# MCS with Catnap policy Testing Report

High Performance Computing

Luca Sannino 1542194

# PREMISE AND CONTEXT

# PREMISE

- The following work is based on a paper written by Youngjoo Woo, Sunghum Kim, Cheolgi Kim and Euiseong Seo, published in 2018, that describes a waiting policy for multi-core systems called Catnap.
- The objective of this project was turn the pseudocode of Catnap into reality and to test it in a real environment.
- This presentation contains the results of such tests.
- The original paper: https://ieeexplore.ieee.org/document/8978918

# CATNAP IN SHORT

- It’s a waiting policy that is supposed to be a middle ground between spinning and sleeping, since threads are technically considered to be active by the system, even though in reality they operating within processors that are working in a low profile/optimized fashion.
- This approach is supposed to save up on performance and energy consumption, and it's achieved by using the MONITOR/MWAIT instructions, which can be executed only at privilege level 0 (kernel mode), hence the need for a custom system call that implements Catnap.

# THE MONITOR INSTRUCTION

- MONITOR: This instruction monitors an address range that has been given in input. If someone writes in that address range, MONITOR is notified and wakes up any thread that is waiting with MWAIT, an instruction which MONITOR is paired with. MONITOR also accepts two other inputs, an hint and an extension, but they are both generally initialized with 0.

# THE MWAIT INSTRUCTION

- MWAIT: If paired with MONITOR, any thread that encounters this instruction will stop until the address range observed by MONITOR is written by someone or until an external factor breaks the wait, such as debug/machine check interruptions and signals like INIT and RESET. MWAIT accepts two inputs: a hint, which specifies in which C-state the processor should go in during the wait, and a wakeup_mode, which specifies if interruptions, even the masked ones, should interrupt the wait.

# SOME COMMON C-STATES

- C1: Halt. Stops CPU main internal clocks via software; bus interface unit and APIC are kept running at full speed.
- C2: Stop Clock. Stops CPU internal and external clocks via hardware.
- C3: Sleep. Stops all CPU internal clocks.
- C4: Deeper Sleep. Reduces CPU voltage.

NOTE: All states described above are supposed to be general: there could be differences across the various types of processors!

# ORIGINAL PSEUDOCODE

```
while True do
    monitor_target ←address_of (locked)
    M ON IT OR (monitor_target)
    if Locked then
        break
    else
        hint ←designated_C–state
        wakeup_mode ←interrupt_wakeup_disable
        M W AIT (hint, wake_up_mode)
    end if
    if Locked then
        break
    end if
end while
```

Remember: it’s important to check that MWAIT has been interrupted for the right reason, that is someone has written on the observed address!

# TESTING TOOLS

- Litl: Allows executing a program based on Pthread mutex locks with other locking algorithms. It also lets the introduction of personal locking and policy algorithms, so I introduced four variants of Catnap (each targetting a different C-State) that are used as waiting policy by MCS.
  Link: https://github.com/multicore-locks/litl
- Lockbench: A benchmark suite for evaluating lock implementations. It lets simulate critical and non-critical sections of code, and its integration with Litl allows for quick and precise tests, and plotting of the results.
  Link: https://github.com/HPDCS/lockbench

# HOW TESTING WAS MADE

- There were four tests: each one had different min-max values for the sizes of the simulated cs (critical section) and ncs (non-critical section).
- Each test is composed of three runs: each run simulated programs whose cs and ncs sizes are contained between the min-max values, that were tested using MCS with the following waiting policies: spinlock, spin-then-park, catnap-c1, catnap-c2, catnap-c3 and catnap-c4.
- Those tests were conducted with 1, 2, 3 and 4 concurrent threads.

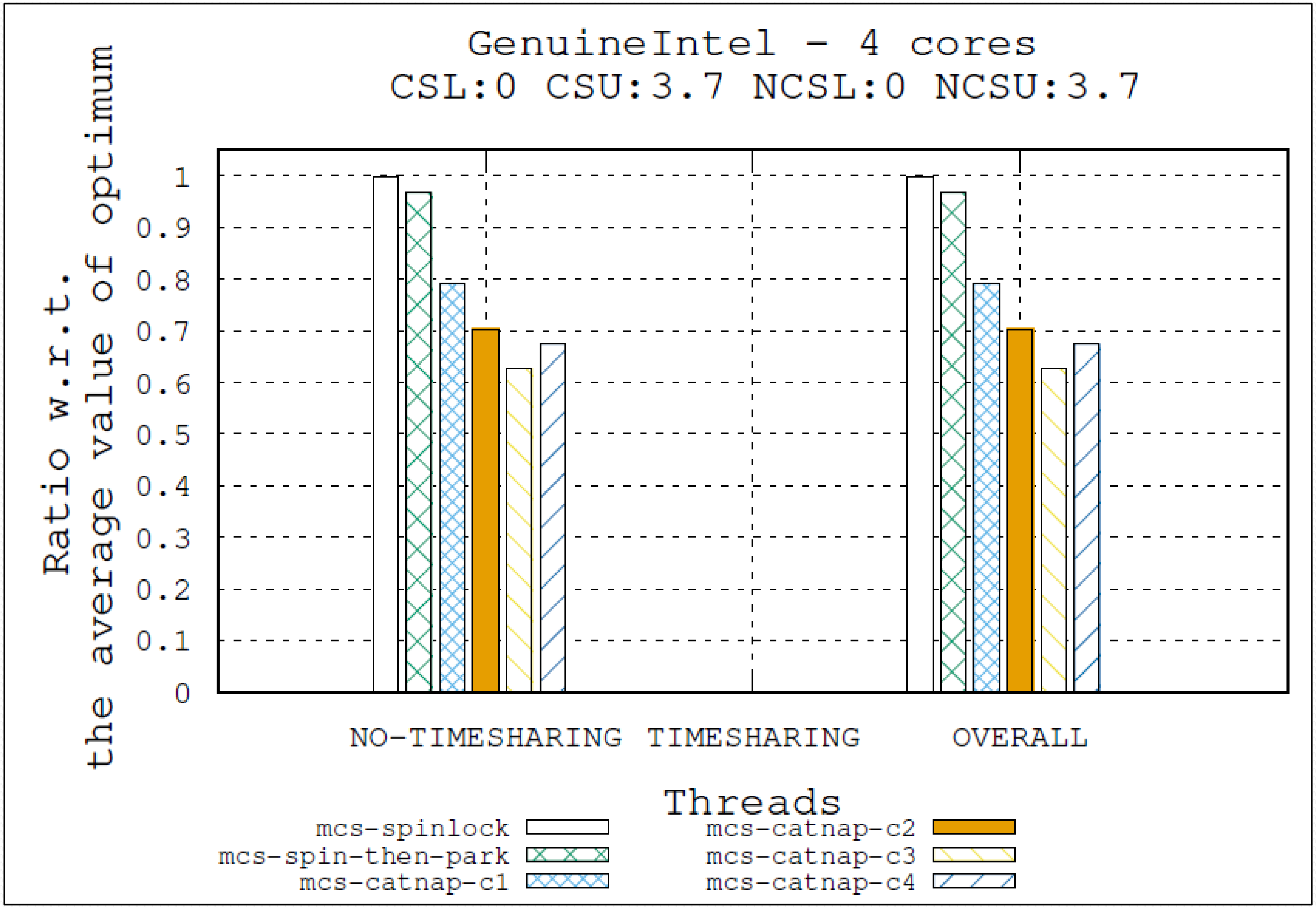
# SPECS OF THE PC USED IN TESTING

- Processor: Intel® Core™ i7-4510U CPU @ 2.00 GHz 2.60 GHz
- Number of physical cores: 2
  (This means a max of 4 concurrent threads)
- RAM: 12 GB
- OS: Ubuntu 18.04

