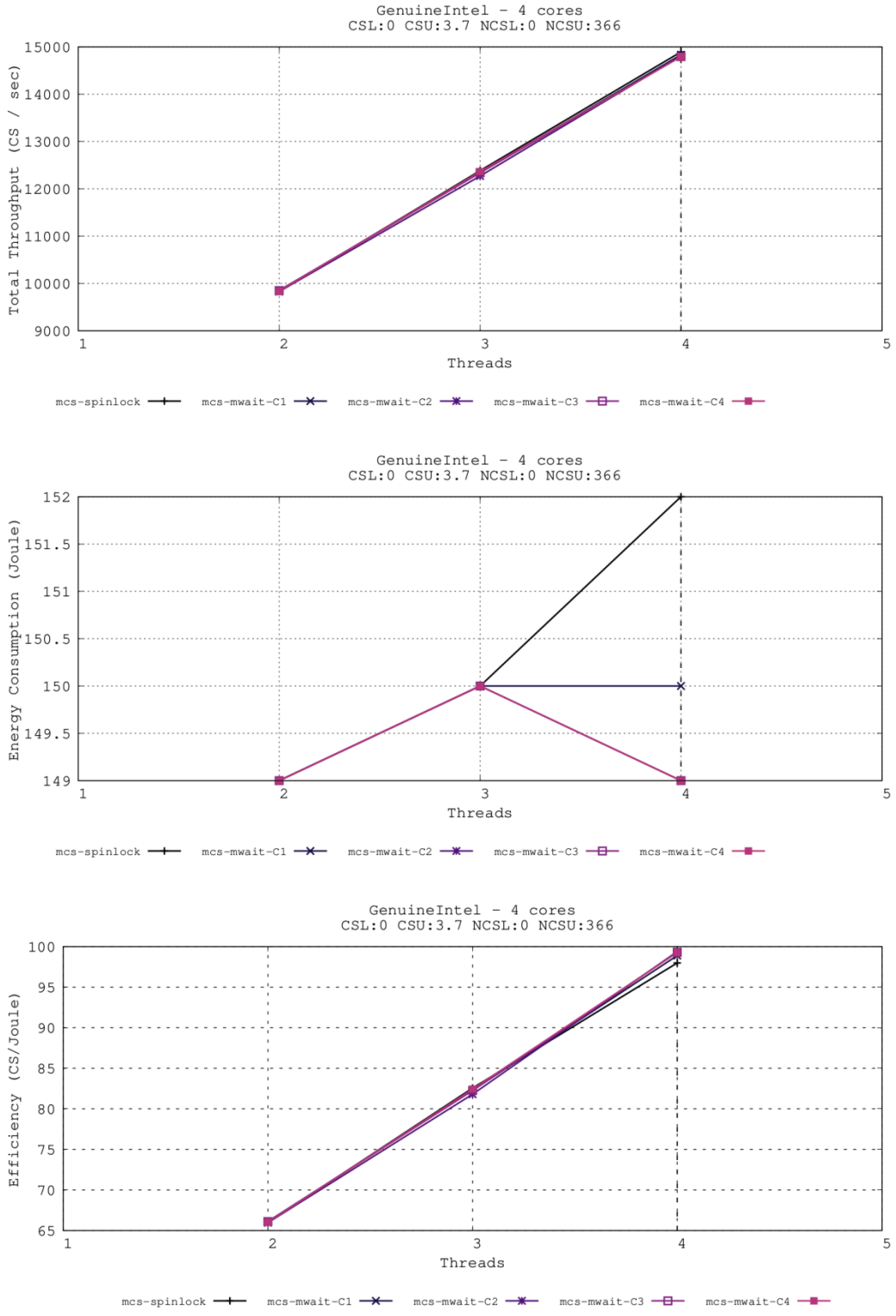

Fig 4.14 – Results for Spinlock MCS and MWAIT policies in benchmark with CS:0-3.7, NCS:0-366.

