computer systems performance
lecture 2
memory hierarchy & CPU parallelism

Pınar Tözün

February 4, 2026

borrows some slides / inspiration from
A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, I. Psaroudakis
How to Stop Under-Utilization and Love Multicores.
ICDE 2015 Tutorial

agenda

lecture

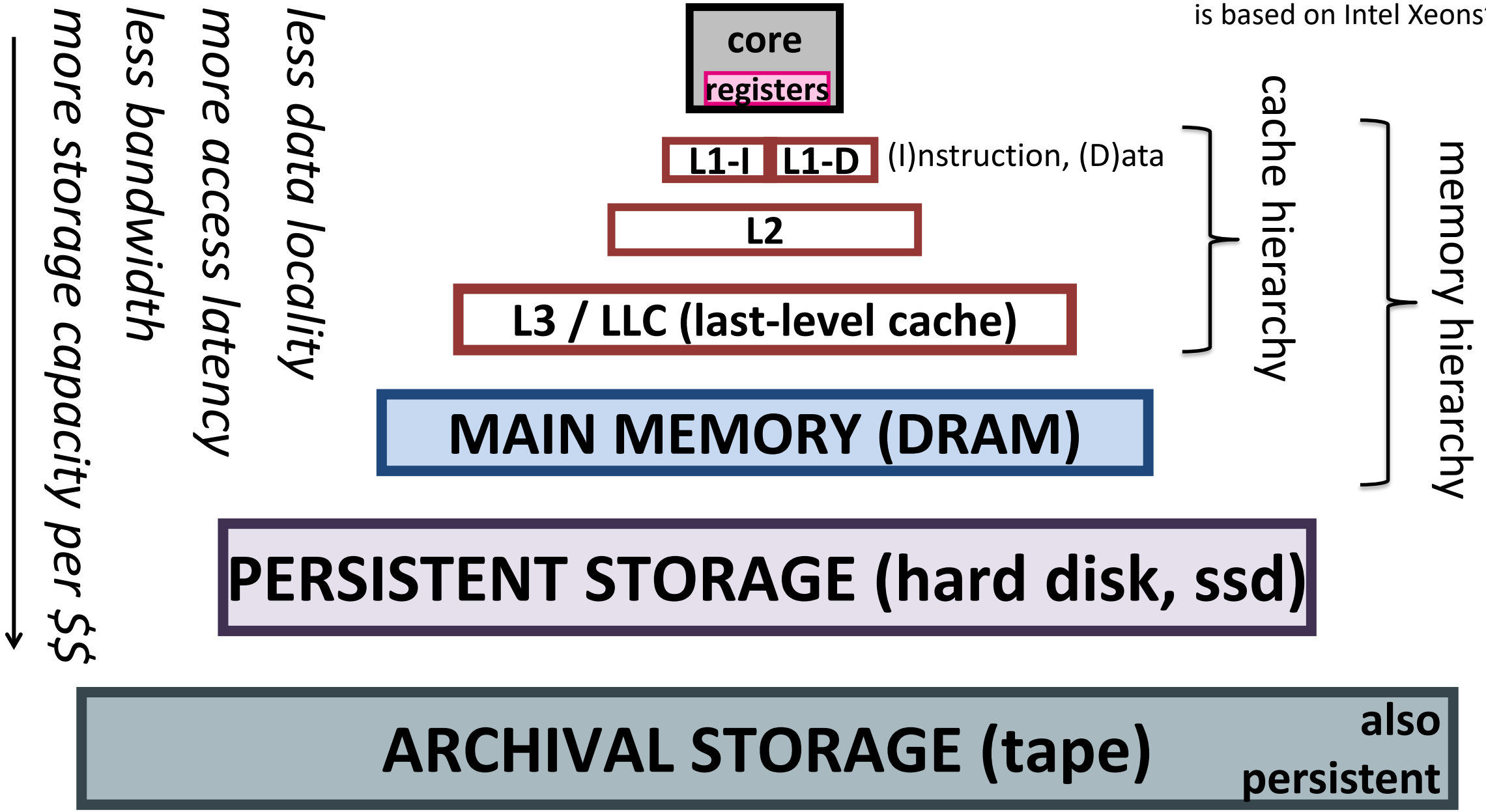
- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