# TEST 1

## CS AND NCS ARE BOTH SHORT

# AVERAGE OF OPTIMUM VALUE

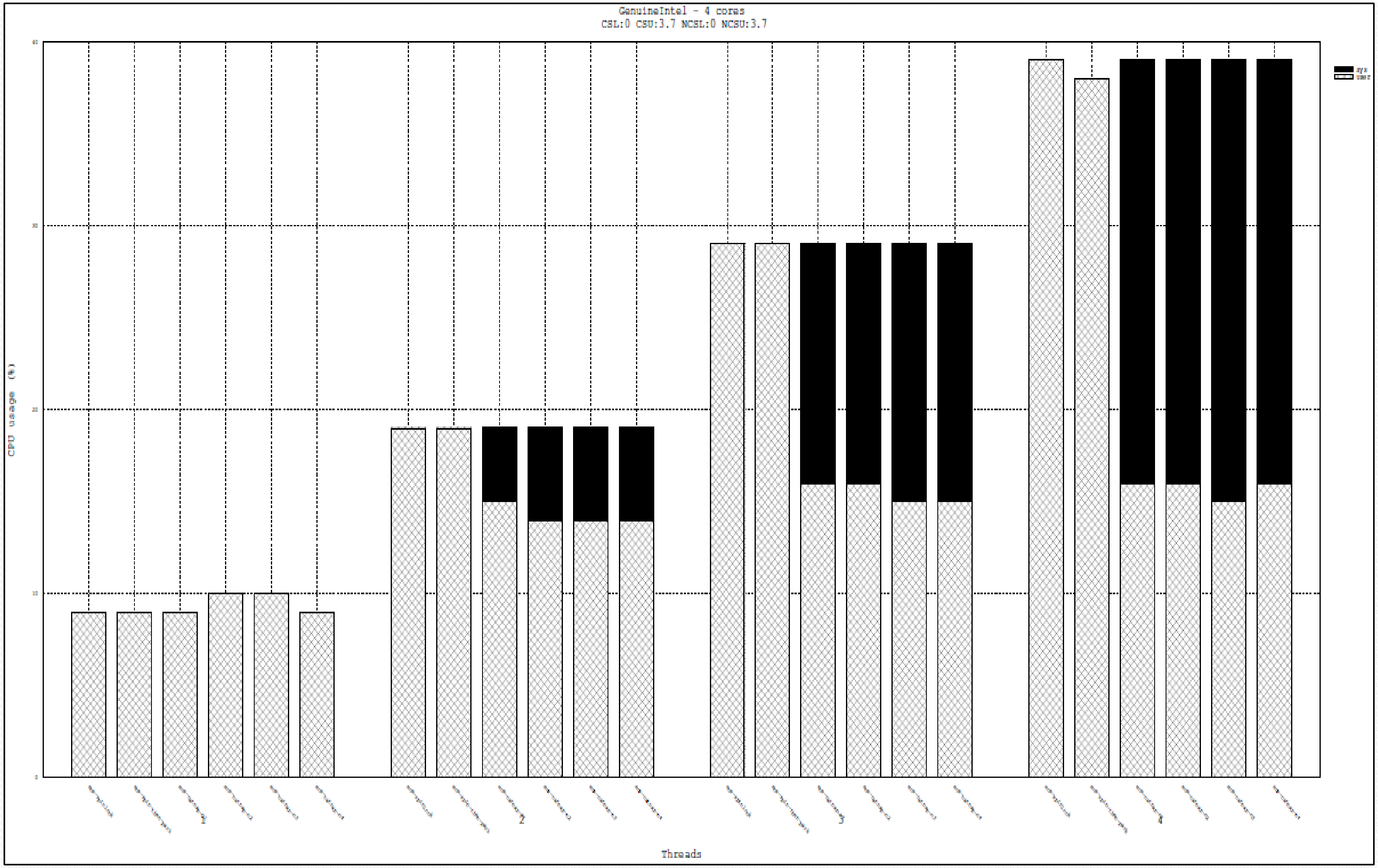


# CPU USAGE

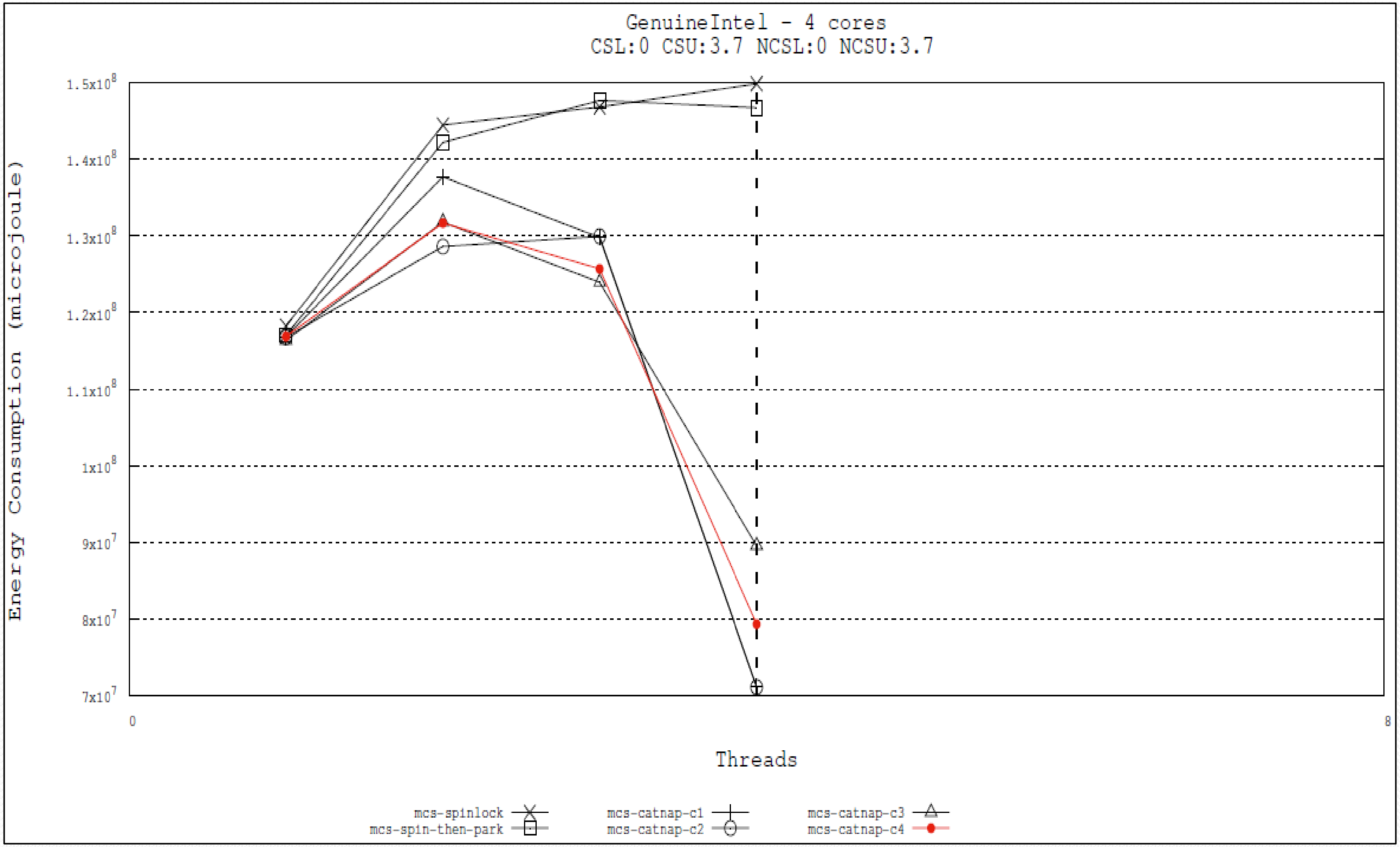
For each cluster, from left to right, the policies are: spinlock, spin_then_park, catnap_c1, c2, c3, and c4

Gray: user lvl
Black: kernel lvl

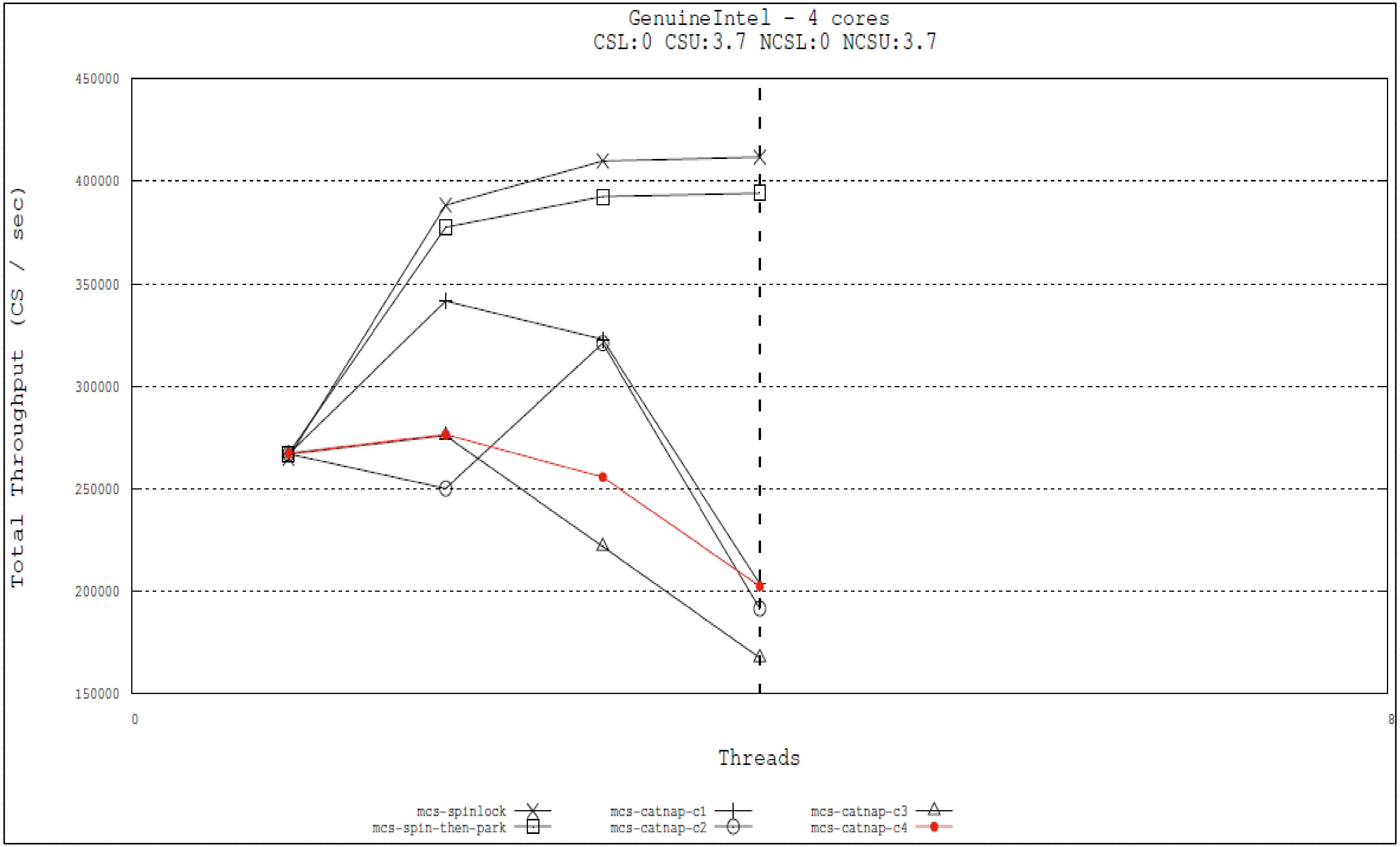
The CPU usages are bottom to top:
10%, 20%, 30% and 40%