The ‘CS:0-366, NCS:0-3.6’ scenario produced what I consider to be impressive results, with a ~50% efficiency increase when targeting C2, C3 and C4 with all cores active: this is the biggest increase recorded so far. The charts:

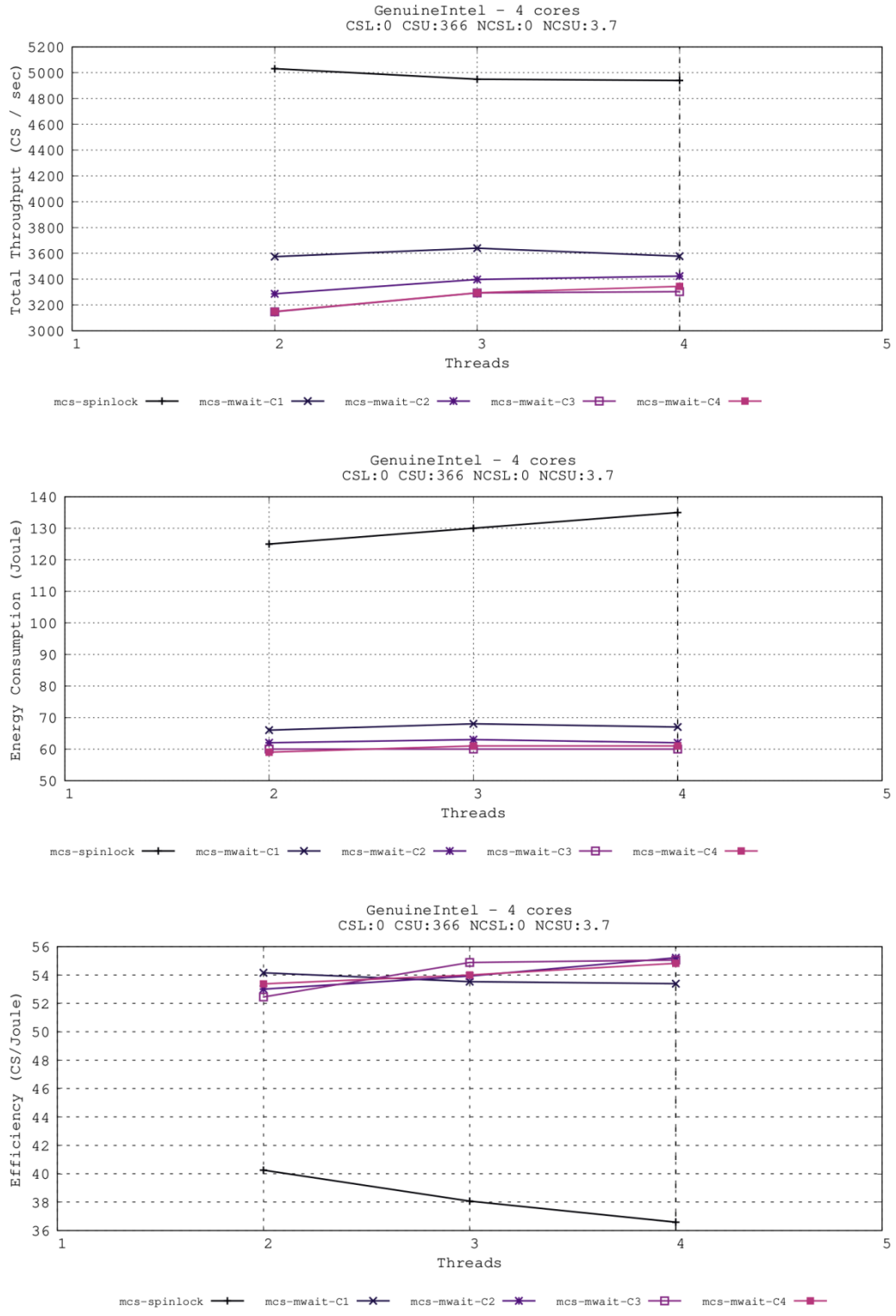

Fig 4.15 – Results for Spinlock MCS and MWAIT policies in benchmark with CS:0-366, NCS:0-3.7.

