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

Graduating
Luca Sannino 1542194

Advisor
Christian Napoli

Co-Advisor
Romolo Marotta

SAPIENZA
UNIVERSITÀ DI ROMA

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

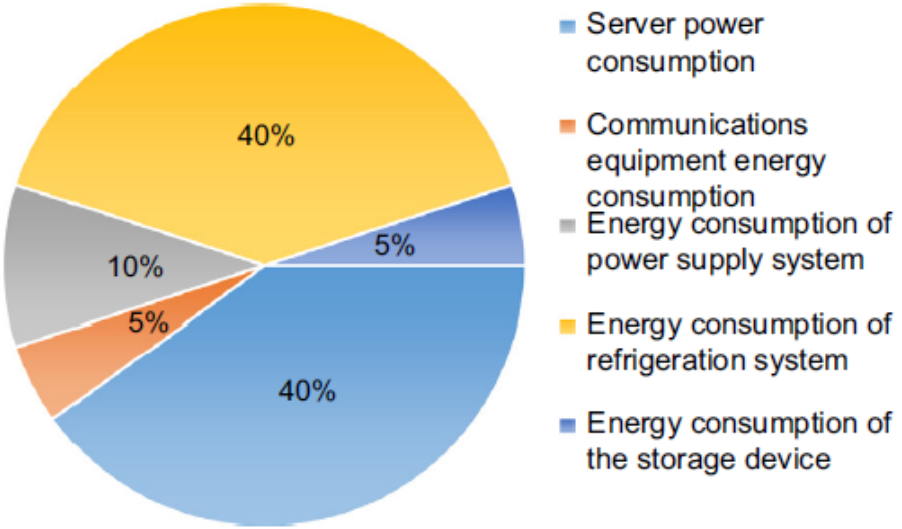
A/A 2020/2021

The Rise of the ICT Industry…

- ICT: Information and Communication Technology.
  Term used to define all the technology that deals with sending, storing and retrieving digital information.
- The worldwide growth of internet users 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.

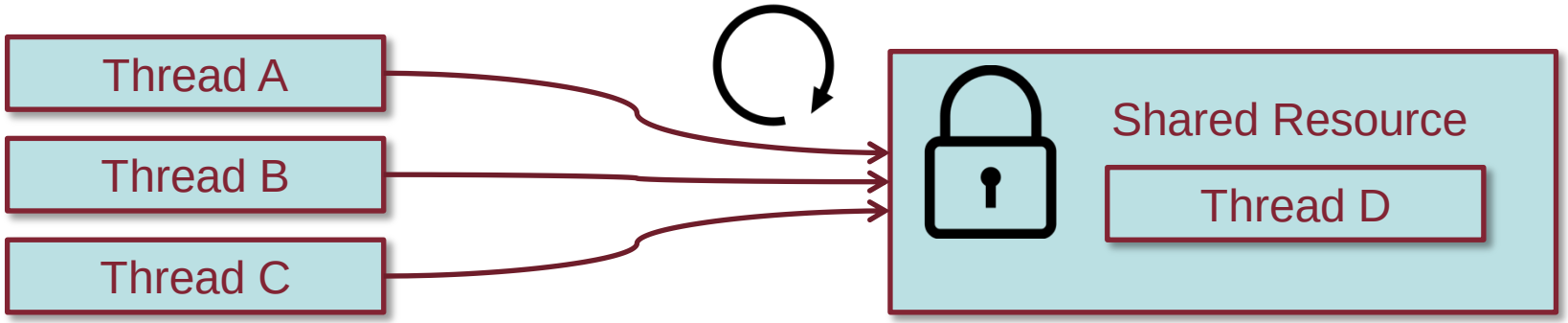
… And Why It Is a Problem for Us!

- These centers need to store huge amounts of data, to work 24/7 and to resist to damage by supporting redundancy.
- They have to consume high amounts of energy in order to power and cool themselves: as of 2017, it is estimated that ITC accounts for 2% of worldwide CO^2 emissions!
- Since servers' computation accounts for about 40% of the total energy consumption, it is critical for High Performance Computing to find new ways to optimize the computation and reduce energy consumption.



Identifying a Possible Area for Optimization

- These centers generally contain multi-core architectures that support parallel computing.
- This enables 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.
- 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.



Optimizing Synchronization… Is It Possible?

- 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:

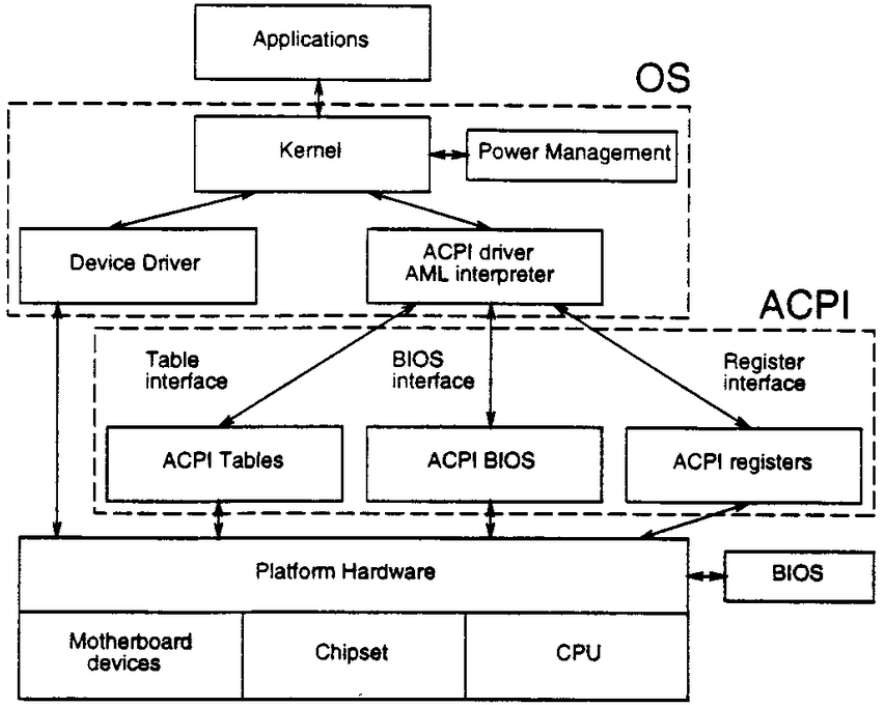
> Is it possible to save up on energy consumption by targeting specific hardware states during spinlock-based synchronization without downgrading the performance to a degree that the end user would consider unacceptable?

- NOTE: For this investigation we have targeted the x86 ISA with a focus on Intel processors.

So… What Are These CPU Hardware States?

- Part of a standard called Advanced Configuration and Power Interface, which acts as an abstract interface that links together hardware and software so that systems can perform power management.
- What types of hardware states are available?

  - Throttling States, which directly affect processors' clock modulation.
  - Performance States, which alter processors' voltage and frequency.
  - CPU Power States, which are able to put to sleep various processors' components.

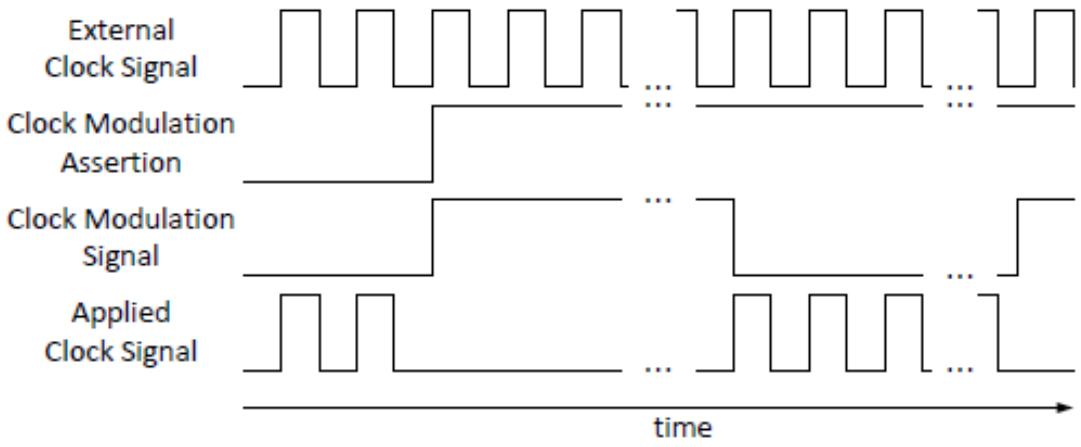


Why Do These Hardware States Even Exist?