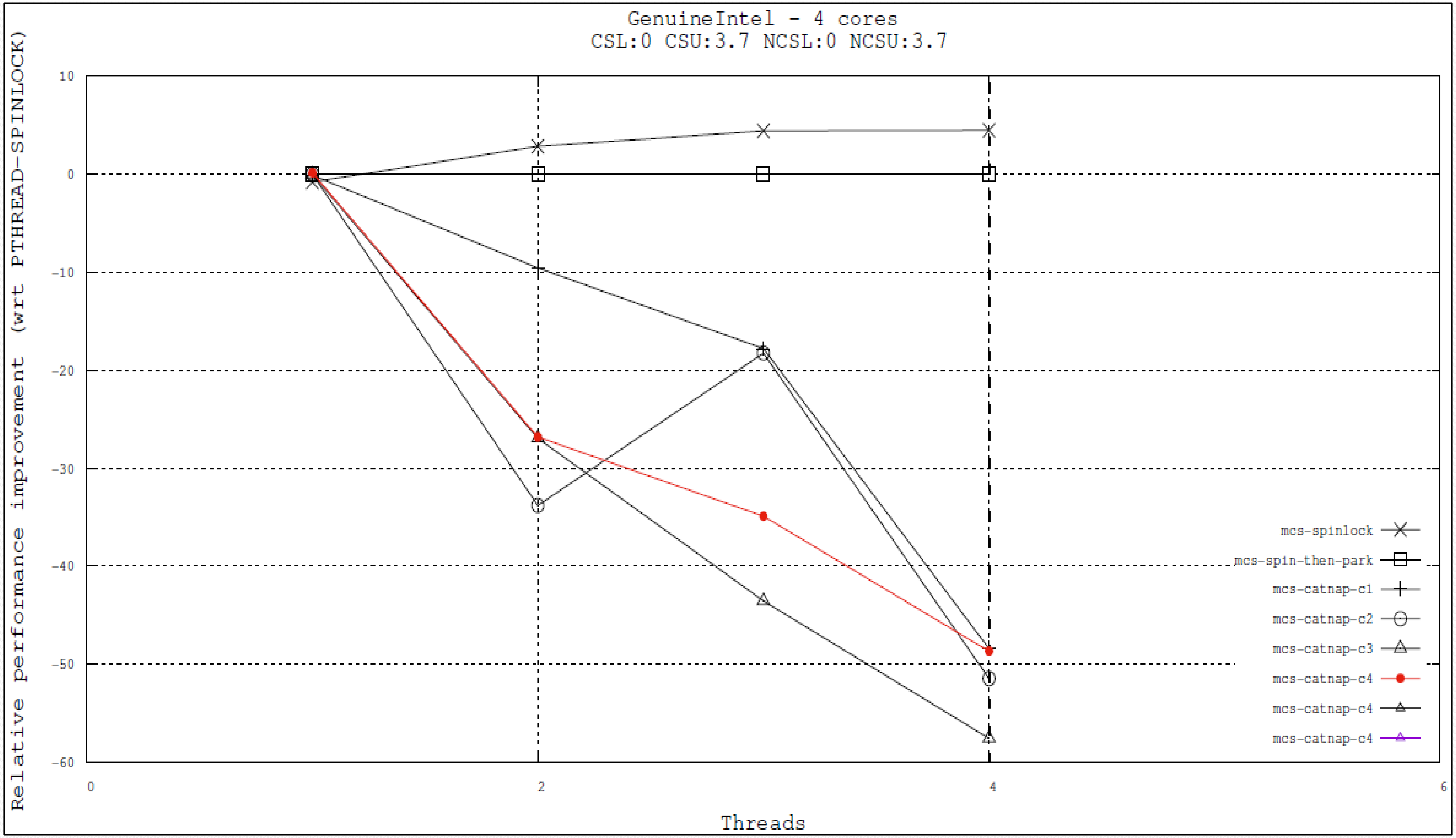
# ENERGY CONSUMPTION



# TOTAL THROUGHPUT



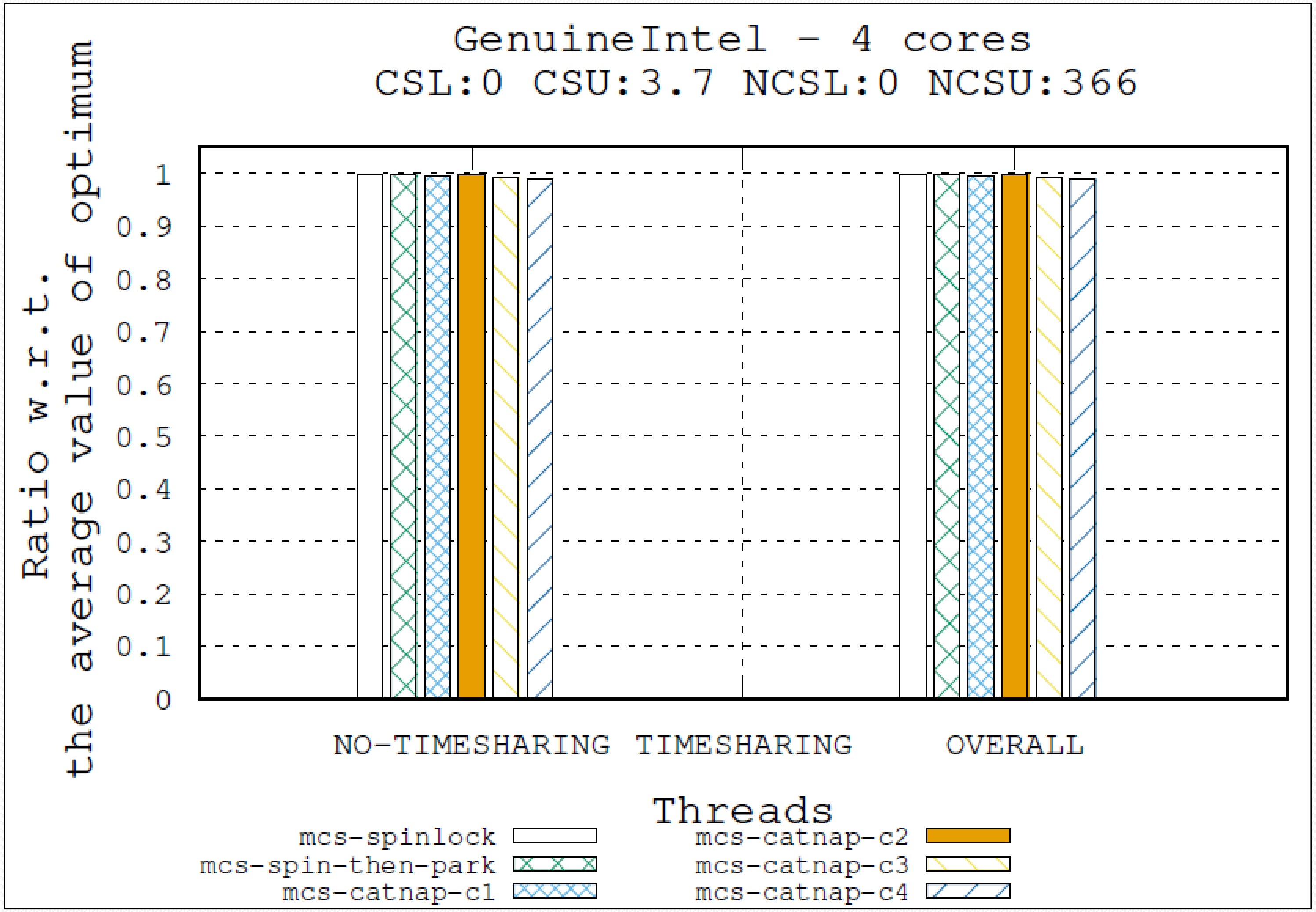
# RELATIVE IMPROVEMENT



# TEST 2

## NCS WAY BIGGER THAN CS

# AVERAGE OF OPTIMUM VALUE

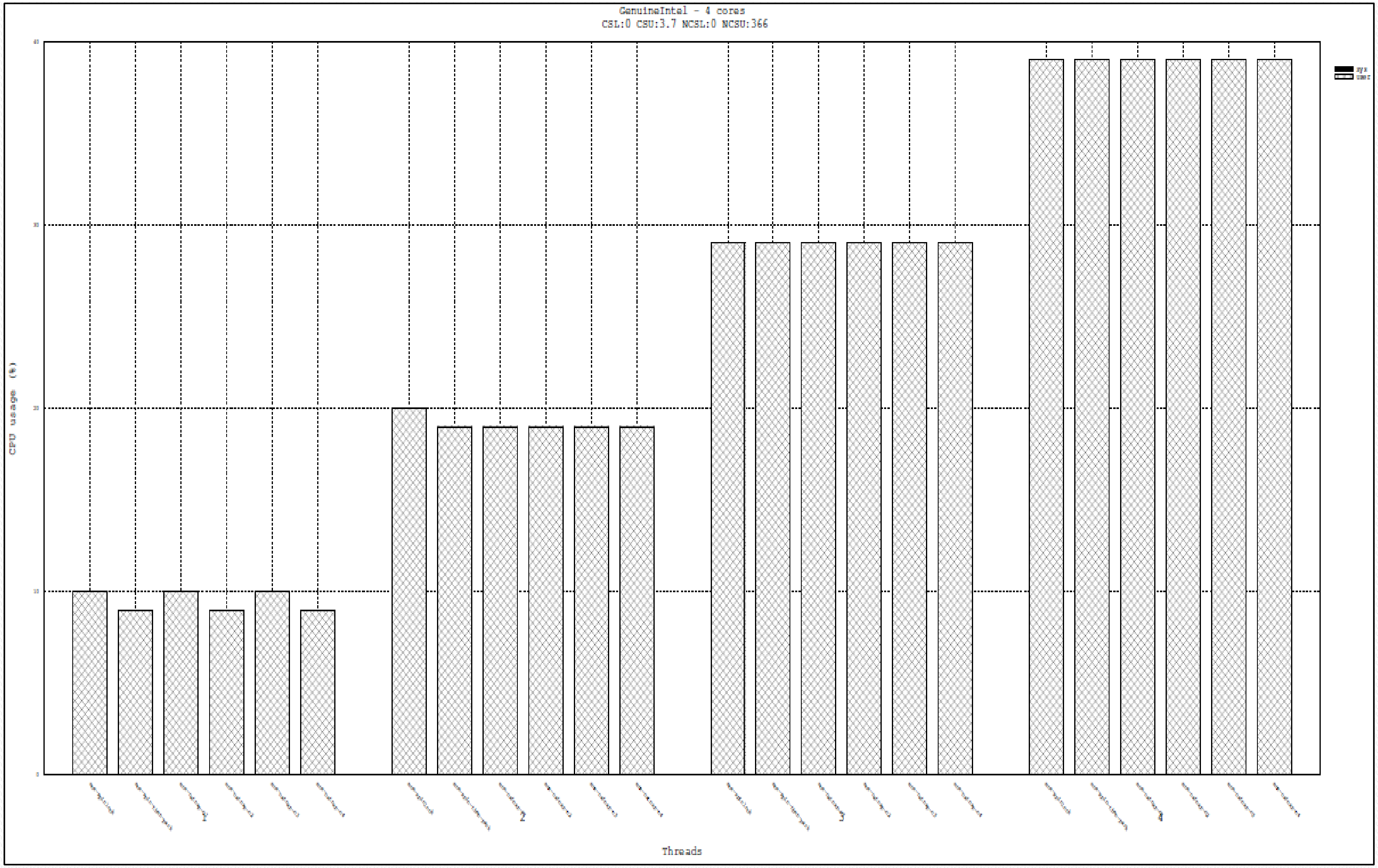