Finally, the 'CS:0-366, NCS:0-366' brought positive results as well, but expectedly to a lesser degree in respect to the previous scenario due to the lengthier non-critical section: the peak increase with the custom policies when all cores are active is now reduced to a ~15% over the default one. The charts:

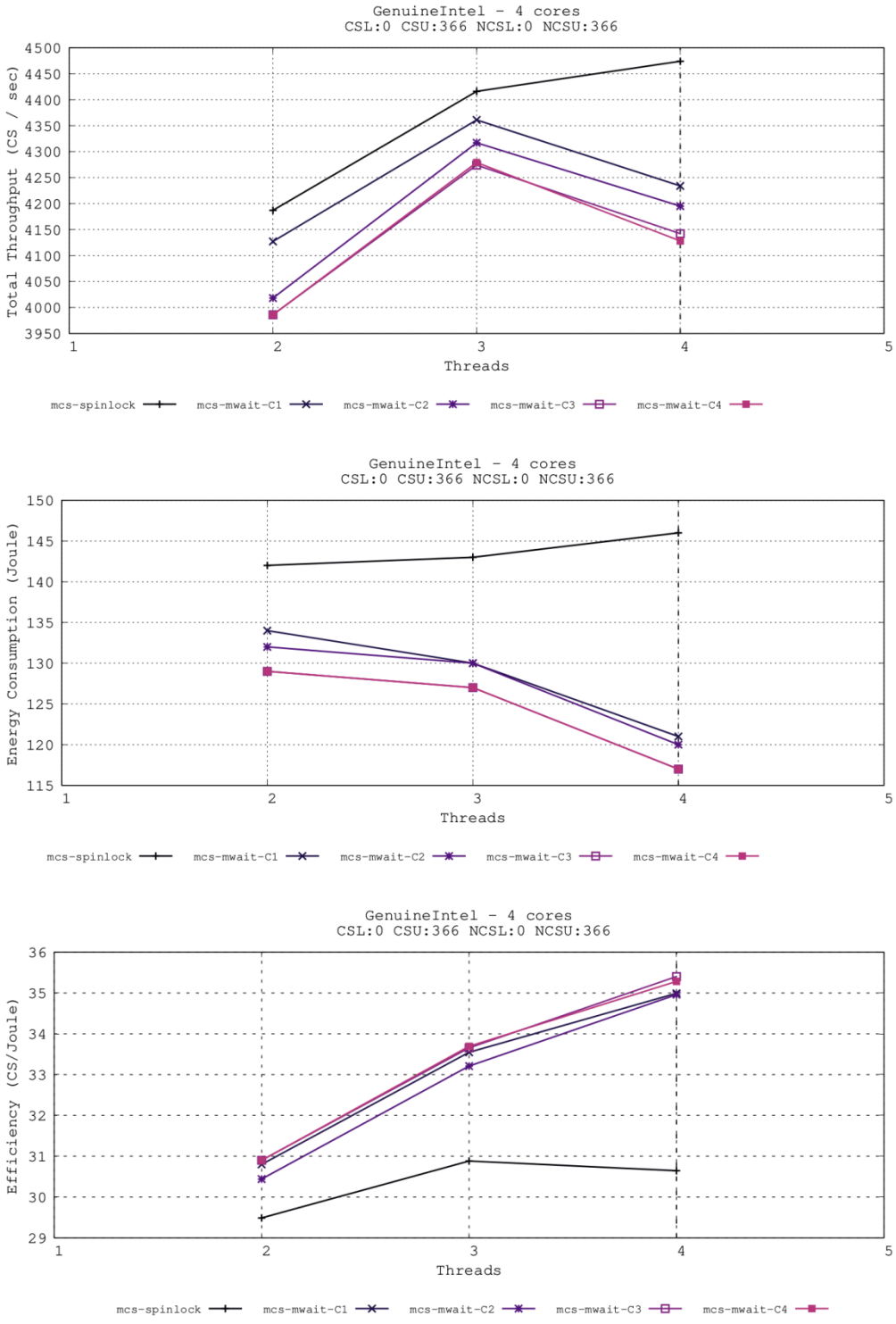

Fig 4.16 – Results for Spinlock MCS and MWAIT policies in benchmark with CS:0-366, NCS:0-366.