- To reduce the power consumption of a processor!
- 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.
- 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.

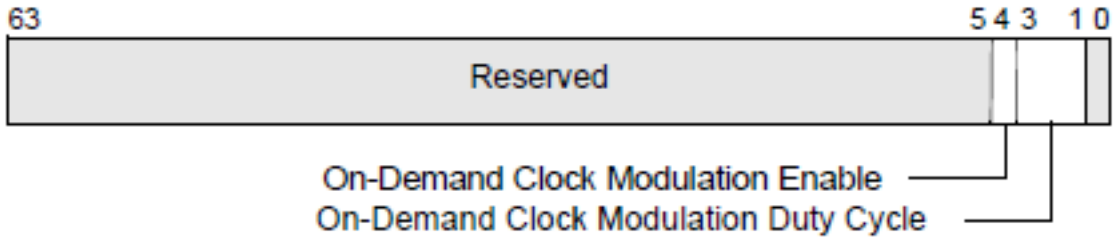
Throttling States (T-States)

- They allow to change a processor's clock modulation.
- 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.



Software-Controlled T-States

- Software can trigger T-States through a model-specific register called IA32_CLOCK_MODULATION.



  - On-Demand Clock Modulation Enable: If set to 1, enables clock modulation.
  - On-Demand Clock Modulation Duty Cycle: It accepts a 3-bit control value that allows to trigger a T-State. Software can only alter duty modulation by 12.5% intervals.

T-States – Advantages and Disadvantages

+ Encodings used to set T-States are architecture-independent, making optimizations that rely on them more portable by default.

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%

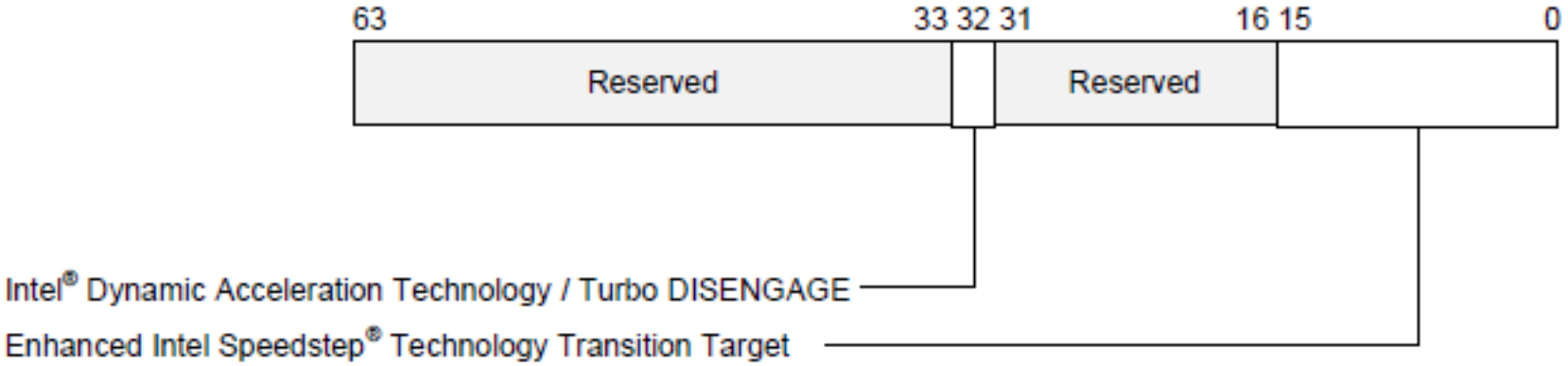
+ They can be enabled with little to no latency: in some processors, the worst case scenario is a 43.5 µs delay.

- They only affect the power consumption to a certain degree, since voltage and frequency are unchanged.

- Performance is slightly more downgraded than what is expected from the percentage belonging to a state.

Performance States (P-States)

- They allow to alter the voltage and frequency of a processor.
- Software can trigger P-States through a model-specific register called IA32_PERF_CTL.

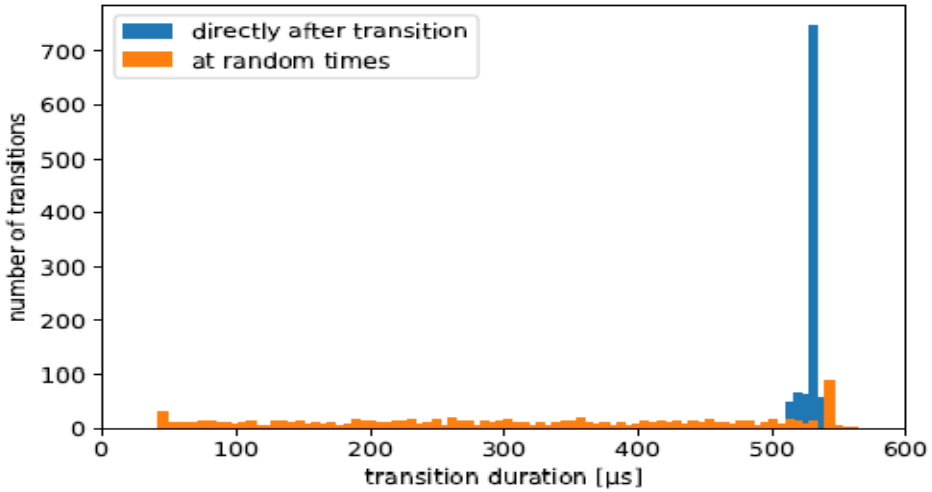


- Enhanced Intel Speedstep Technology Transition Target: It accepts a 16-bit control value that allows to target a P-State.
- Intel Dynamic Acceleration Technology / Turbo DISENGAGE: It disables boost mode, which allows processor to target higher performances if there is enough thermal headroom available.

P-States – Advantages and Disadvantages

+ By altering both voltage and frequency, they can lead to more impactful optimizations than T-States.

- 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.



- The control values used to define the various performance configurations are different for each processor's architecture.

CPU Power States (C-States)

- They allow the processor to put some of its components to sleep.
- There are various C-States, and what each of them precisely does may be different relative to the processor's architecture, but generally:
  - C0: Active. Every component of the CPU is fully active.
  - C1: Halt. The processor halts its execution and disables its core clock.
  - 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.

MONITOR and MWAIT

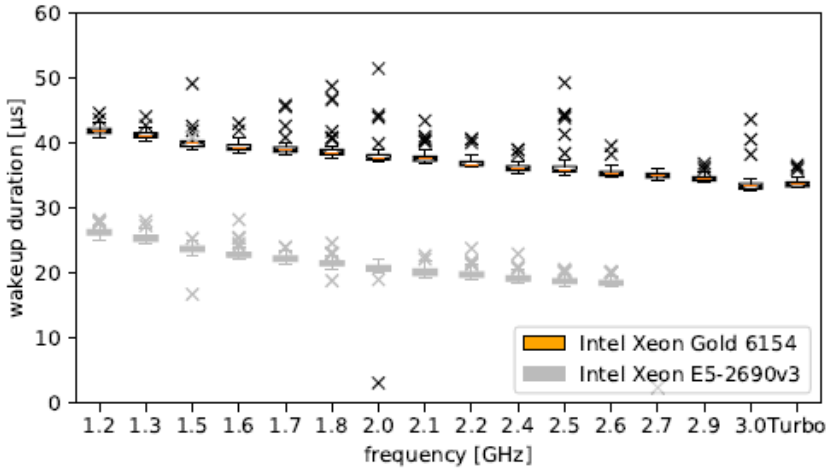
- Software can manually target a C-State by taking advantage of a pair of privileged x86 instructions, MONITOR and MWAIT.

  - MONITOR: This instruction triggers the monitoring of an address that has been specified as input. If a store in that address range is detected, MONITOR is notified and wakes up any thread that is waiting with MWAIT.
  - MWAIT: If paired with MONITOR, any thread that encounters this instruction will put the processor in a input-selected T-State until one of the three following events happen: MONITOR is triggered by a store in the monitored range address, or an external signal or a machine exception is detected.