# CPU USAGE

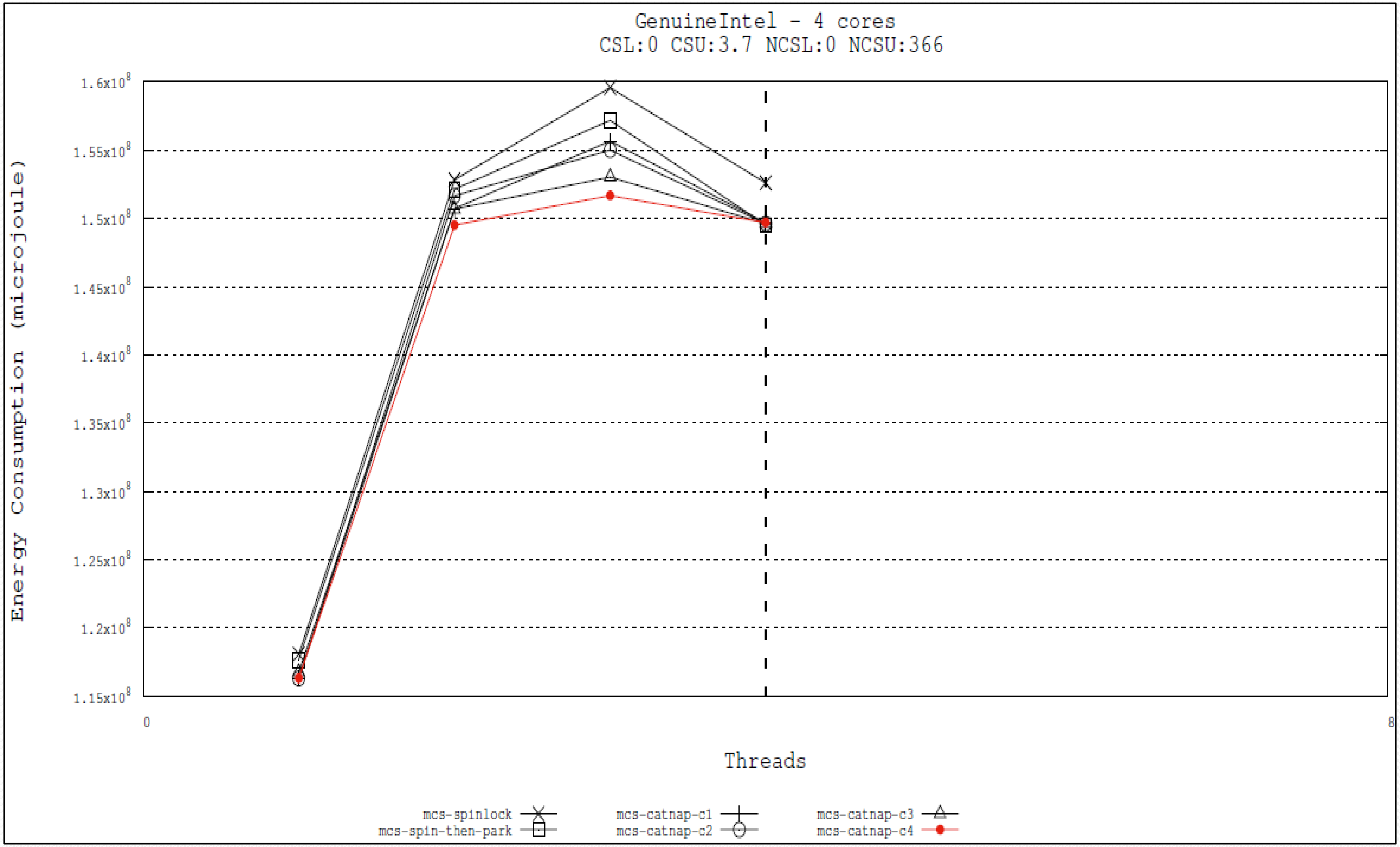
For each cluster, from left to right, the policies are: spinlock, spin_then_park, catnap_c1, c2, c3, and c4

Gray: user lvl
Black: kernel lvl

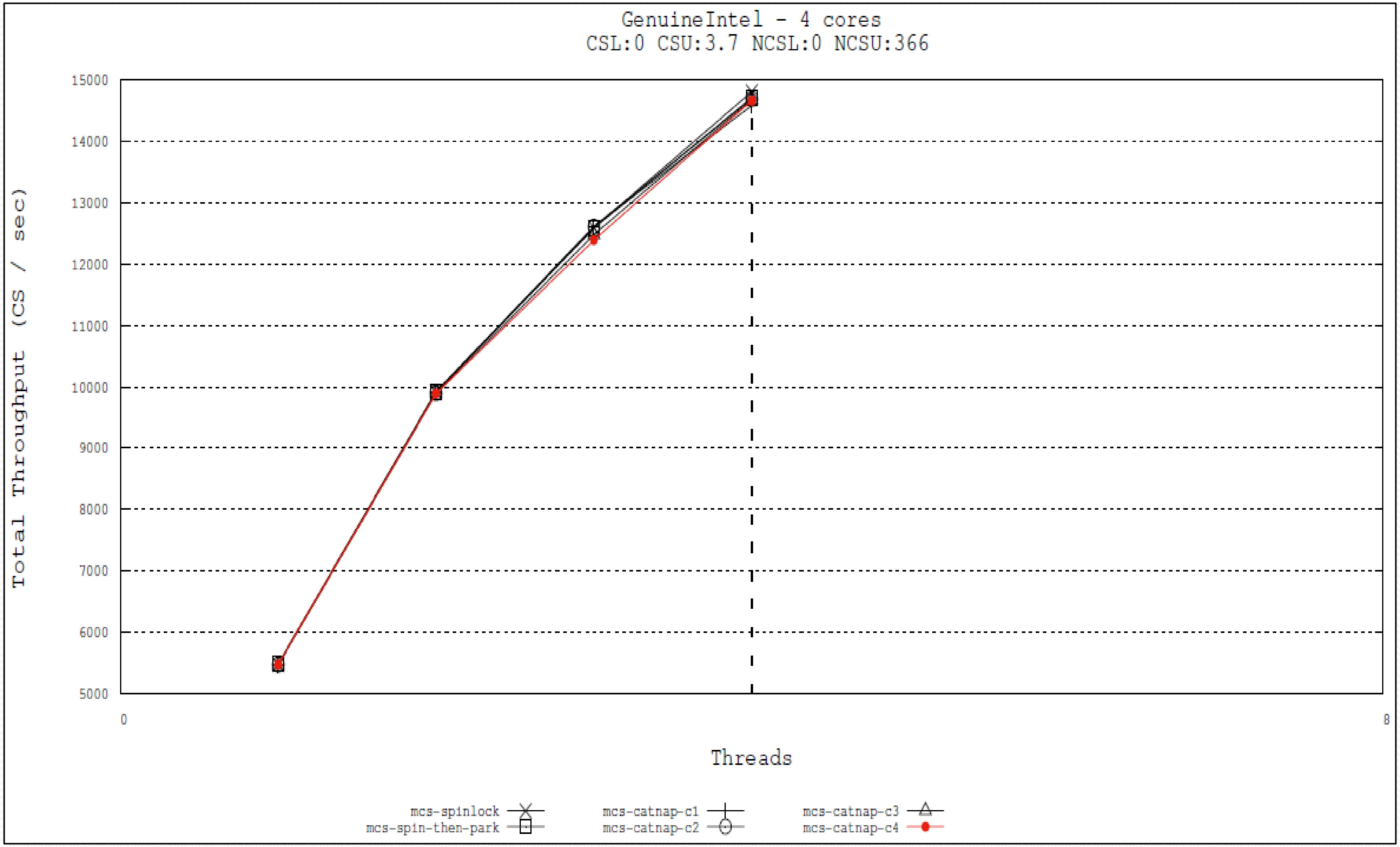
The CPU usages are bottom to top: 10%, 20%, 30% and 40%