3.4.4 Discussion

I came away from these results feeling both relieved and disappointed. Relieved because I could identify scenarios where our custom waiting policies brought tangible benefits to the efficiency with all types of hardware states, especially T and C-States, meaning that changing these states during spinlock-based synchronization is a strategy that is worth looking into for further expansions and experimenting. Disappointed because I could not ultimately find explanations to some of the most unusual behaviors with some scenarios with P-States and C-States, suggesting that some pieces of the puzzle are still missing somewhere despite the promising start.

Regarding possible areas of interest where this work could be expanded, I think modifying Powerctl to account for logical threads would be the first natural step. By modifying the hashtable and probing system to request new states for pairs of logical cores mapped to the same physical one only when they target the same P or T-States would allow further tests where hyper-threading can be safely enabled without the risk of introducing complications.

And speaking of further tests, I definitely think that is worth applying these custom waiting policies to other spinlock algorithms besides MCS, so to find new possible benefits. And since we have proven that the sizes of the critical and non-critical sections have a noticeable impact on the efficiency, I also think that creating further benchmarks scenarios with new lengths for these sections will certainly help us find that set of values that bring the best out of our custom policies, especially since there is quite a gap between the pair of values we have used for all the tests (3.6 and 366 μs).

5. Conclusion

In this thesis I have investigated how changing processors' hardware states during spinlock-based synchronization can alter the throughput and energy consumption of a multi-threaded application, with the intention of improving the efficiency of its computation.

After a deep dive on the key concepts behind the hardware states and how they can be manipulated via model-specific registers and privileged instructions, and after an introduction to the most well known spinlock algorithms, I illustrated all the technical details that led to the creation of a Linux kernel module designed to aid us in our investigation.

Called Powerctl, we showed how this module allows users to target new hardware states on the fly thanks to a suite of system calls that ensure quicker latencies in respect to other solutions that are based on the utilization of the pseudo-file system. These quick latencies made possible for us to investigate the potential usefulness of custom waiting policies for the MCS algorithm that allow a thread to request new hardware states via Powerctl when it fails to acquire the lock, so that it does not keep spinning at full power, and to restore the system to its full capacity when it ultimately acquires it.

After executing benchmark tests that applied our custom waiting policies in a variety of scenarios that differ in the size of the critical and non-critical sections and the number of active cores, we could find cases where we could improve the efficiency of the computation by targeting specific hardware states during synchronization. These first positive results suggest that this initial investigation is worth enhancing with additional improvements to Powerctl, so that it can support more features and ways to trigger these states, and with further testing, so to possibly localize more scenarios where our custom policies prove to be even more efficient.

However, these tests also showed some unusual behaviors that could not be easily explained via the notions I acquired during my studies, suggesting that the world of processors' hardware states can be even more complex than what it seemed at first glance.

Finally, once Powerctl has been enhanced with positive improvements that can be verified through further testing phases, our last objective is to apply it to a robust benchmark suite for parallel applications such as Splash-3 [68], so that we can predict how its benefits would carry over to real applications.

References

[1] «Cisco Annual Internet Report (2018-2023),» [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[2] «Data Centres and Data Transmission Networks,» [Online]. Available: https://www.iea.org/reports/data-centres-and-data-transmission-networks.

[3] B. Whitehead, D. Andrews, A. Shah e G. Maidment, «Assessing the environmental impact of data centres part 1: Background, energy use and metrics,» *Building and Environment,* vol. 82, pp. 151-159, 2014.

[4] L. Castellazzi, A. Maria e P. Bertoldi, «Trends in data centre energy consumption under the European Code of Conduct for data centre energy efficiency.,» 2017.

[5] «The Impressive Stats Behind Amazon's Dominance of the Cloud on VisualCapitalist,» [Online]. Available: https://www.visualcapitalist.com/stats-amazon-dominance-cloud/.