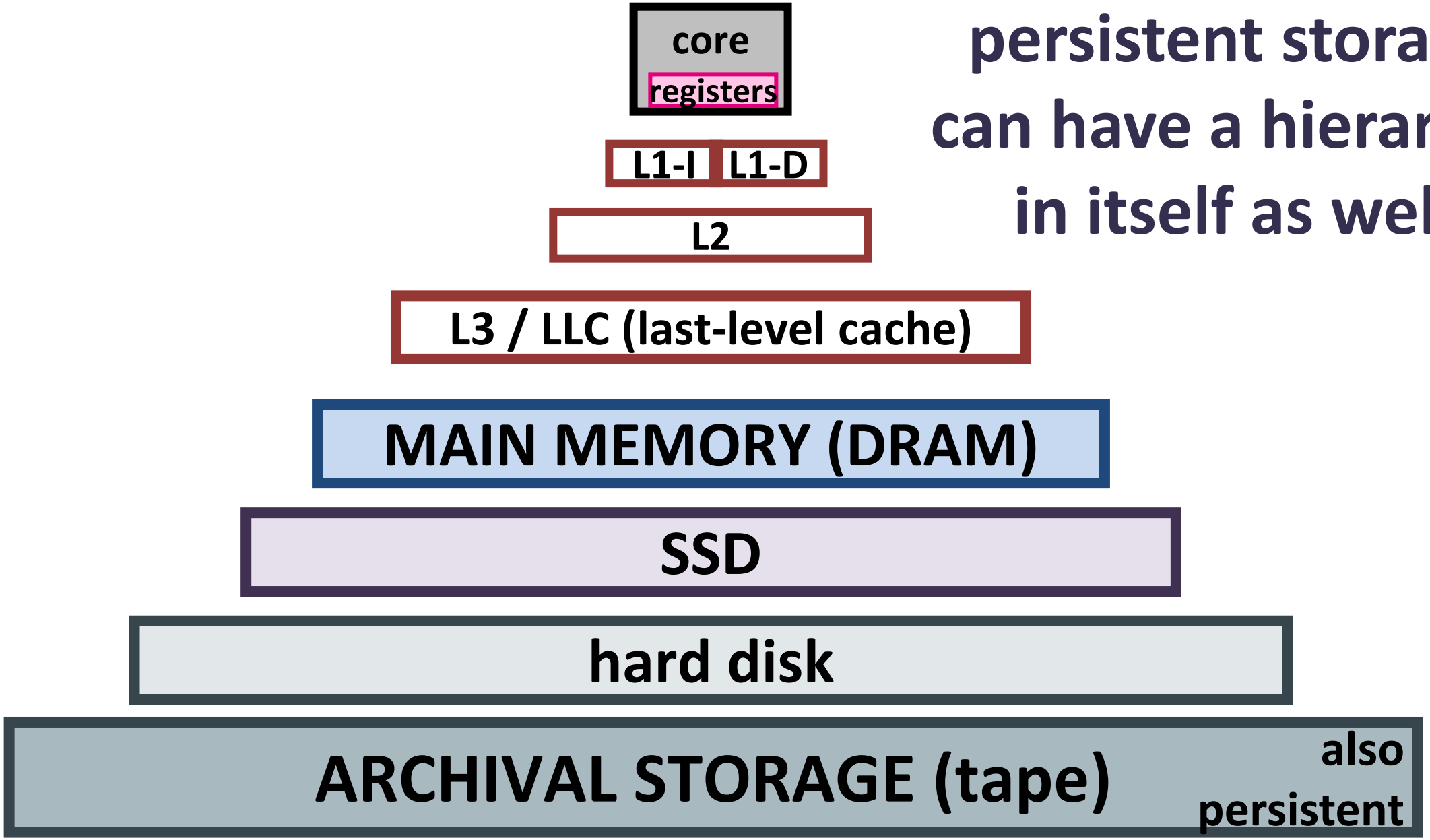
- paper discussion
- c/cpp examples
- setting core affinity