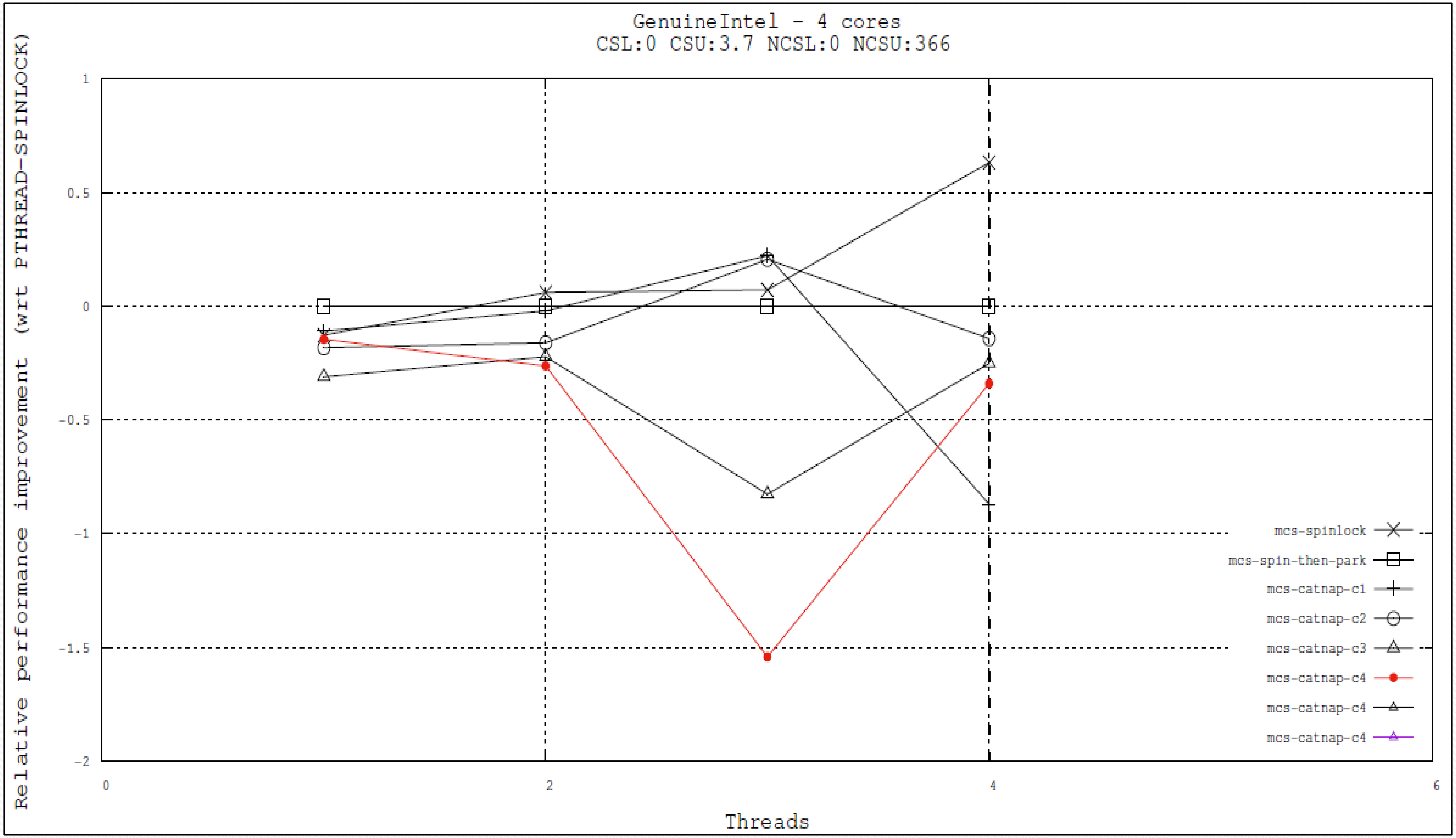
# ENERGY CONSUMPTION



# TOTAL THROUGHPUT



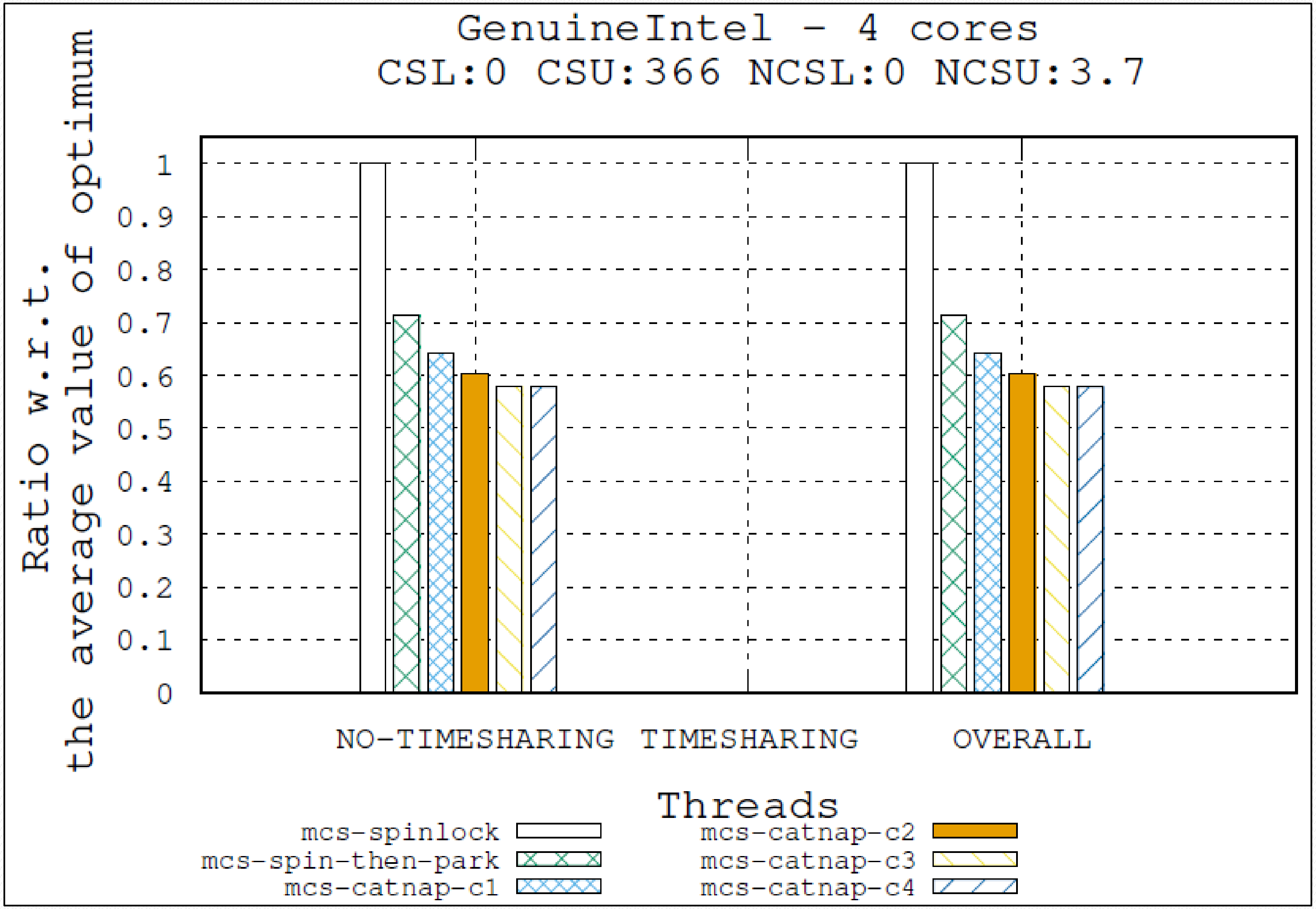
# RELATIVE IMPROVEMENT



# TEST 3

## CS WAY BIGGER THAN NCS

# AVERAGE OF OPTIMUM VALUE

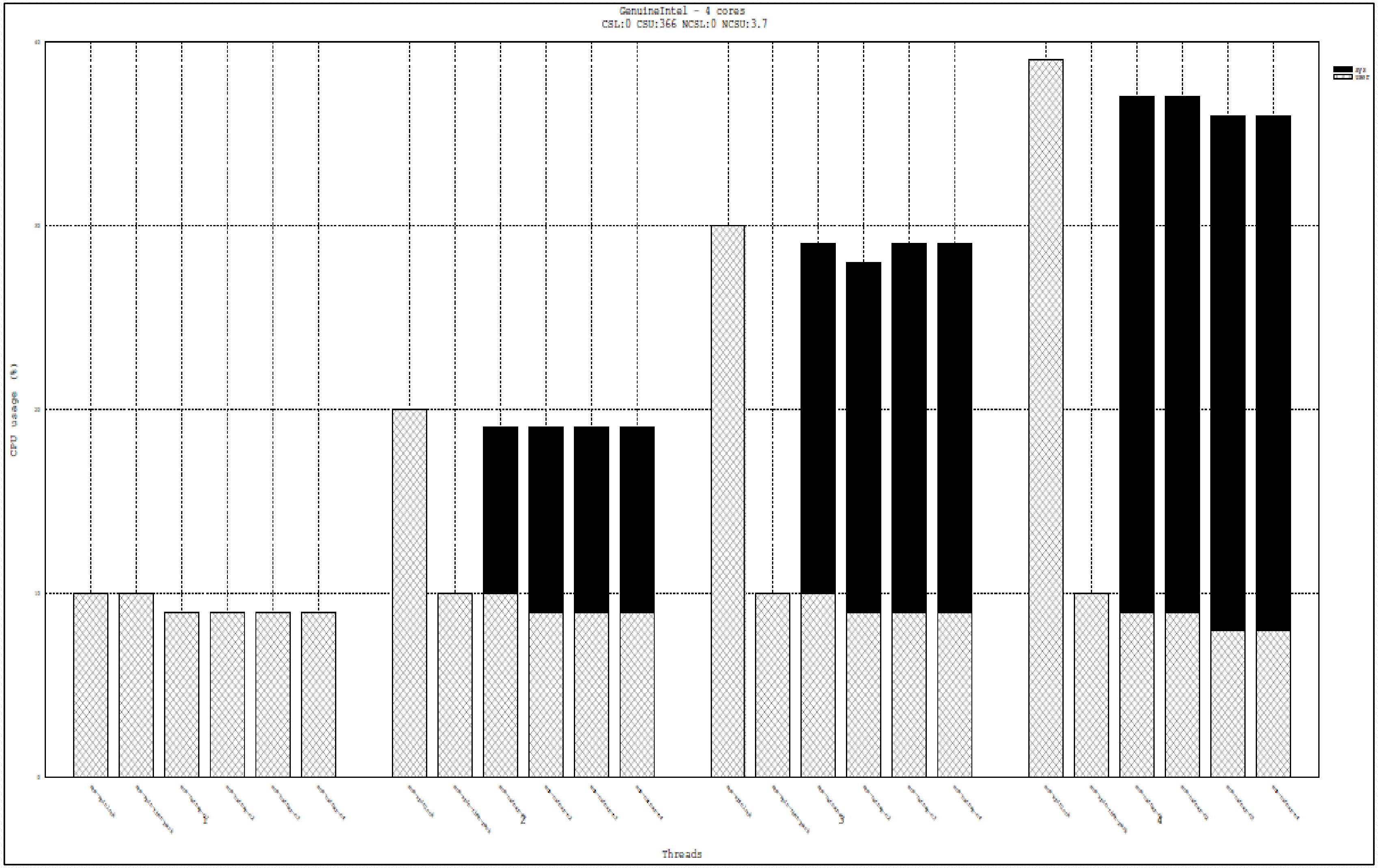