C-States – Advantages and Disadvantages

+ C-States affect power consumption to a higher degree than P-States and with lower latencies too.

- Studies on recent Intel processors have shown that these latencies are slightly increasing over the years, with up to ~40 µs when waking up from a deeper sleep state.

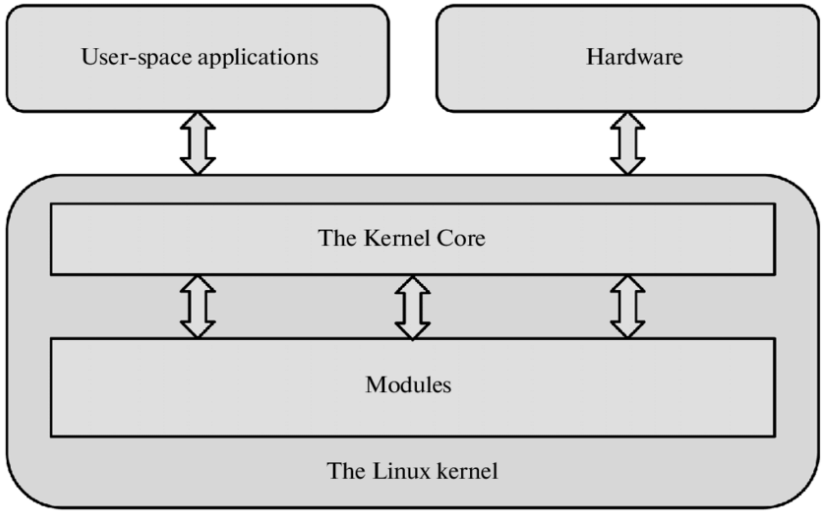


- The more is disabled in a C-State, the higher is the latency when waking up from that state.

From Theory to Practice

- Triggering these states require the running software to have unrestricted access to 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.
- The solution?

We create and load a new module that allows software to request new states!



Introducing Powerctl

- It is a Linux kernel module that introduces a suite of system calls that allow user-space software to change hardware states by writing on the MSRs responsible of such states' transitions or by executing privileged operations.
- Why system calls?

  Because they allow software to target new states while keeping latency as low as possible, in contrast to other solutions that are based on the utilization of the pseudo-file system.

- This is critical since we want to request these states on the fly during synchronization.

Powerctl – 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!
- How to avoid this issue?

We introduce a probing system and a thread-safe internal data structure that keeps track of all the various requests, so that Powerctl can always enforce the correct performance profile for each thread after a context switch while keeping latency at minimum.

Testing Powerctl – Methodology I

- To test Powerctl, we first created custom waiting policies for MCS.
- Why MCS?

> Because it can be considered the state of the art in terms of spinlock algorithms, since it ensures a FIFO policy, lack of memory contention and a relatively small space cost.

- What are these custom waiting policies?

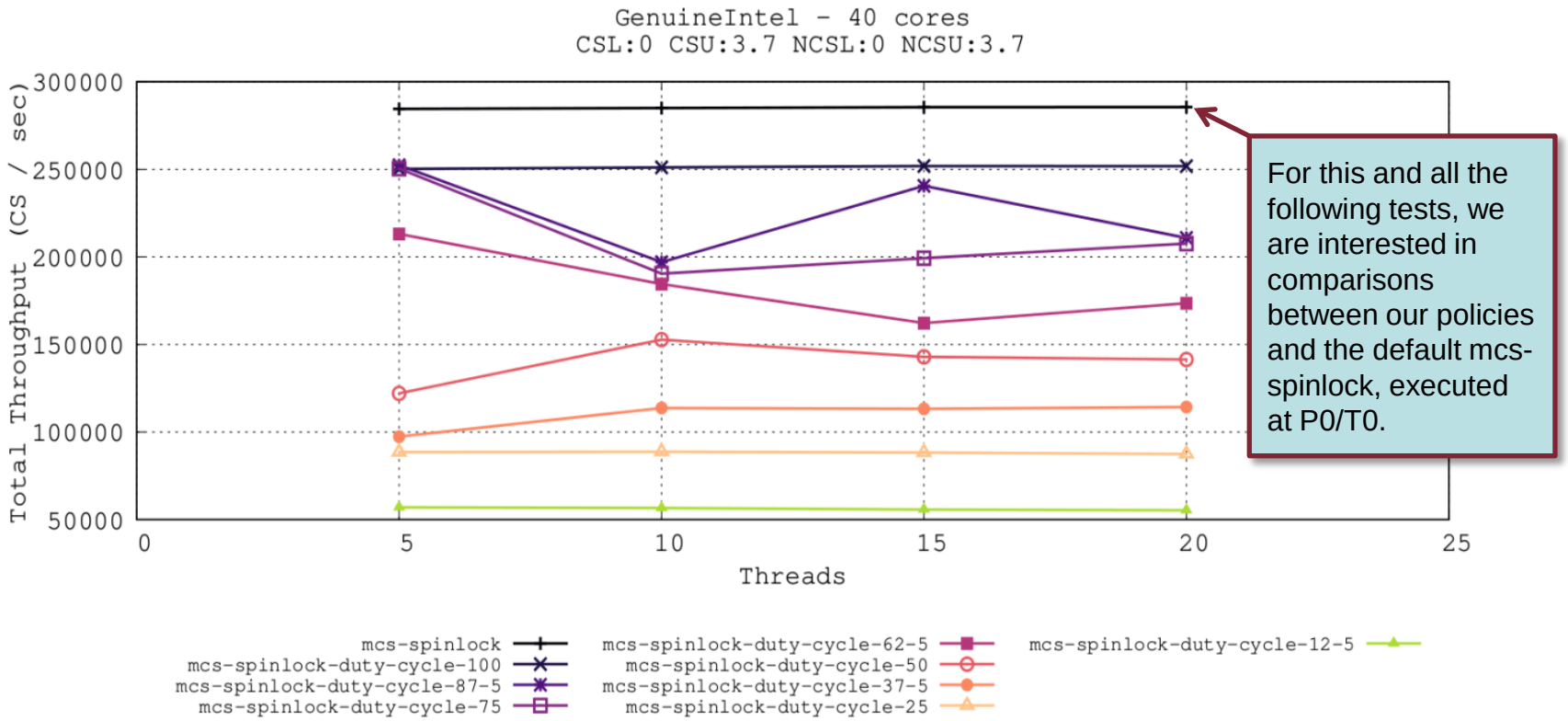
> These policies allow a thread to target a hardware state when it fails to acquire the lock, so that it keeps spinning in a depowered processor. When it acquires the lock, it then restores the processor to its default settings.

Testing Powerctl – Methodology II

- We then tried these custom waiting policies on benchmark tests that differ in various aspects, such as the us sizes of the critical and non-critical sections, and the number of active cores.
- Each single test was performed multiple times in order to minimize the possible effects of outlier executions.
- We were interested in the efficiency of the execution, which is the ratio between the throughput (expressed in number of lines executed in the critical section per second) and the energy consumption (expressed in Joules).
- The system 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.