(typical) storage hierarchy

disclaimer: memory hierarchy is based on Intel Xeons' here



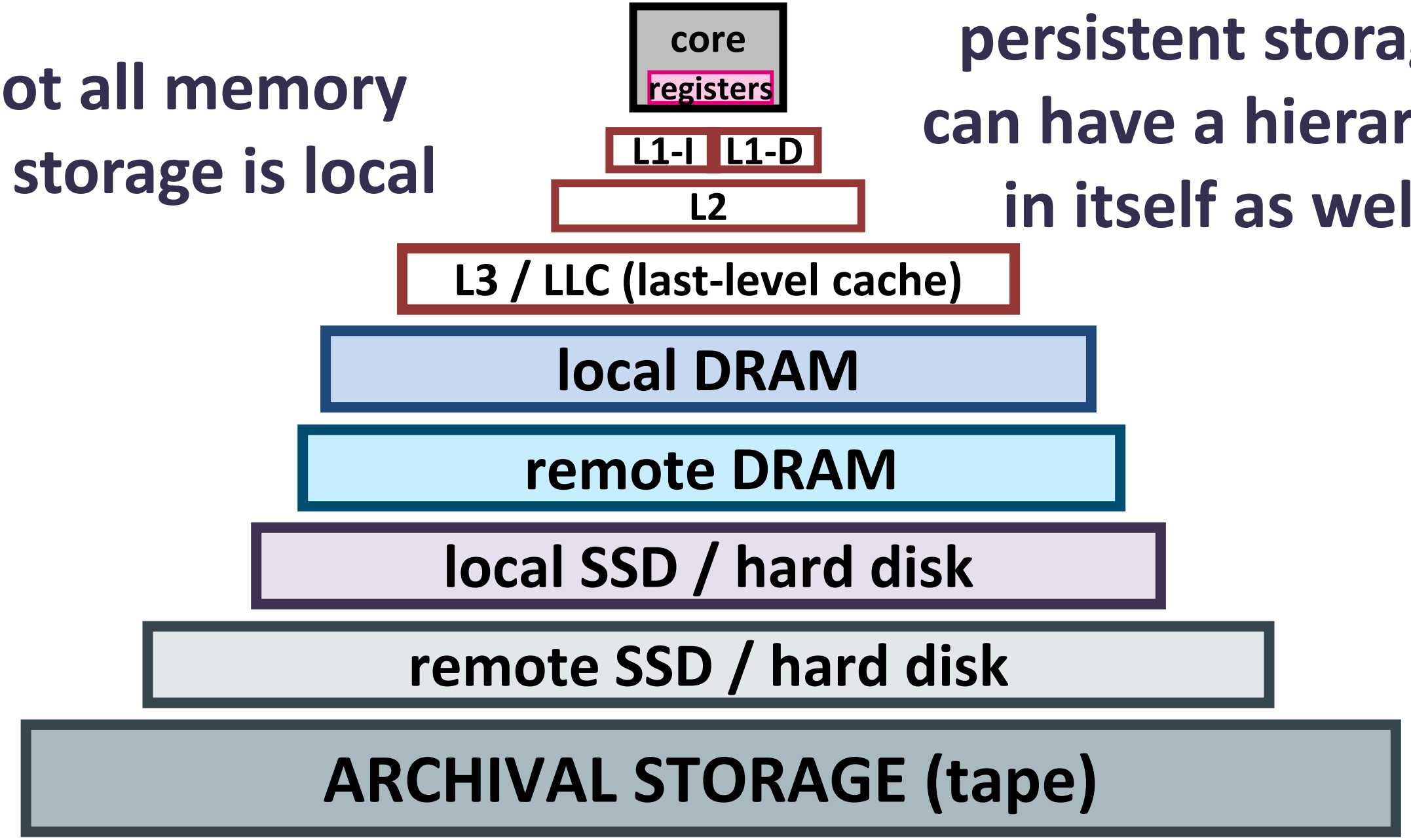
(typical) storage hierarchy



persistent storage can have a hierarchy in itself as well

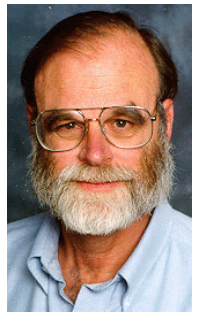