# CPU USAGE

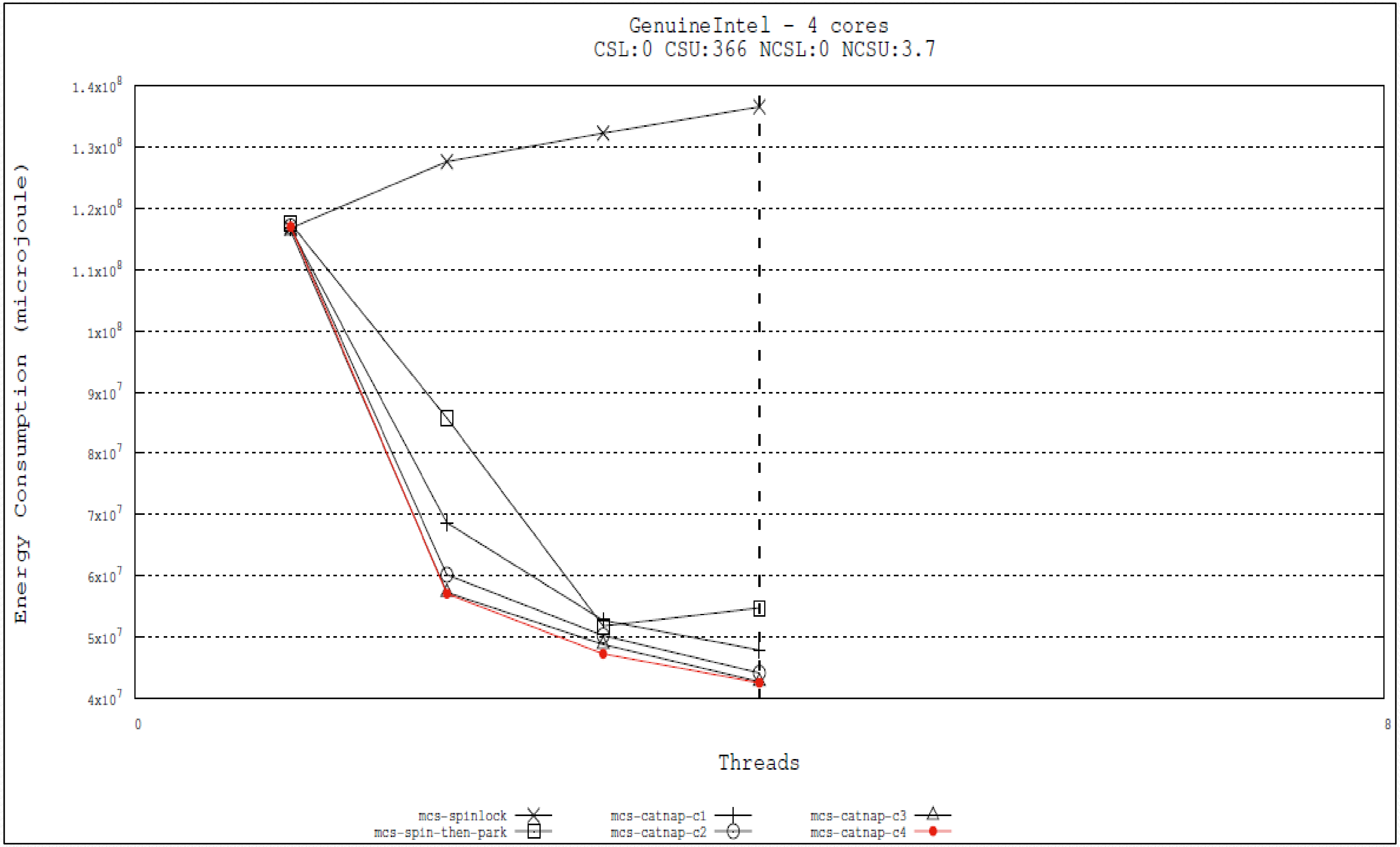
For each cluster, from left to right, the policies are: spinlock, spin_then_park, catnap_c1, c2, c3, and c4

White: user lvl
Black: kernel lvl

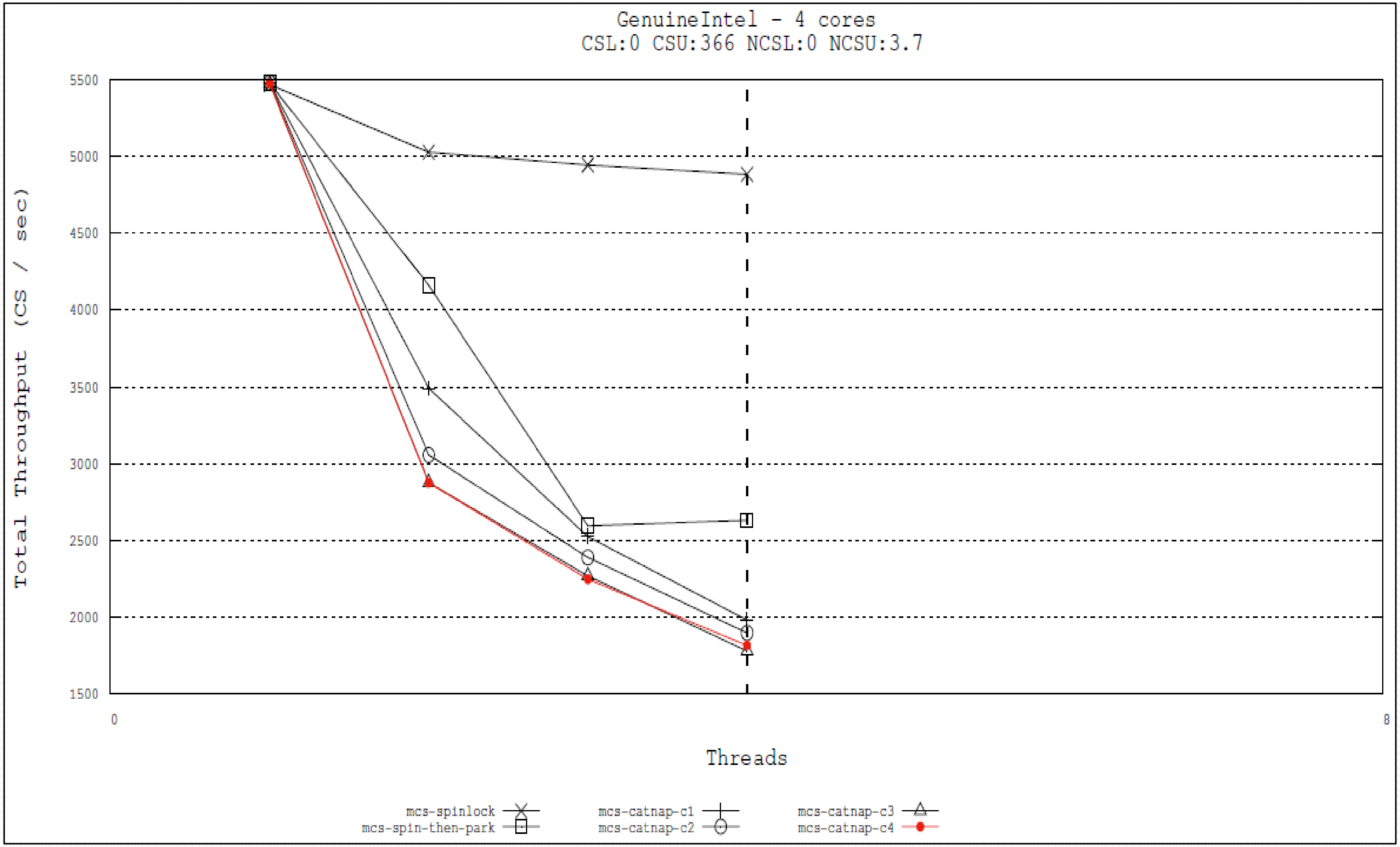
The CPU usage is bottom to top: 10%, 20%, 30% and 40%