Results and Discussion – Short CS

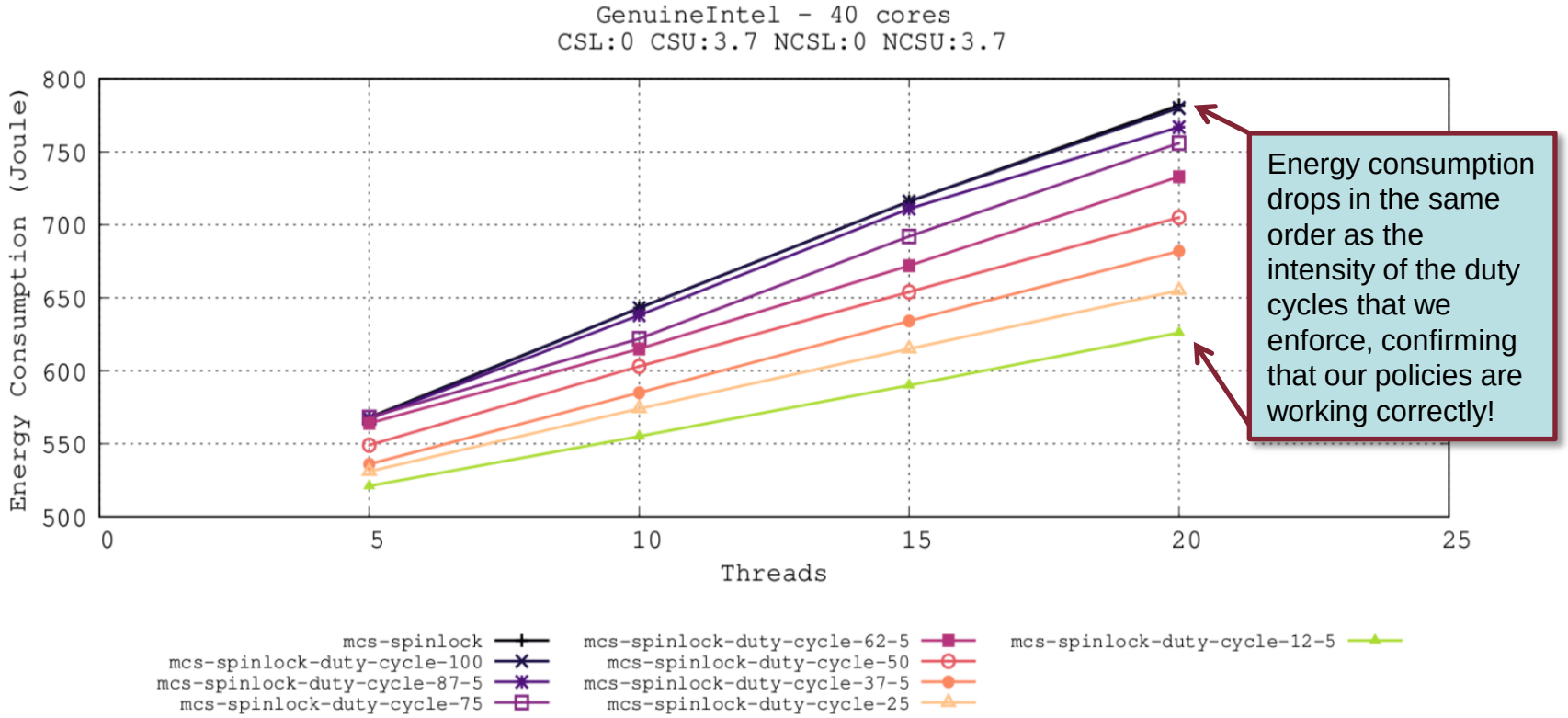
- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



For this and all the following tests, we are interested in comparisons between our policies and the default mcs-spinlock, executed at P0/T0.