(typical) storage hierarchy

not all memory or storage is local

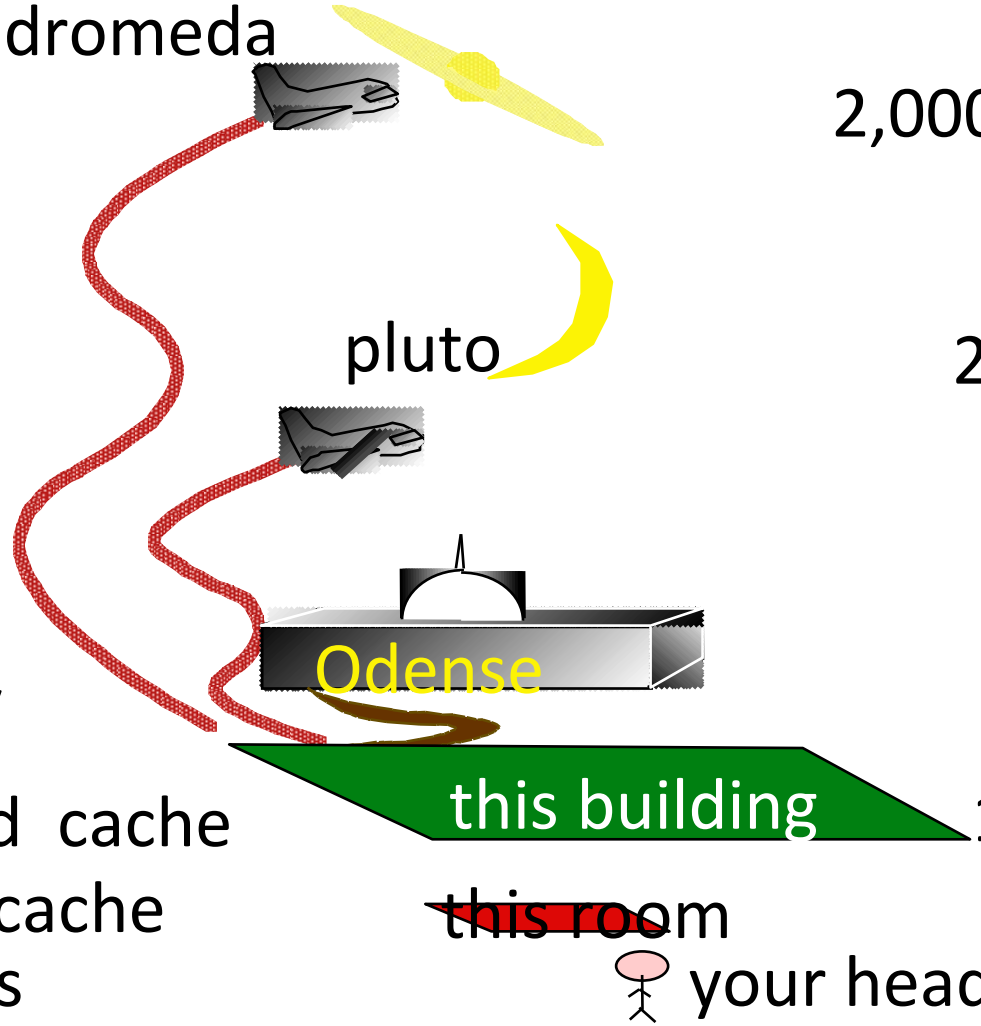


persistent storage can have a hierarchy in itself as well

Jim Gray's storage latency analogy



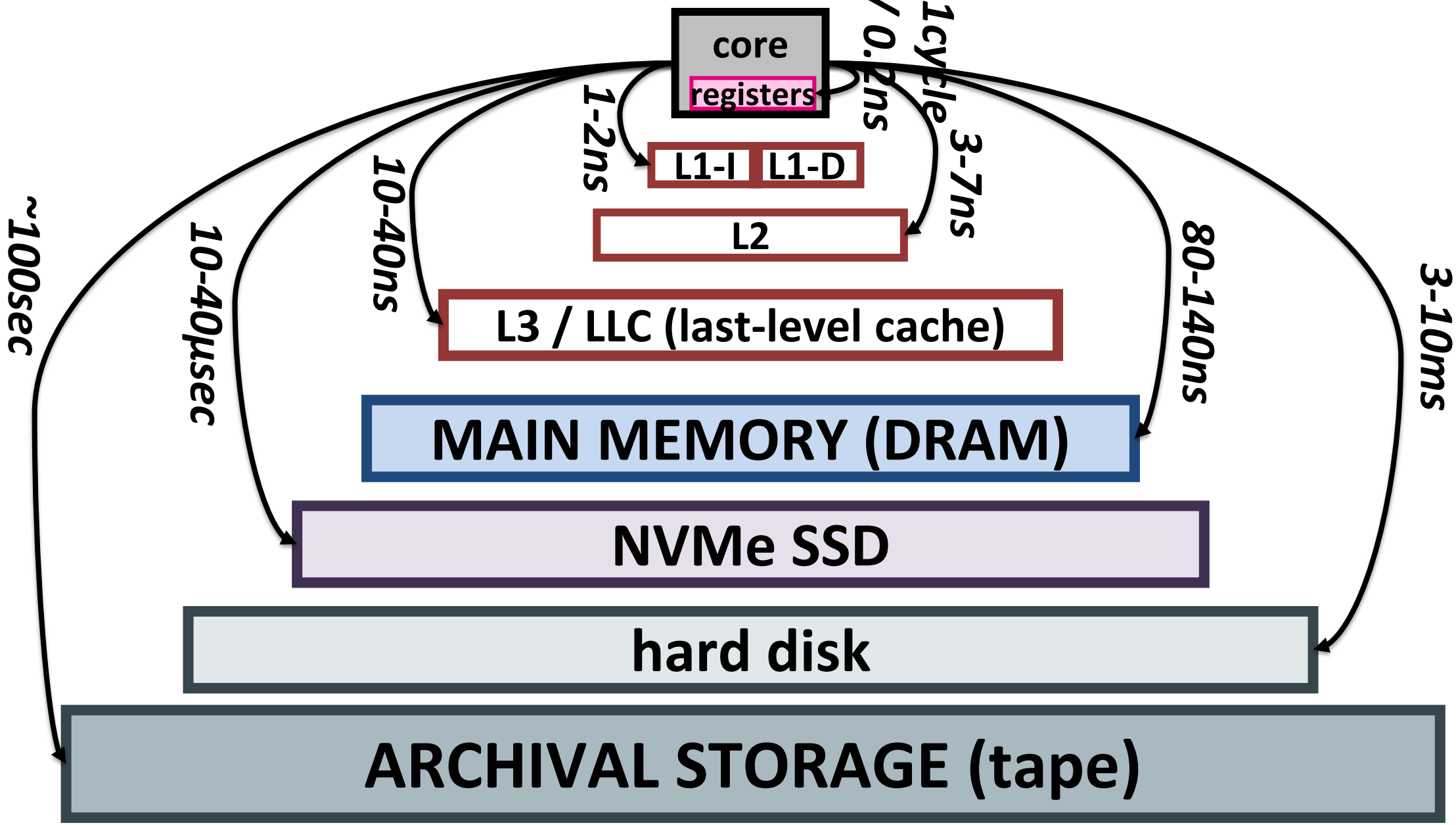
how far away is the data?