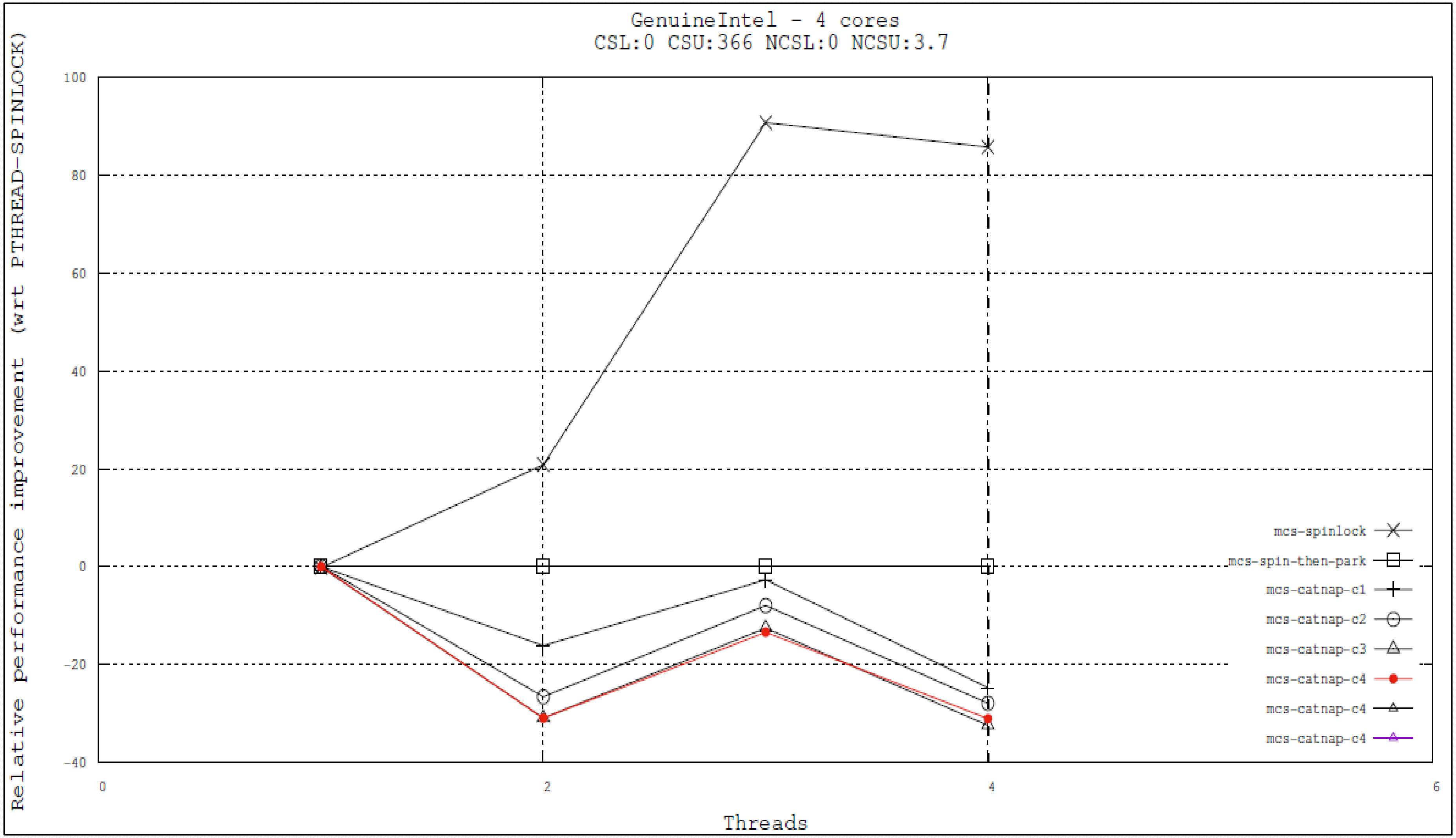
# ENERGY CONSUMPTION



# TOTAL THROUGHPUT



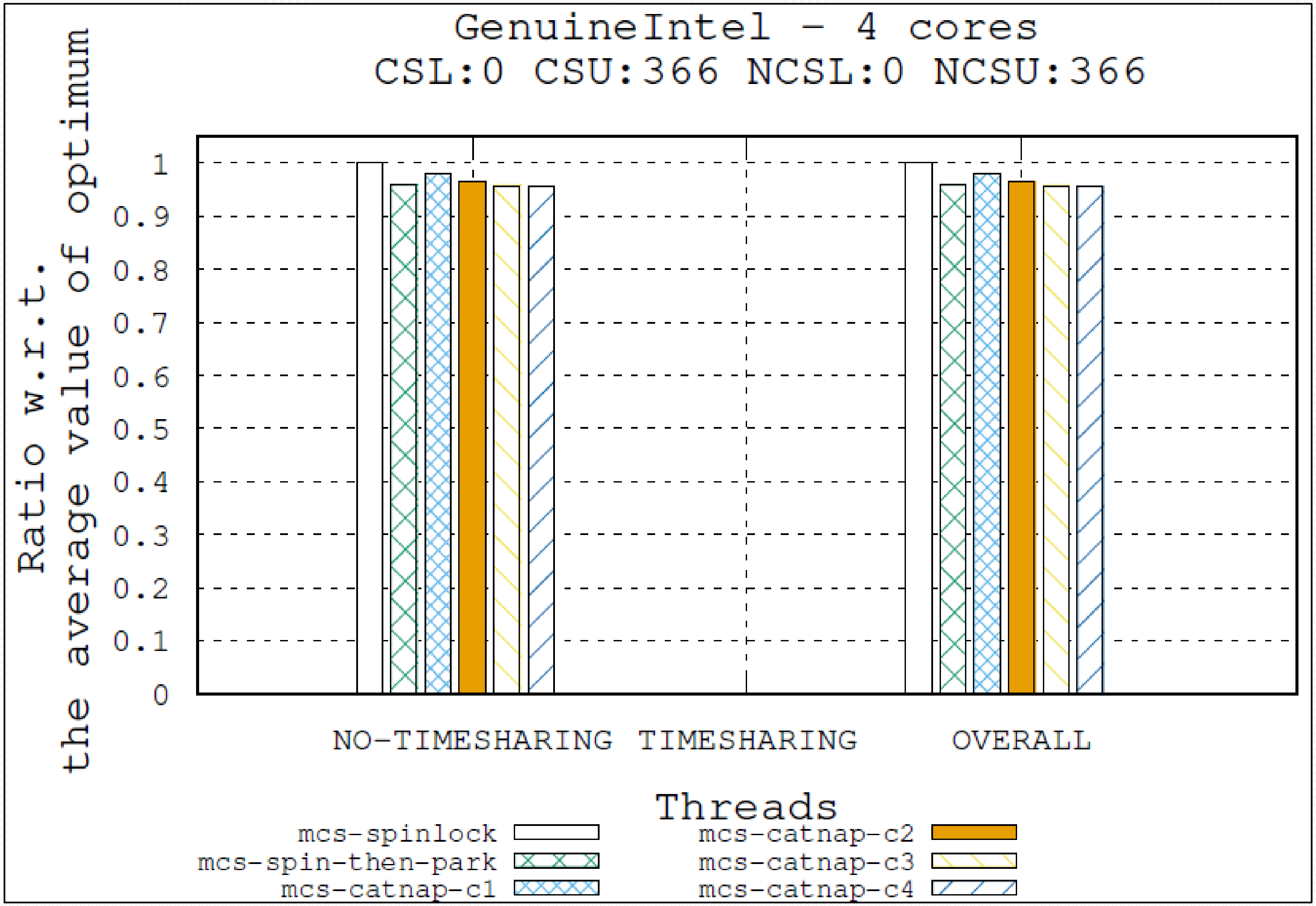
# RELATIVE IMPROVEMENT



# TEST 4

## CS AND NCS ARE BOTH LONG

# AVERAGE OF OPTIMUM VALUE

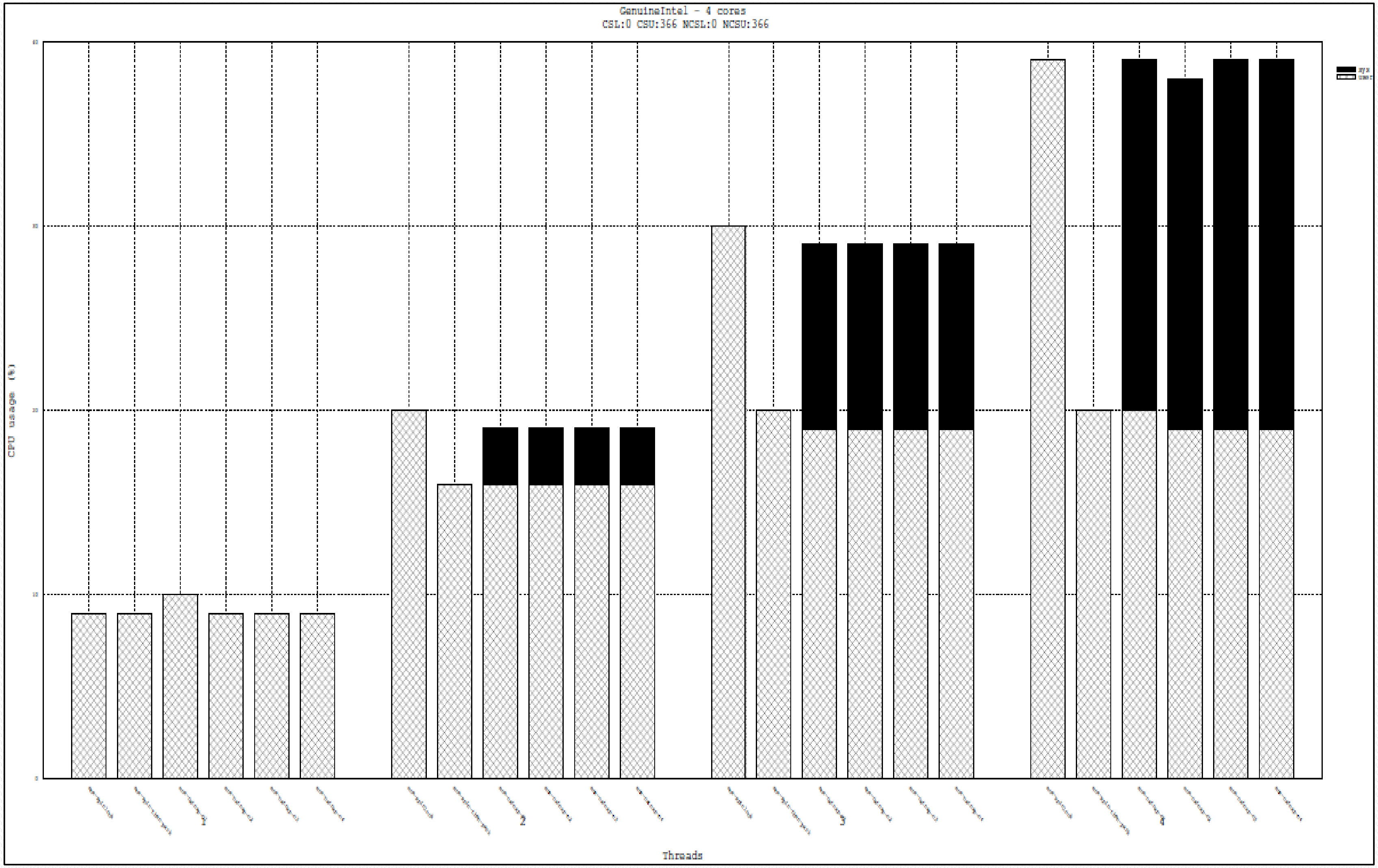


# CPU USAGE

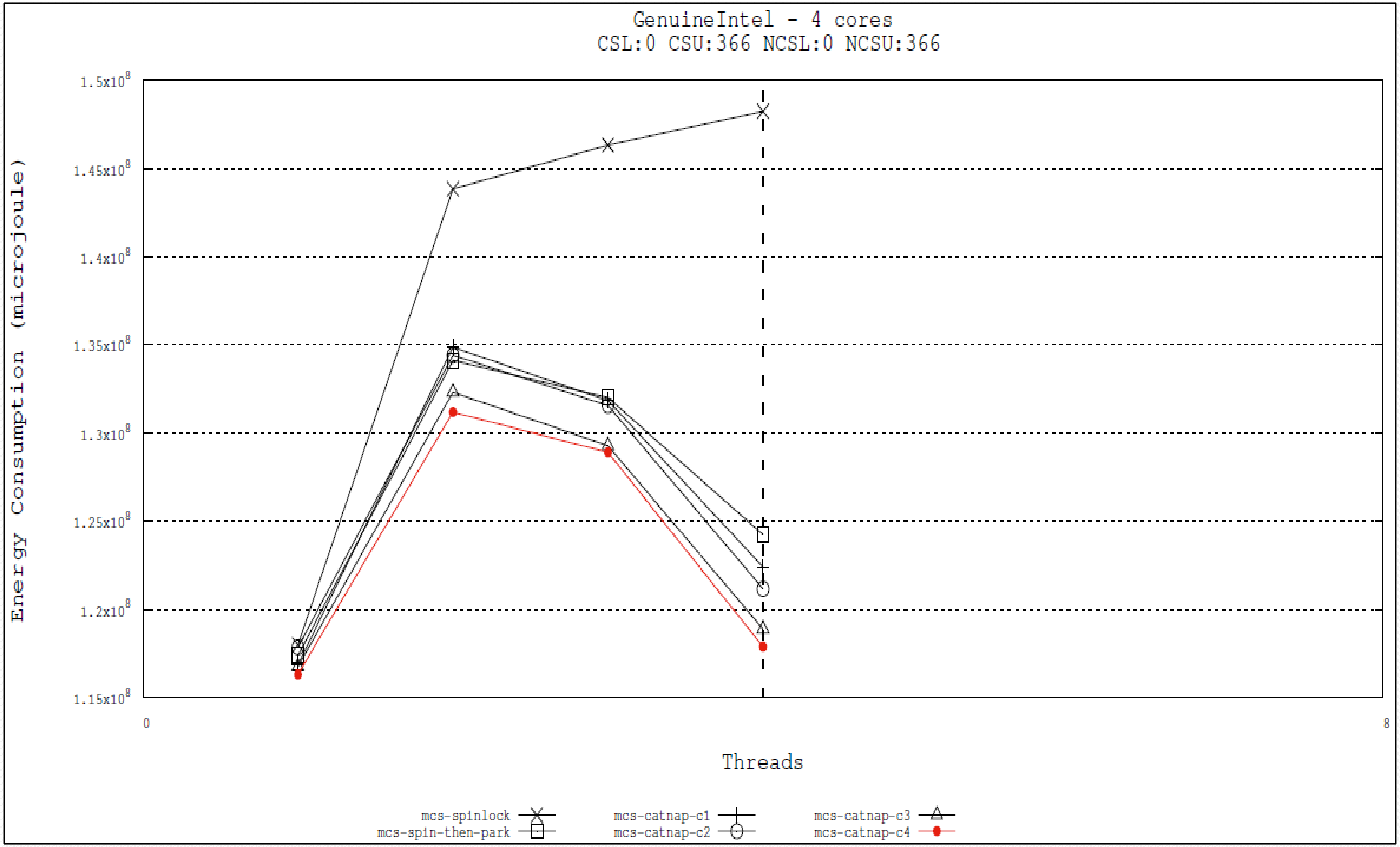
For each cluster, from left to right, the policies are: spinlock, spin_then_park, catnap_c1, c2, c3, and c4