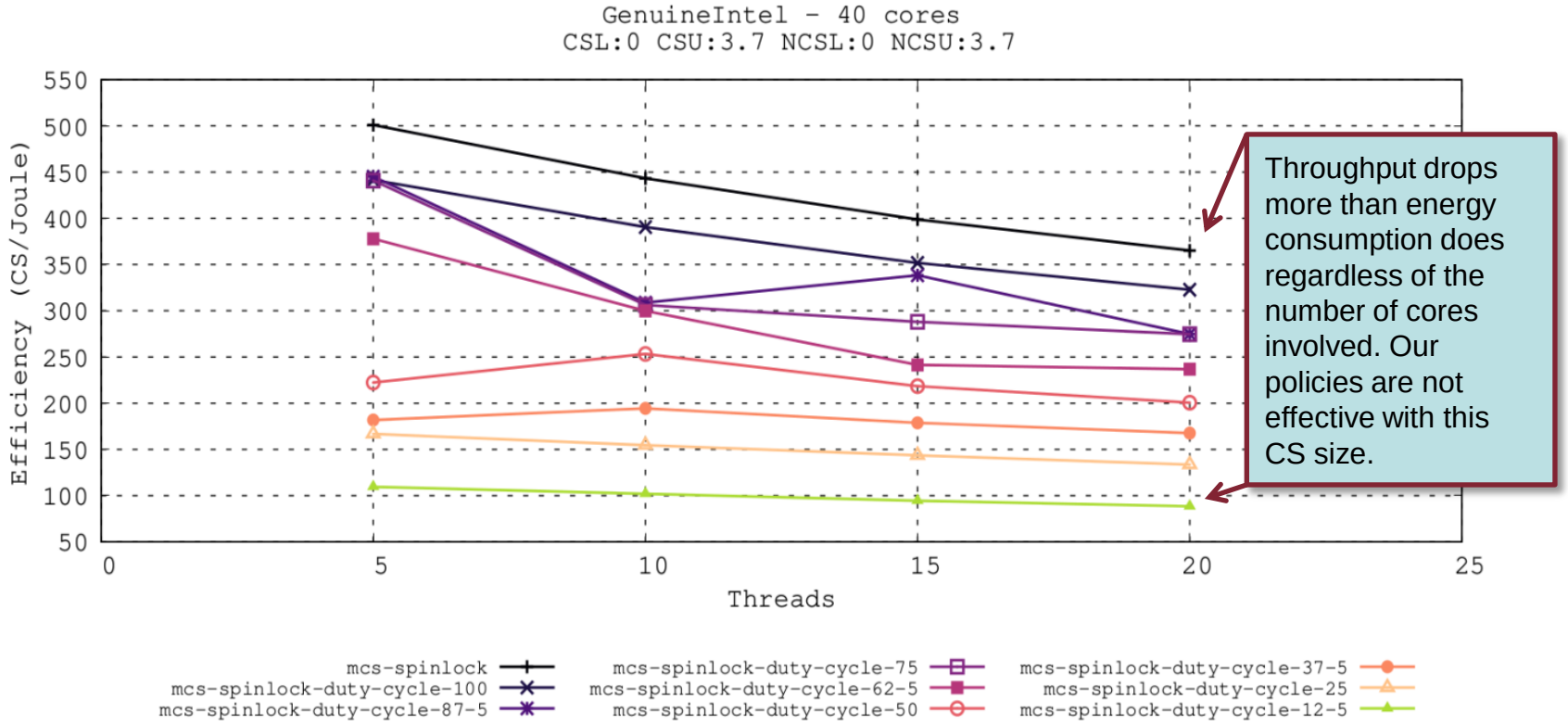
Results and Discussion – Short CS

- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



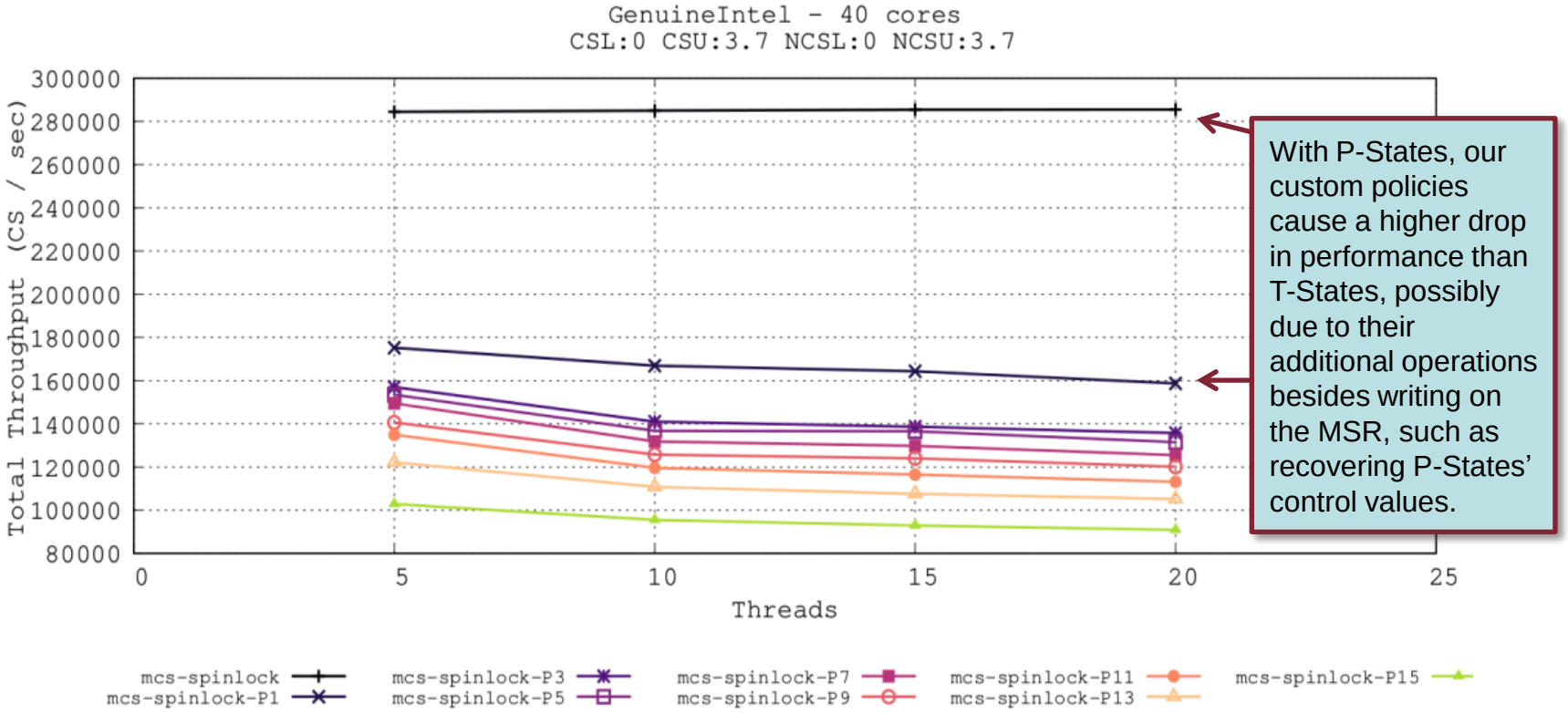
Results and Discussion – Short CS

- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



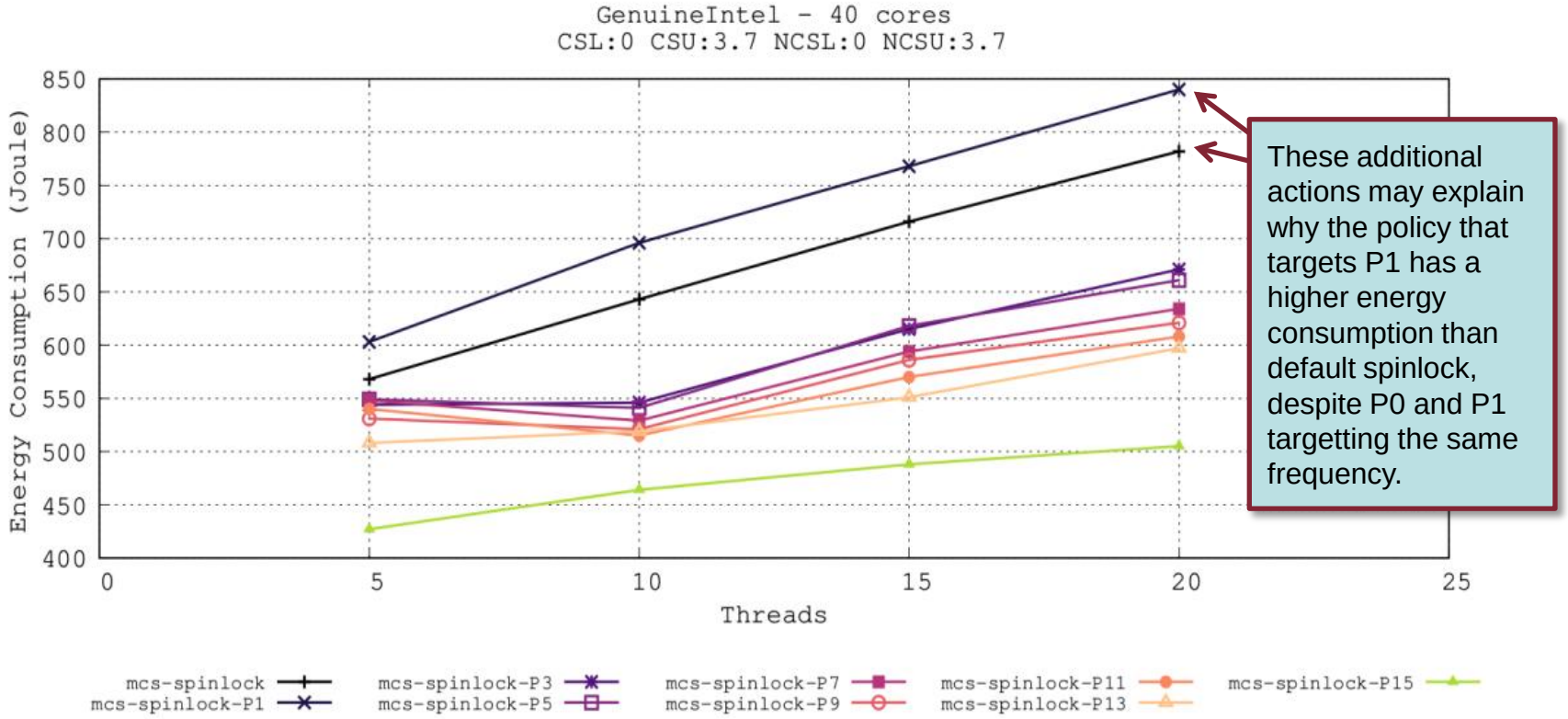
Results and Discussion – Short CS

- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