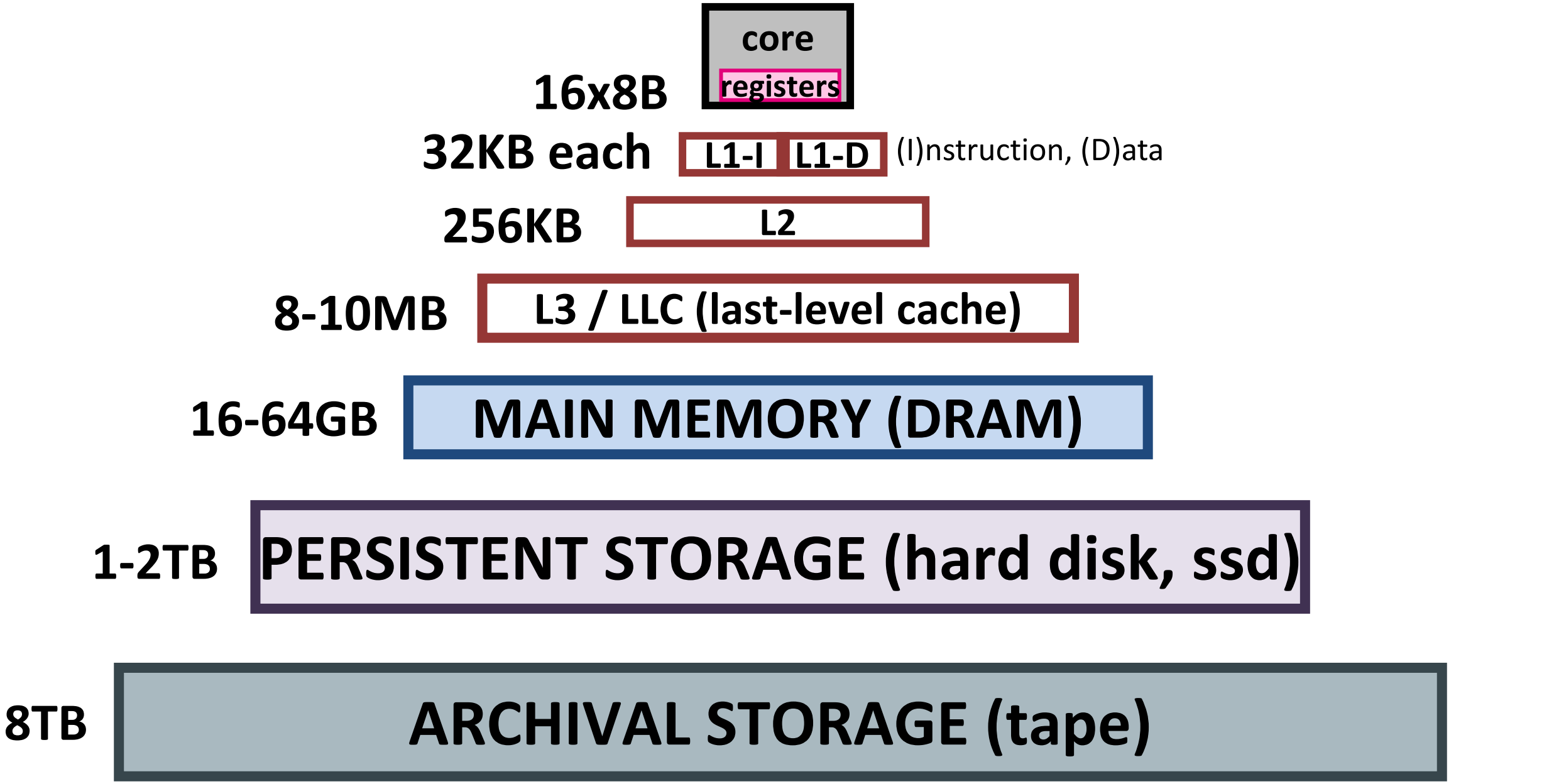
10^9	tape	2,000 Years
10^6	disk	2 Years
100	memory	1.5 hr
10	on board cache	10 min
2	on chip cache	
1	registers	

https://en.wikipedia.org/wiki/Jim_Gray_(computer_scientist)