White: user lvl
Black: kernel lvl

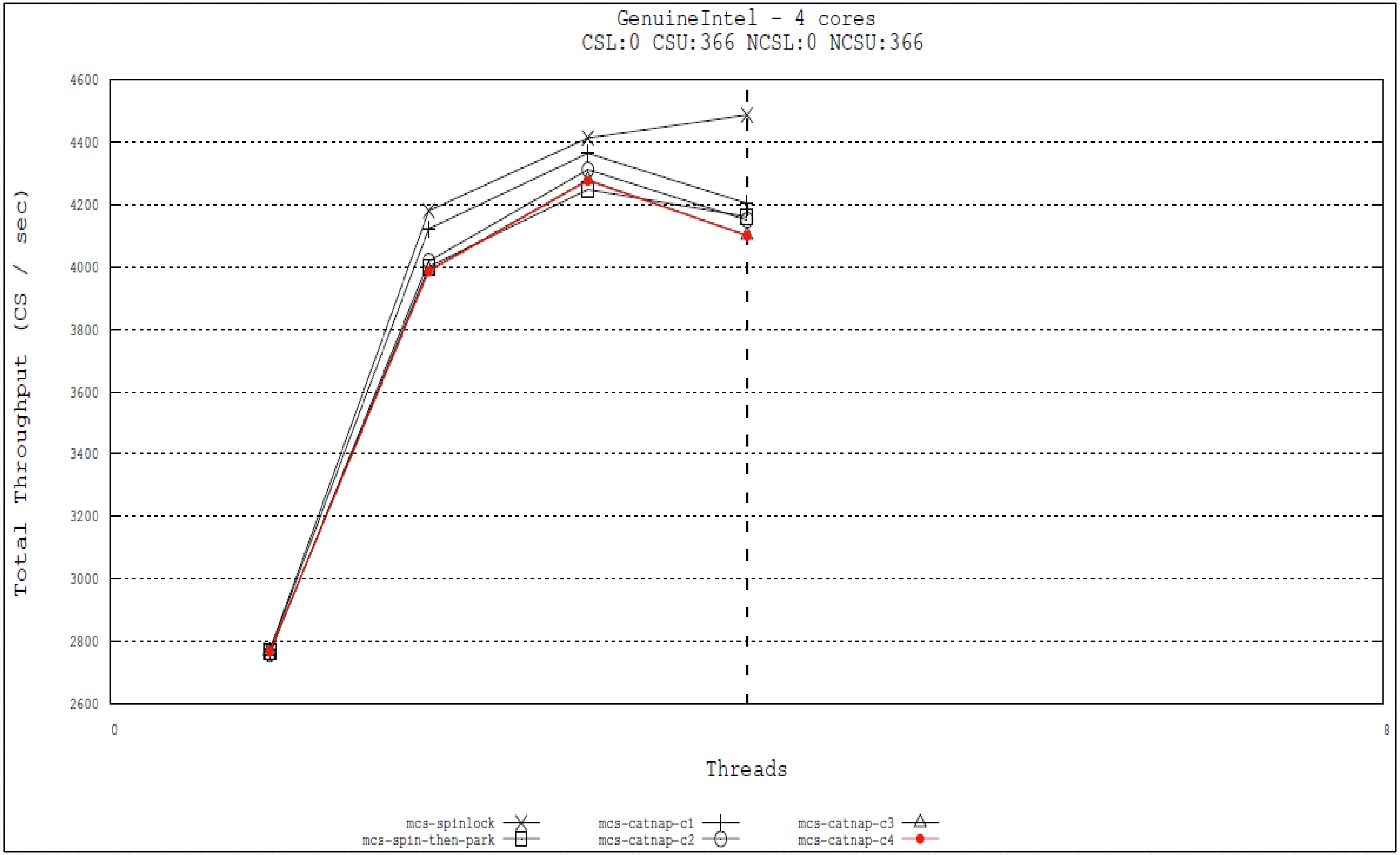
The CPU usage is bottom to top: 10%, 20%, 30% and 40%



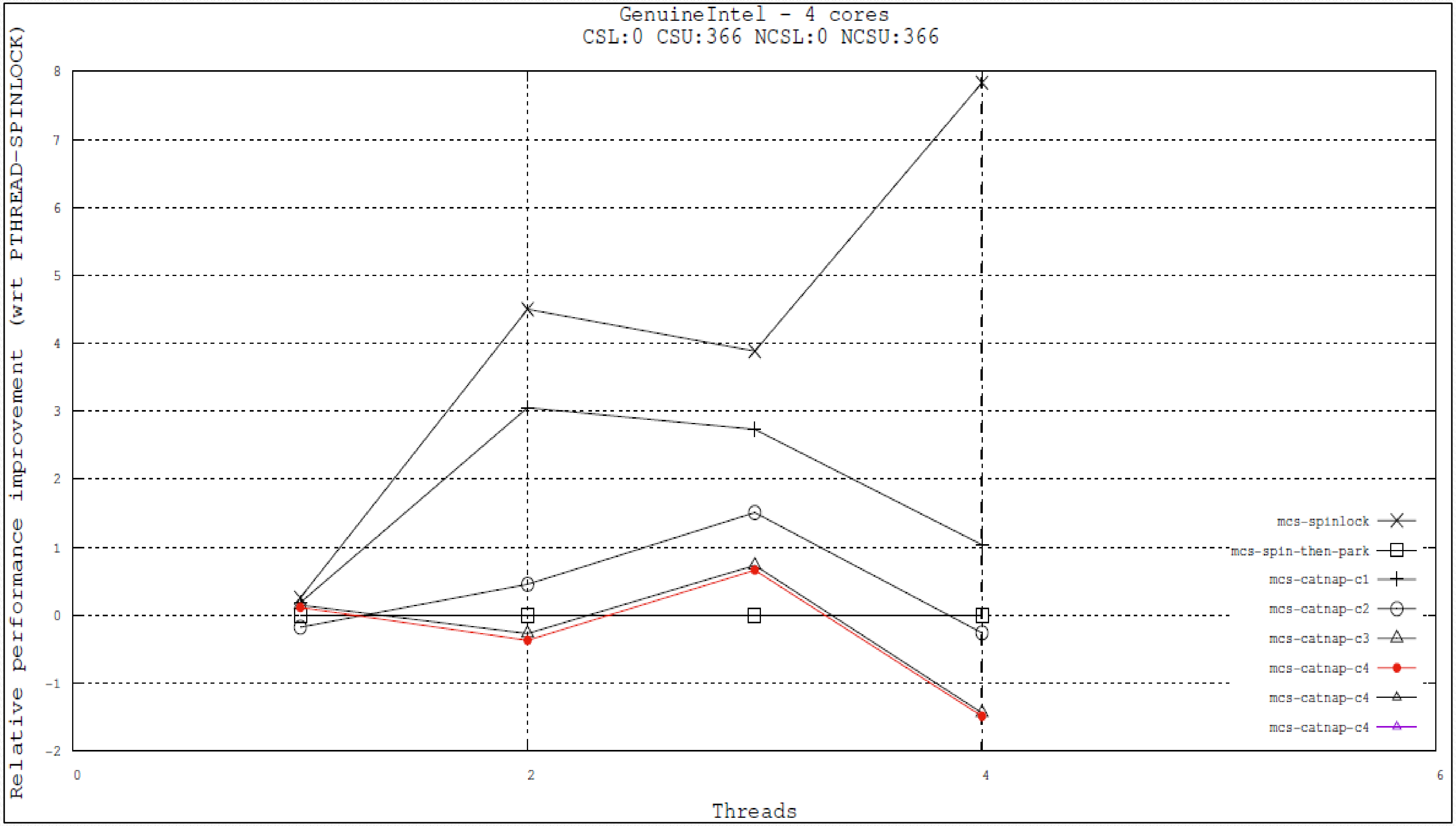
# ENERGY CONSUMPTION



# TOTAL THROUGHPUT



# RELATIVE IMPROVEMENT



# FINAL REMARKS

# FINAL REMARKS - PART 1

- The CPU usage in the Catnap tests is comparable to spinlock and spin-then-park because from the system point of view, the cores are still seen as normally working, despite their optimized status when ‘mwaiting’ in kernel mode(remember that we use Catnap through a system call!)
- But from the energy consumption point of view, we can see there’s a big save of resources using Catnap, even of an entire order of magnitude when using 4 threads in tests 1 and 3.

# FINAL REMARKS - PART 2

- Sadly there’s also a drop of throughput when using Catnap. These losses appear to be directly proportional to the energy consumption.
- The good news is that the gains in the ‘energy consumption’ department can be more impactful then the drops of perfomance. For example, in test 3 when using 4 threads the throughput is halved, but the energy consumed is 10 times less!

# FINAL REMARKS - PART 3

- The other good news is that the drop of performance of Catnap when increasing the number of threads is expected. This is because catnap is meant to be used in a data structure where the threads go from the MWAIT phase to spinning, and then from spinning to working. It takes some time for a core to return to its normal status, and without the spinning phase, there's nothing that hides this loss of time.

# FINAL REMARKS - PART 4

- This means that when used in precise contexts, (i.e. with the right size for the cs and appropriate data stuctures), Catnap can save a lot of resources, while still being performant, as promised in the paper itself.
- NOTE: There's no info about timesharing because MSC follows a FIFO policy (i.e. there's no one else to share the resources with).
- NOTE: The differences between all the various versions of Catnap (each targetting a different C-State) were less noticeable than expected.