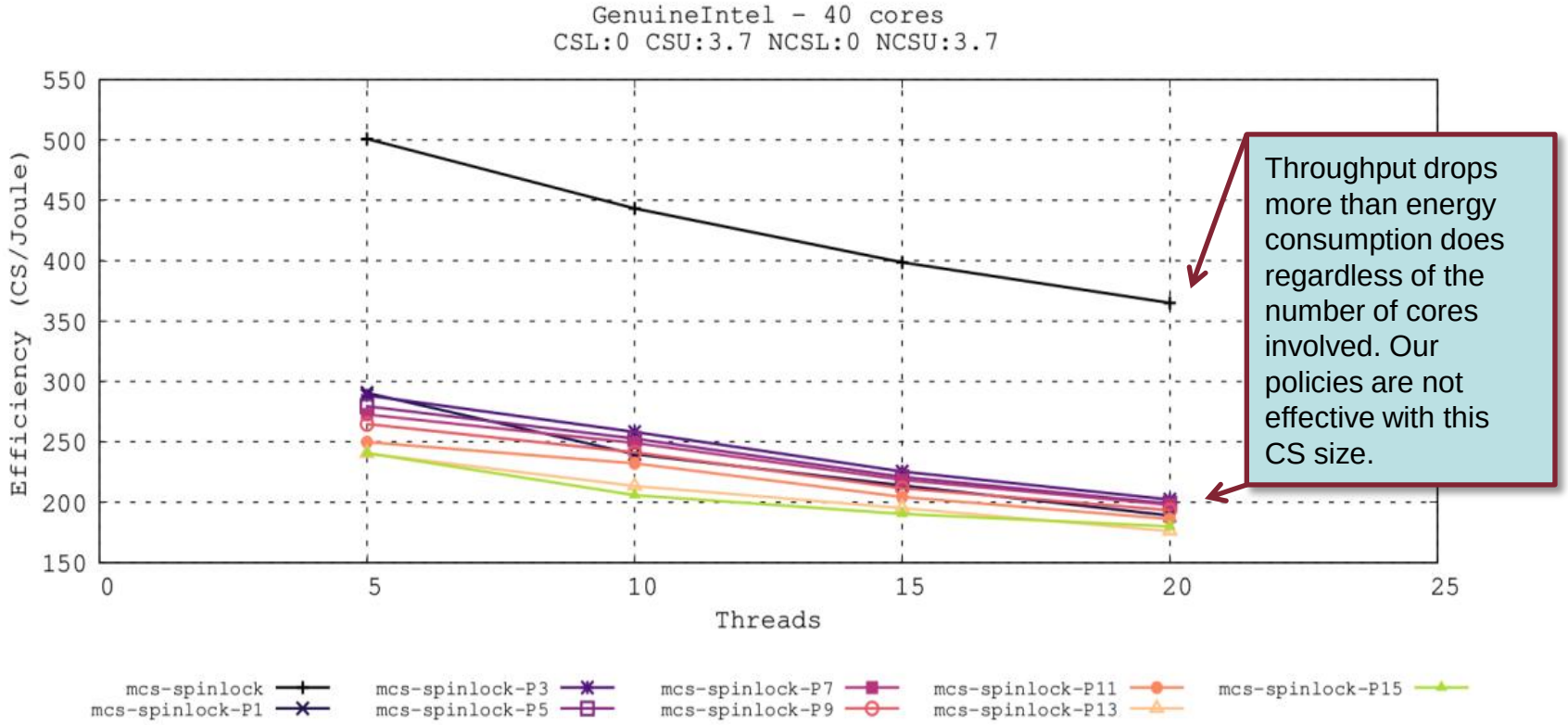
Results and Discussion – Short CS

- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



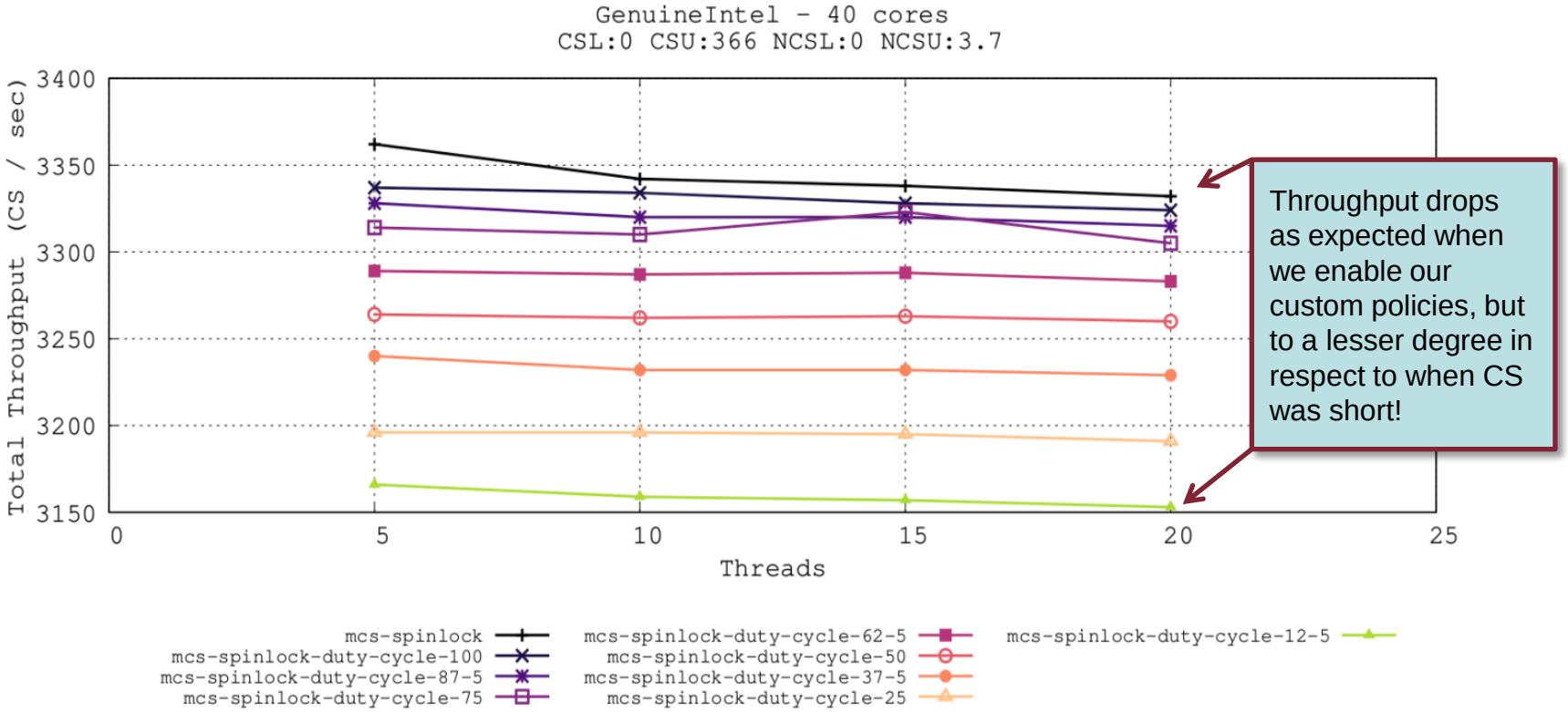
Results and Discussion – Short CS

- Tests with a small critical section didn't bring positive results for us. The lock is handed over very quickly between threads, which leads to frequent writes on the MSRs that cause a drop in the throughput.