(typical) storage hierarchy – access latency



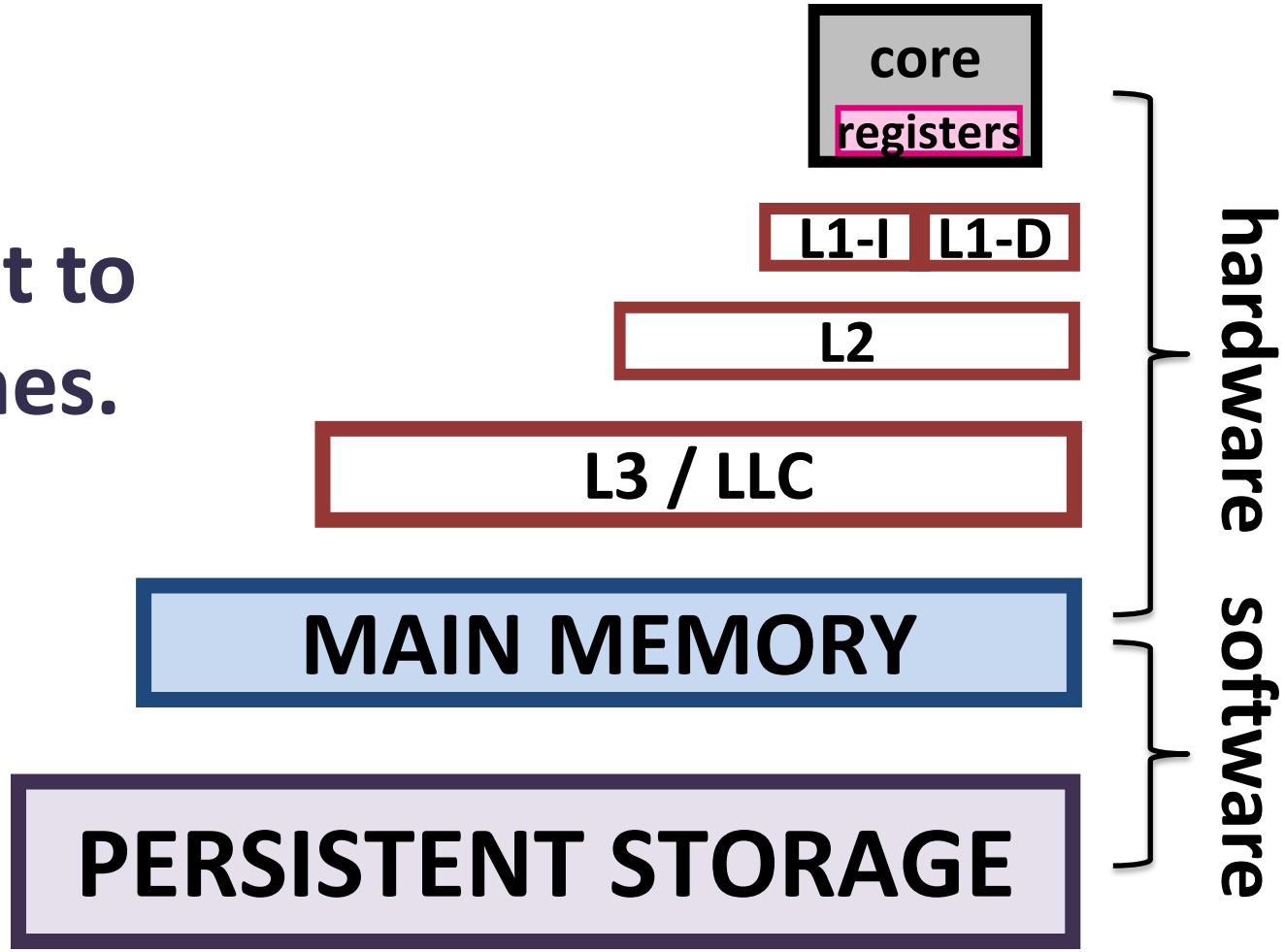
(typical) storage hierarchy – capacity



(typical) storage hierarchy – capacity

means that we do not write code to explicitly manage data movement to registers, L1, L2, L3 caches.

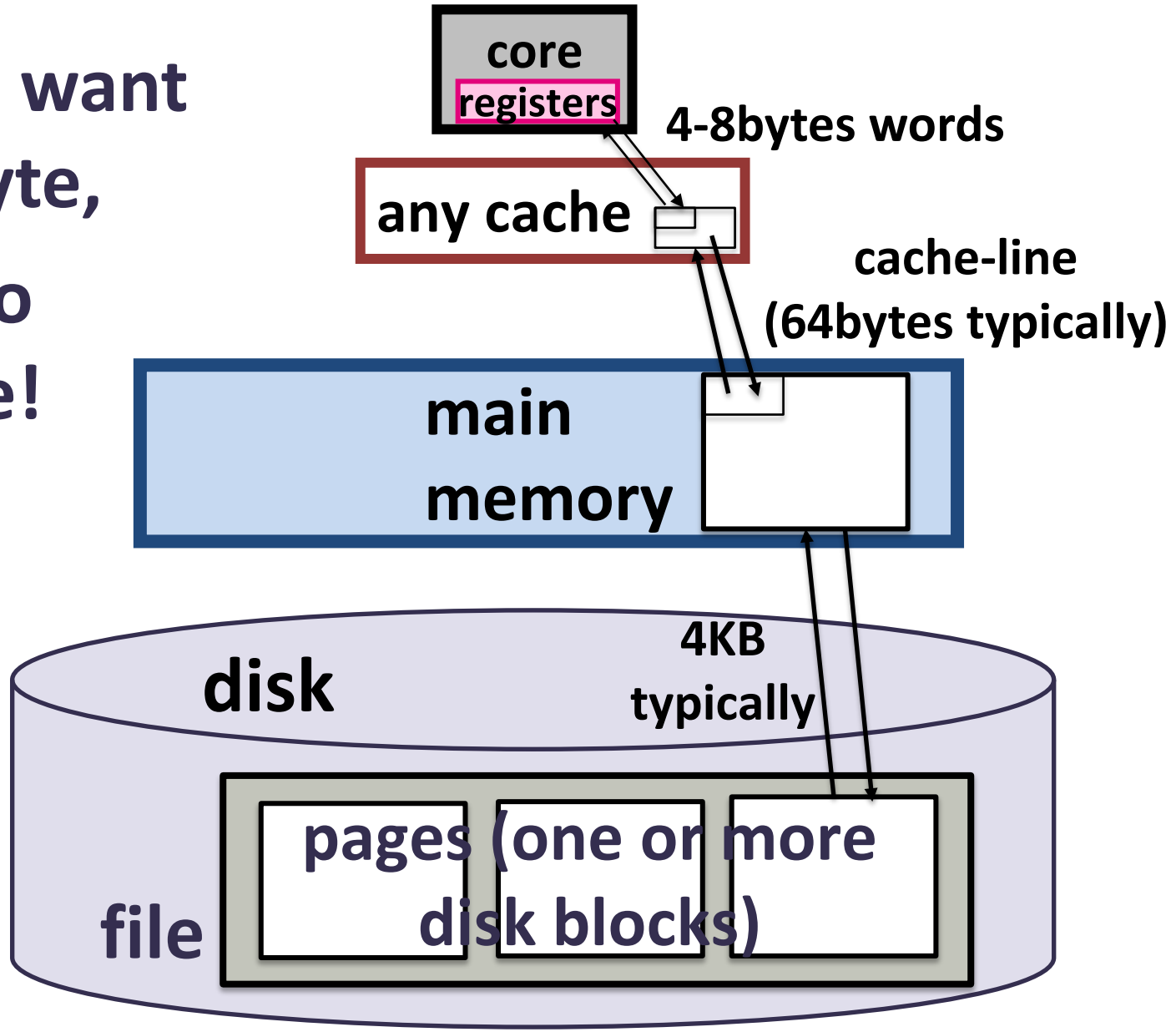
but we do this for moving data from/to persistent storage.



movement of data in storage hierarchy

even if you want to read 1byte,

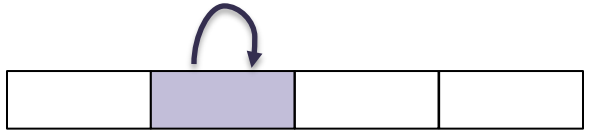
you need to move more!