[6] H. Rong, H. Zhang, S. Xiao, C. Li e C. Hu, «Optimizing energy consumption for data centers,» *Renewable and Sustainable Energy Reviews,* vol. 58, pp. 674-691, 2016.

[7] «Advanced Configuration and Power Interface (ACPI) Specification,» [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf.

[8] L. Benini, A. Bogliolo e G. Micheli, «A survey of design techniques for system-level dynamic power management,» *IEEE Trans. Very Large Scale Integration (VLSI) Syst.,* vol. 8, 2000.

[9] «ACPI Overview,» [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_Overview.pdf.

[10] L. Brown, «The State of ACPI in the Linux Kernel,» *Proceedings of the Linux Symposium,* vol. 1.

[11] «Advanced Configuration and Power Interface,» [Online]. Available: https://www.sciencedirect.com/topics/computer-science/advanced-configuration-and-power-interface.

[12] N. H. E. Weste e D. Money Harris, CMOS VLSI Design: A Circuits and Systems Perspective, Pearson, 2011.

[13] R. Schöne, T. Ilsche, M. Bielert, D. Molka e D. Hackenberg, «Software Controlled Clock Modulation for Energy Efficiency Optimization on Intel Processors,» 2016.

[14] «Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol. 3B,» [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[15] «P6 Family of Processors,» [Online]. Available: https://download.intel.com/design/PentiumII/manuals/24400101.pdf.

[16] «Intel® Pentium® 4 Processor on 90 nm Process,» [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/pentium4/sb/30056103.pdf.

[17] «Intel® Pentium® M Processor,» [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/mobile/pm/sb/25261203.pdf.

[18] «C-States, P-States, Where the Heck are Those T-States?,» [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/c-states-p-states-where-the-heck-are-those-t-states.html.

[19] «What Is Hyper-Threading?,» [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html.

[20] «Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol. 3A,» [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html.

[21] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart e R. Geyer, «An Energy Efficiency Feature Survey of the Intel Haswell Processor,» 2015.

[22] R. Schöne, T. Ilsche, M. Bielert, A. Gocht e D. Hackenberg, «Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance,» 2019.

[23] «Power Management of Intel Architecture Servers,» [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/power_management_of_intel_architecture_servers.pdf.

[24] «Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors,» [Online]. Available: https://www.dragonflybsd.org/~sephe/320354.pdf.

[25] V. Pallipadi e A. Starikovsky, «The ondemand governor: past, present and future,» *Proceedings of Linux Symposium,* vol. 2, pp. 223-238, 2006.

[26] «CPU frequency scaling on ArchLinux,» [Online]. Available: https://wiki.archlinux.org/title/CPU_frequency_scaling.

[27] «CPU governor performance on Percona,» [Online]. Available: https://www.percona.com/blog/2016/05/06/cpu-governor-performance/.

[28] R. Hebbar e A. Milenkovic, «An Experimental Evaluation of Workload Driven DVFS,» 2021.

[29] «Cpu freq and num of cores on Variwiki,» [Online]. Available: https://variwiki.com/index.php?title=CPU_freq_and_num_of_cores.

[30] «Controlling Processor C-State Usage in Linux,» [Online]. Available: https://docplayer.net/33712101-A-dell-technical-white-paper-describing-the-use-of-c-states-with-linux-operating-systems-stuart-hayes-enterprise-linux-engineering.html.

[31] «Difference Between Deep and Deeper Sleep States for Processors,» [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000006619/processors/intel-core-processors.html.

[32] «Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol. 2B,» [Online]. Available: https://www.intel.it/content/www/it/it/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2b-manual.html.

[33] J. M. Mellor-Crummey e M. L. Scott, «Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,» 1991.

[34] T. E. Anderson, «The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors,» *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, n. 1, 1990.

[35] L. Rudolph e Z. Segall, «Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,» *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture,* pp. 340-347, 1984.

[36] G. Graunke e S. Thakkar, «Synchronization algorithms for shared-memory multiprocessors,» *Computer,* vol. 23, n. 6, pp. 60-69, 1990.