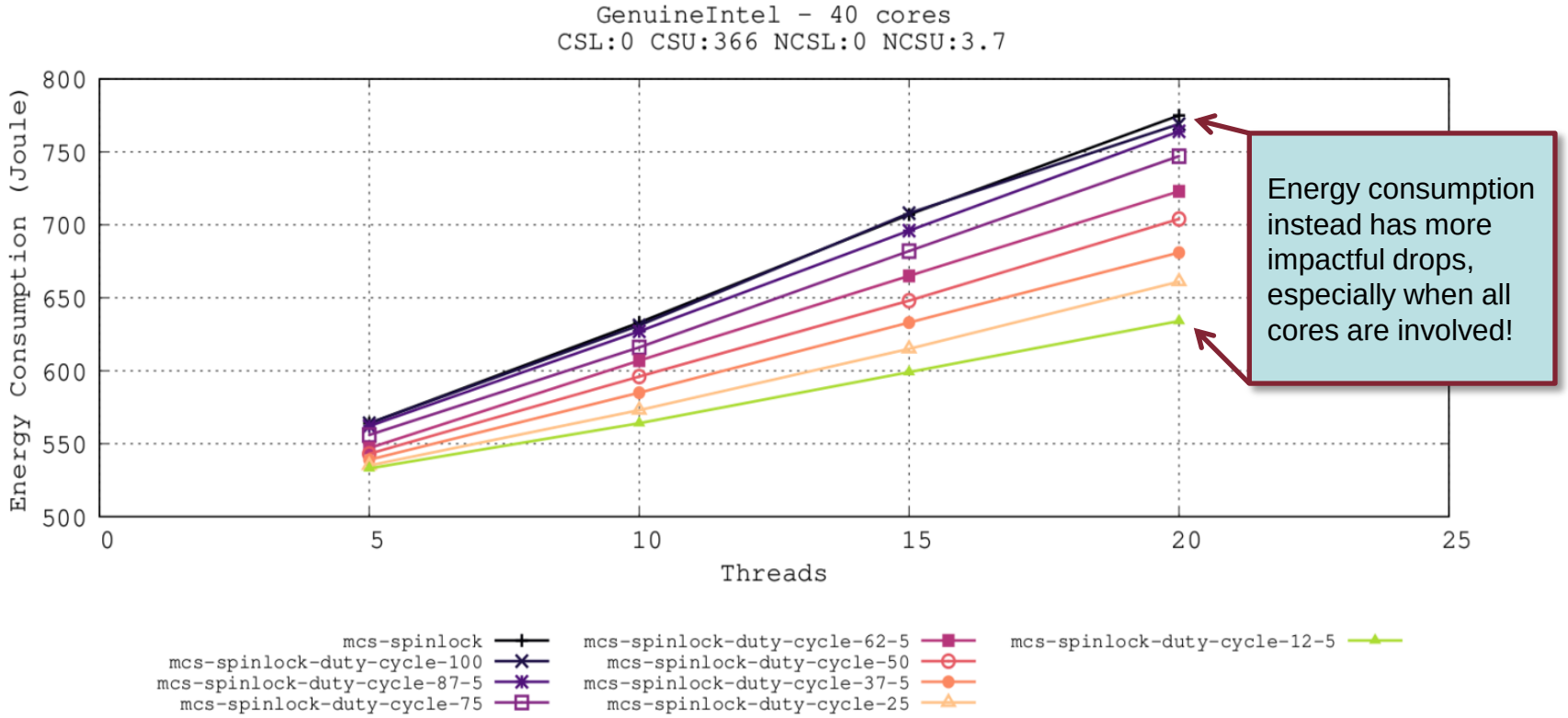
Results and Discussion – Long CS

- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



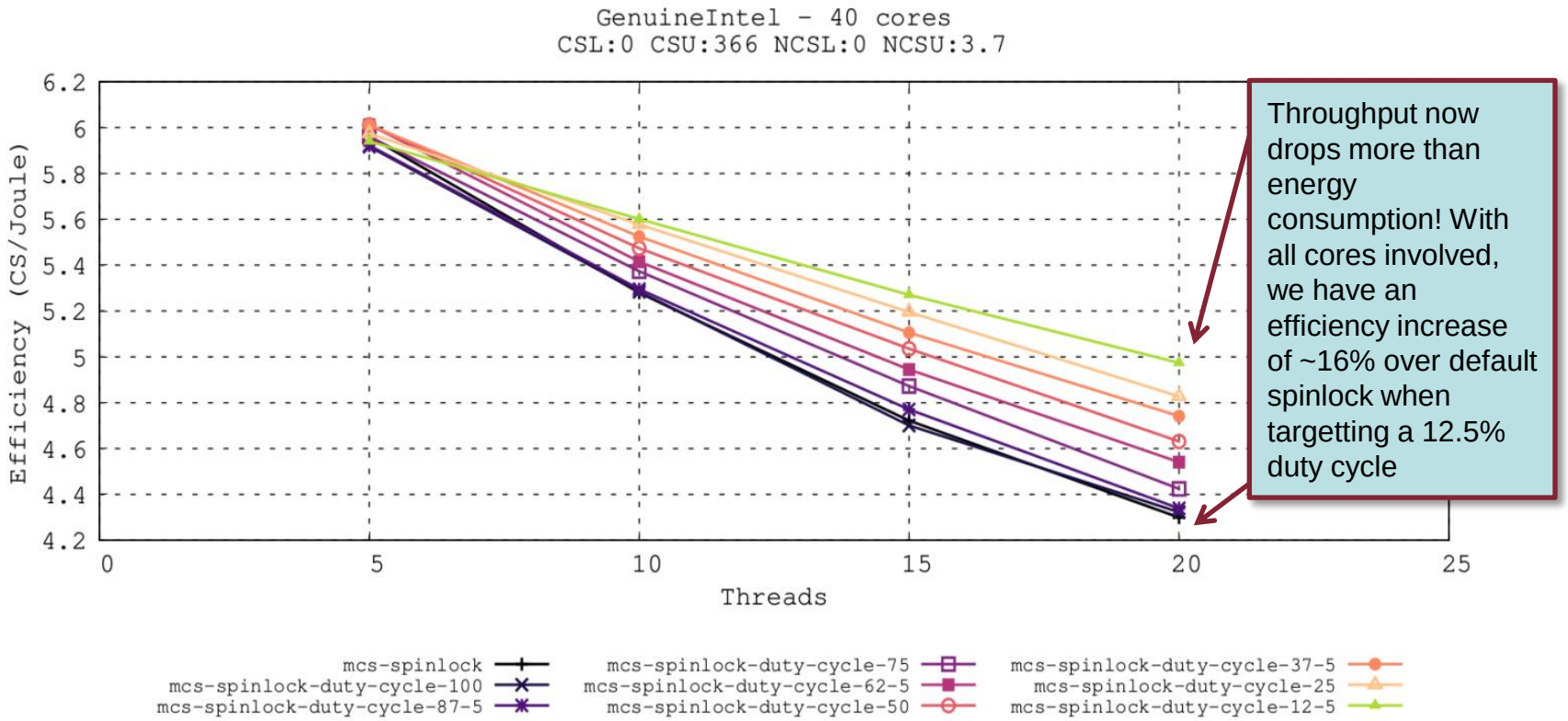
Results and Discussion – Long CS

- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



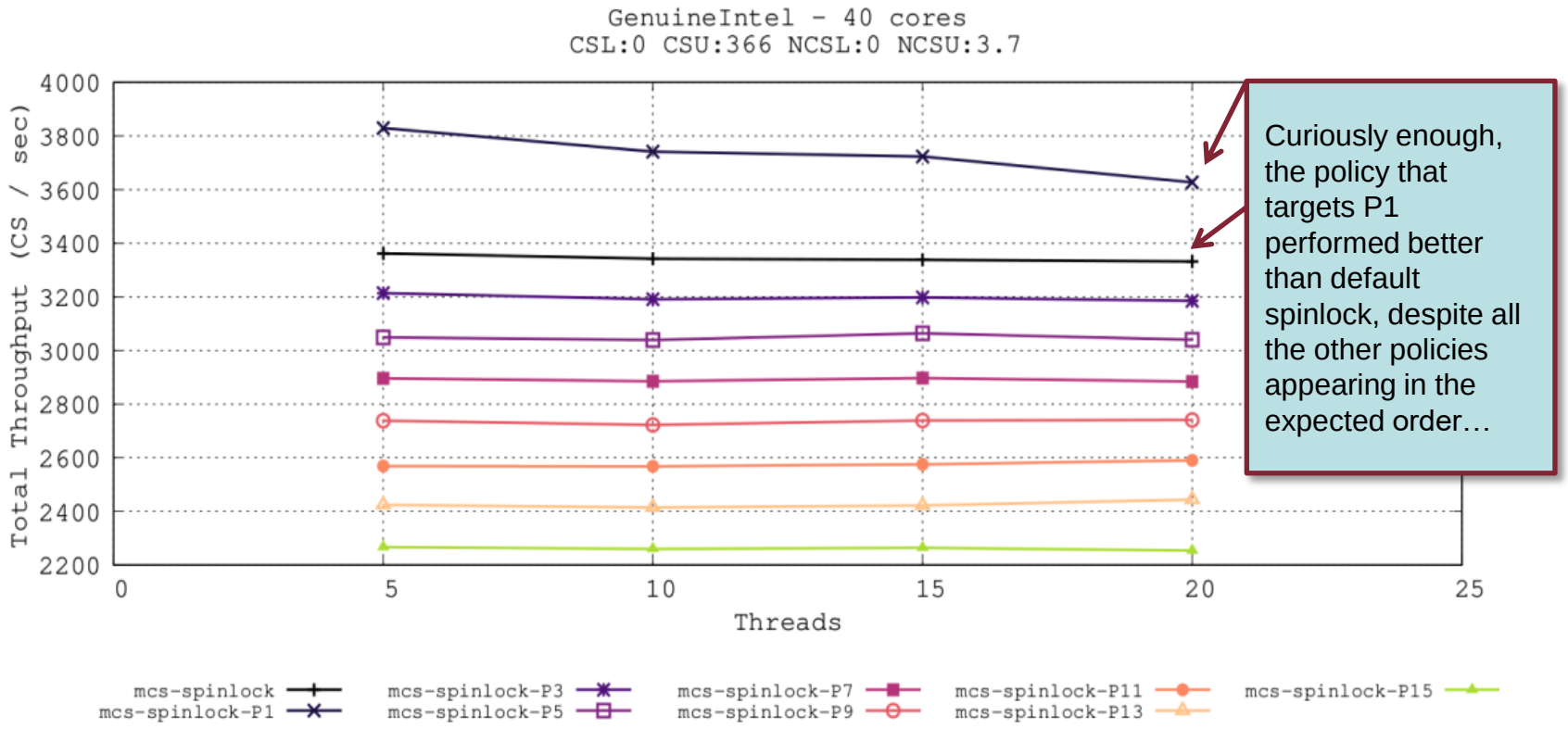
Results and Discussion – Long CS

- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



Results and Discussion – Long CS

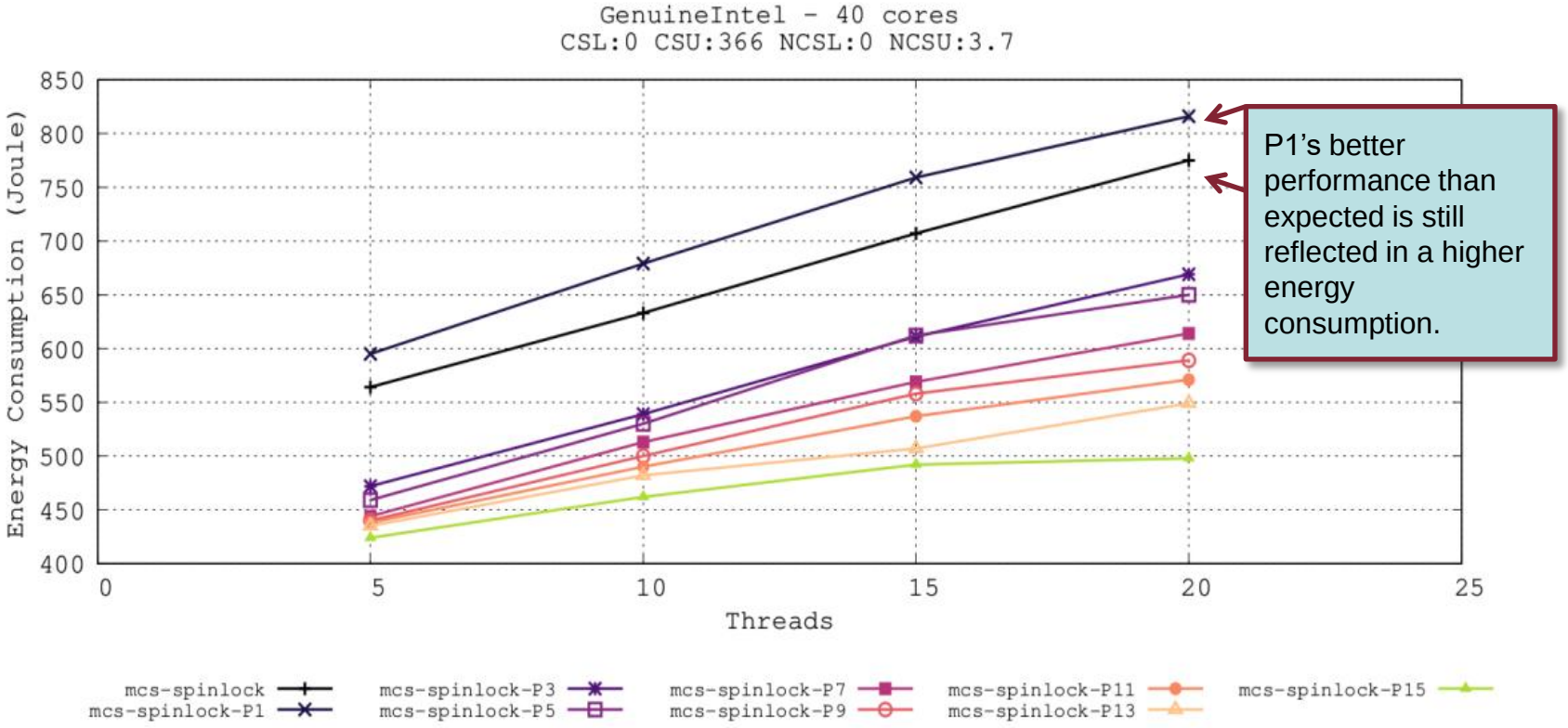
- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



Curiously enough, the policy that targets P1 performed better than default spinlock, despite all the other policies appearing in the expected order…

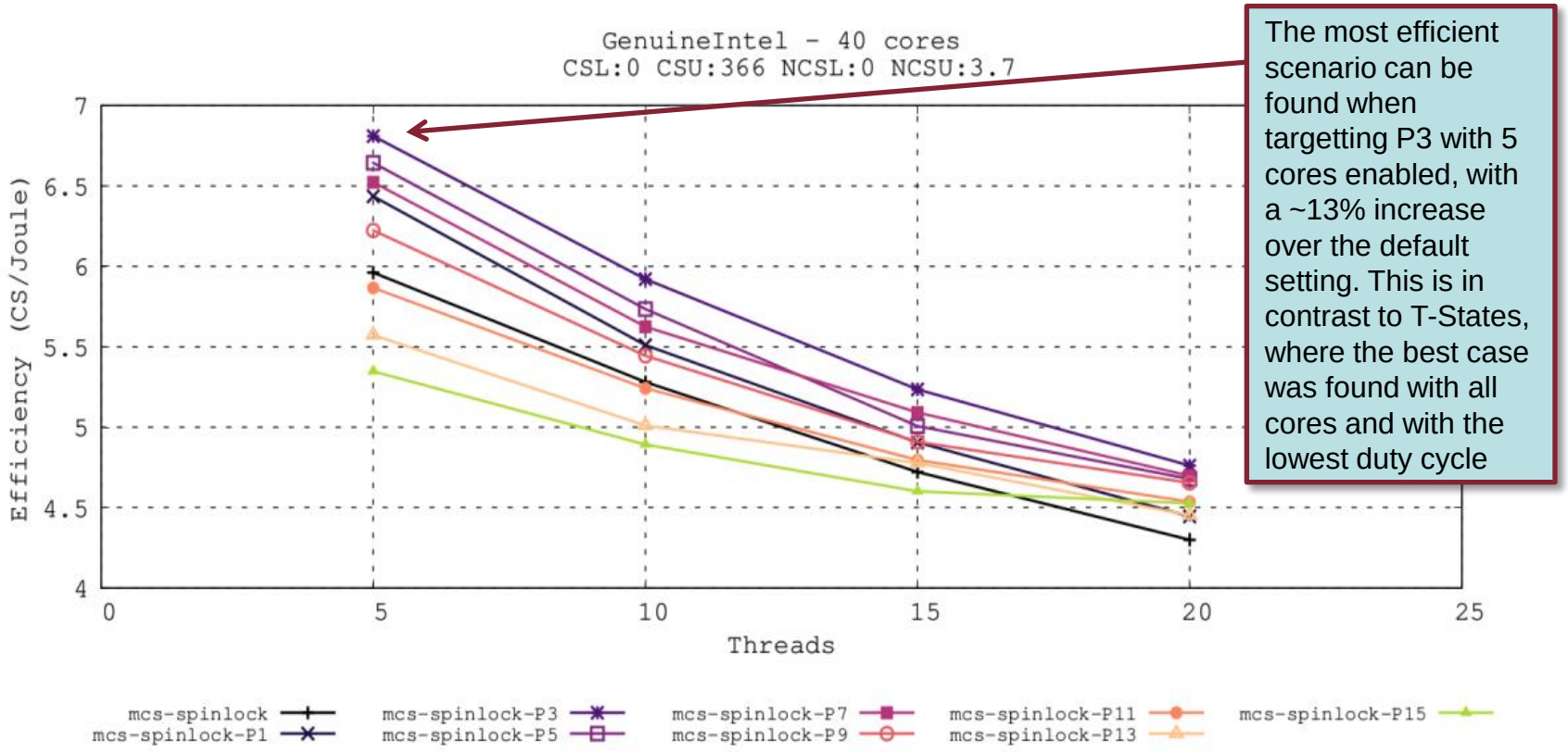
Results and Discussion – Long CS

- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



Results and Discussion – Long CS

- By increasing the length of the critical section instead, we obtained positive results for our custom waiting policies, regardless of the size of the non-critical section.



The most efficient scenario can be found when targetting P3 with 5 cores enabled, with a ~13% increase over the default setting. This is in contrast to T-States, where the best case was found with all cores and with the lowest duty cycle

Conclusions and Looking Ahead

- We could identify scenarios where our custom waiting policies brought tangible benefits to the efficiency, meaning that changing hardware states during spinlock-based synchronization is a strategy that is worth looking into for further expansions and experimenting.
- In particular we intend to:
  - Add new features to Powerctl, such as the ability to support hyper-threading.
  - Apply these custom waiting policies to other spinlock algorithms besides MCS, so to possibly find new better candidates.
  - Accurately test C-States, which in preliminary tests have shown great potential, and further benchmark scenarios that differ in the size of the critical section, since we have seen it has a great impact on the efficiency.
  - Apply Powerctl, once it has been enhanced with positive improvements, to a robust benchmark suite for parallel applications (ex. Splash-3), so to predict how its benefits would carry over to real applications.

Thanks for Your Attention!

- Interested in the topic? Here is a brief list of my main sources:

- 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.
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B
- R. Schöne, T. Ilsche, M. Bielert, D. Molka e D. Hackenberg, «Software Controlled Clock Modulation for Energy Efficiency Optimization on Intel Processors,» 2016.
- P. Jay Salzman, M. Burian e O. Pomerantz, The Linux Kernel Module Programming Guide, 2007.