principle of locality

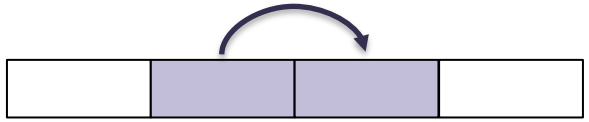
programs tend to use data & instructions with addresses near or equal to those they have used recently

temporal locality



recently referenced items are likely to be referenced again in near future

spatial locality



items with nearby addresses tend to be referenced close together in time

impacts design decisions for both software & hardware

principle of locality – example

```
for (int i = 0; i < array.size(); i++) {
   sum += array[i];
}
```

data

- **temporal** → referencing *sum* in each iteration
- **spatial** → iterating over array element

instructions

- **temporal** → looping over the same code
- **spatial** → executing instructions in sequence

caching

goal is to increase locality for cores

to reduce access latency for frequently accessed data

higher levels cache data from lower levels

inclusivity

- most hardware vendors build inclusive cache hierarchy
- software controlled caching can be more complex
  e.g., some main-memory optimized database systems don't persist indexes

data replacement when no space left

replacement policy can be a crucial optimization

dealing with modified data

write through – immediately or
write back – at replacement time

agenda

lecture

- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

- paper discussion
- c/cpp examples
- setting core affinity

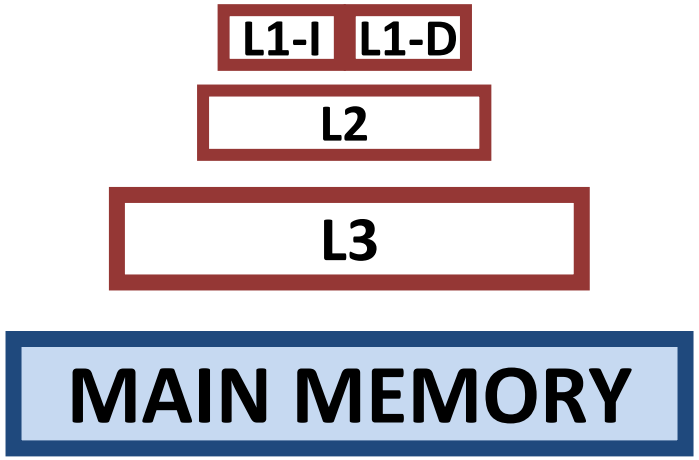
random-access memory (RAM)

(almost) ***constant random-access latency*** wherever the data is

volatile ➔ will loose data once power is lost

sequential access is slightly faster
because of ***prefetching***:

if you fetch a block from main-memory to caches,
hardware usually prefetches the adjacent block



D(ynamic)RAM vs S(tatic)RAM

- DRAM is most common for *main memory*, SRAM is used for *caches & registers*
- DRAM requires refresh, otherwise looses data even when power is on
- SRAM is more energy-efficient, but also more expensive,
  so DRAM is preferred for main memory

caches

consider cache like a 2-dimentional array

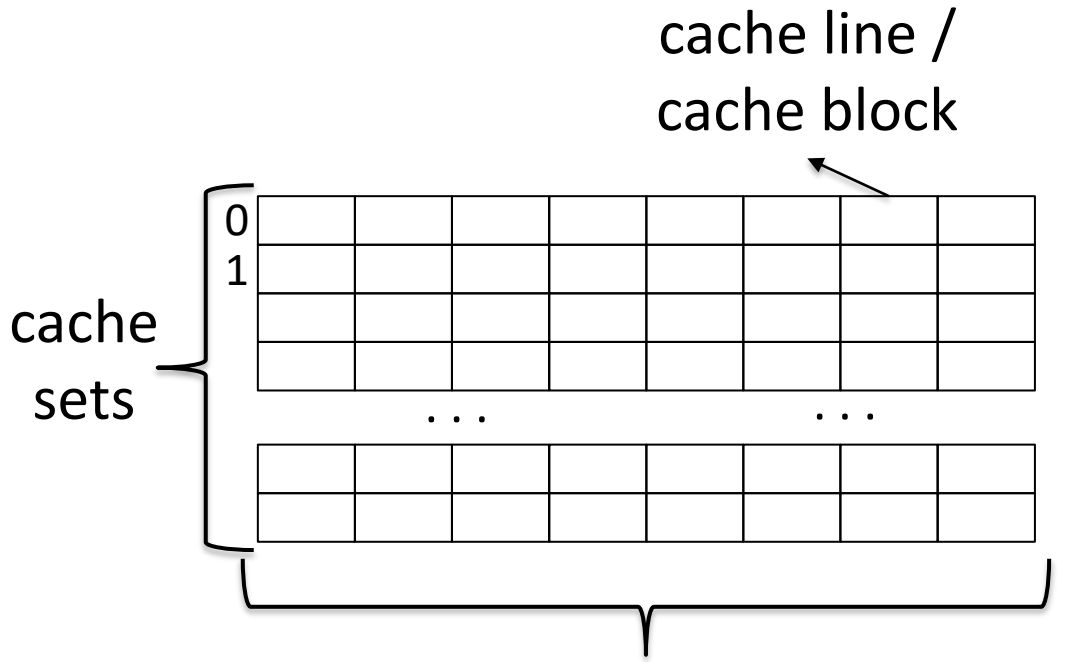
example:
32KB 8-way L1 cache
64bytes cache lines
64bit memory addresses