[37] T. S. Craig, «Building FIFO and priority-queueing spin locks from atomic swap,» 1993.

[38] P. Magnusson, A. Landin e E. Hagersten, «Queue locks on cache coherent multiprocessors,» *Proceedings of 8th International Parallel Processing Symposium,* pp. 165-171, 1994.

[39] Franke, T. J. Watson e M. Kirkwood, «Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux,» 2005.

[40] Y. Woo, S. Kim, C. Kim e E. Seo, «Catnap: A Backoff Scheme for Kernel Spinlocks in Many-Core Systems,» *IEEE Access,* vol. 8, pp. 29842-29856, 2020.

[41] P. Jay Salzman, M. Burian e O. Pomerantz, The Linux Kernel Module Programming Guide, 2007.

[42] I. Zakharov, V. Mutilin e A. Khoroshilov, «Pattern-based environment modeling for static verification of Linux kernel modules,» *Lecture Notes in Computer Science,* vol. 41, pp. 183-195, 2015.

[43] «'ioctl' on the Linux Programmer's Manual,» [Online]. Available: https://man7.org/linux/man-

pages/man2/ioctl.2.html.

[44] «The GNU C Library (glibc),» [Online]. Available: https://www.gnu.org/software/libc/.

[45] «Kernel System Calls on Linux.it,» [Online]. Available: https://www.linux.it/~rubini/docs/ksys/.

[46] «'sys_ni_syscall' on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/syscalls.h#L1273.

[47] «Linux Kernel System Call Table on Microsoft Docs,» [Online]. Available: https://docs.microsoft.com/en-us/security/research/project-freta/report/kernel-system-call-table.

[48] «__SYSCALL_DEFINEx on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/v3.14.10/source/include/linux/syscalls.h#L187.

[49] «ASLR on Oracle® Linux 6 Security Guide,» [Online]. Available: https://docs.oracle.com/en/operating-systems/oracle-linux/6/security/ol_aslr_sec.html.

[50] «System.map on KernelNewbies,» [Online]. Available: https://kernelnewbies.org/FAQ/System.map.

[51] «proc_create on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/v4.14/source/fs/proc/generic.c#L504.

[52] «Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol. 2C,» [Online]. Available: https://www.intel.it/content/www/it/it/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2c-manual.html..

[53] «'ams' on GCC,» [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[54] « The AMD64 Architecture Programmer's Manual, Volume 1: Application Programming,» [Online]. Available: https://developer.amd.com/wordpress/media/2012/10/24592_APM_v11.pdf.

[55] «acpidump on FreeBSD System Manager's Manual,» [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=acpidump&sektion=8&format=html.

[56] «acpidump on Ubuntu Manpage,» [Online]. Available: https://manpages.ubuntu.com/manpages/precise/en/man1/acpidump.1.html.

[57] «'linux/wait.h' on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/wait.h.

[58] «What is RCU, Fundamentally?,» [Online]. Available: https://lwn.net/Articles/262464/.

[59] «"linux/hashtable.h" on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/hashtable.h.

[60] «Kernel Probes on The Linux Kernel Archives,» [Online]. Available:

https://www.kernel.org/doc/Documentation/kprobes.txt.

[61] «'finish_task_switch' on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L4772.

[62] «'linux/kprobes.h' on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/kprobes.h.

[63] «'task_struct' on bootlin,» [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L723.

[64] «'proc' on The Linux Programming Interface,» [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html.

[65] H. Guiroux e R. Lachaize, «Multicore Locks: the Case is Not Closed Yet».

[66] «Lockbench on Github,» [Online]. Available: https://github.com/HPDCS/lockbench.

[67] «'Power and performance tuning' on Microsoft Docs,» [Online]. Available: https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/hardware/power/power-performance-tuning.

[68] C. Sakalis, C. Leonardsson, S. Kaxiras e A. Ros, «Splash-3: A properly synchronized benchmark suite for contemporary research,» *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS),* pp. 101-111, 2016.