32 * 1024 / 64 = 512 cache lines
512 / 8 = 64 cache sets
64bit address (*a*) decomposition for cache ➔
TAG = 52 bits | SET = 6 bits | OFFSET = 6 bits

to determine if *a* is cache hit/miss

which set to look at

which byte to read in cache line



cache misses

… when an item is not in the cache

compulsory misses

misses that occur because cache is empty

conflict misses

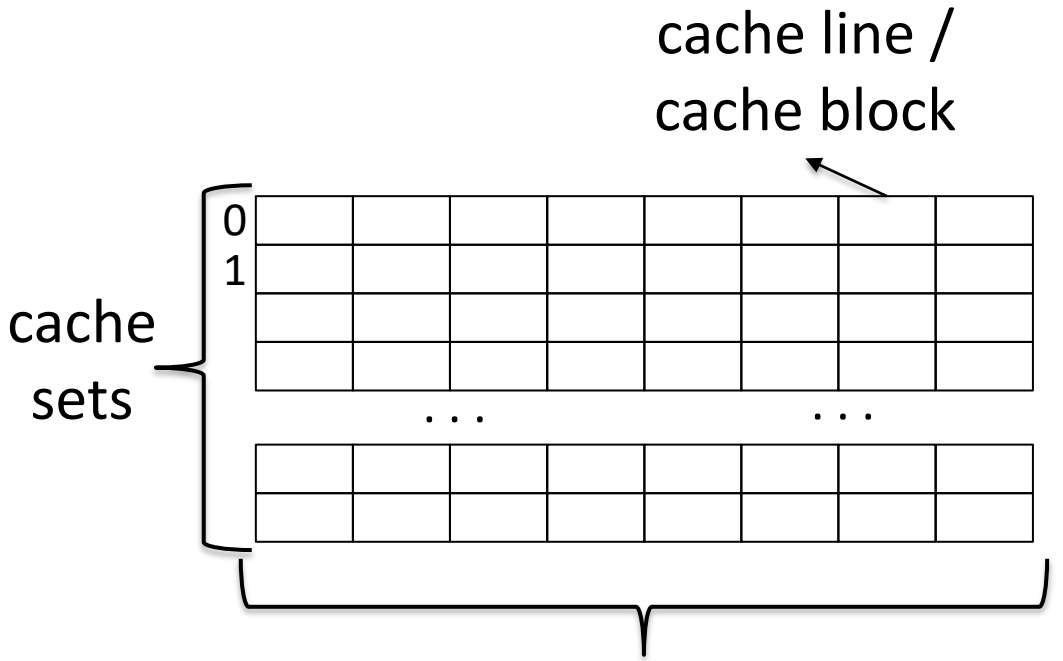
misses that occur due to a cache set being full even though cache has space

→ higher with lower associativity

capacity misses

misses due to cache being full

→ common when active data/instruction set size bigger than total cache size



hardware prefetching

a = address

- ***next-line*** prefetching: miss a → fetch a+1
- ***stream*** prefetching: miss a, a+1 → fetch a+2, a+3

✓ favors sequential access & spatial locality

✘ instructions: branches, function calls

- branch prediction handles this to some extend

✘ data: pointer chasing – common for index accesses

- ***stride*** prefetching handles this to some extend
- miss a, a+20 → fetch a+40, a+60

prefetchers on real hardware are kept simple
better accuracy requires more complexity

summary: memory hierarchy

Different layers of the hierarchy have different characteristics & require different optimizations from the software side.

Principle of ***caching*** is to improve locality for frequently accessed data.

Sequential access is faster even in main-memory, where random and sequential access latency is the same, because of hardware prefetching.

It is important to be aware of the hierarchy to utilize it more effectively & have more efficient software layers.

goal: minimize data access latency!

agenda

lecture

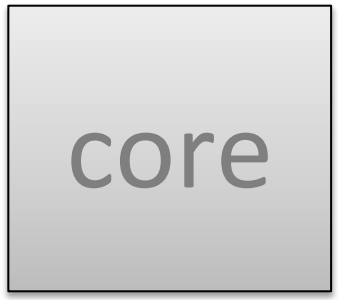
- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

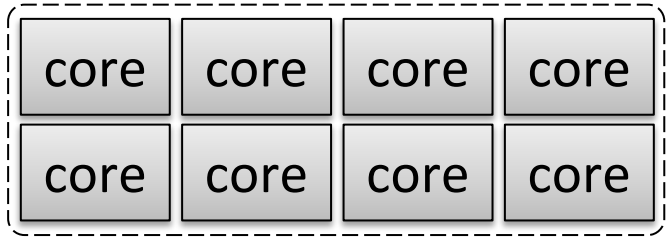
- paper discussion
- c/cpp examples
- setting core affinity

central processing unit (CPU) evolution

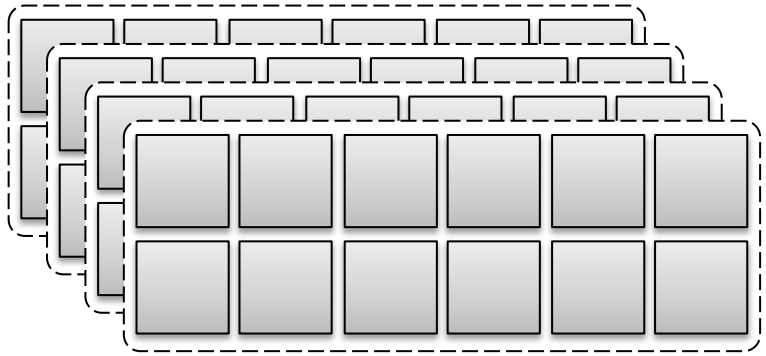
2005



single-core CPUs



multicore CPUs



multisocket multicore CPUs

faster & more-complex cores over time

similar speed & complexity in a core, more parallelism over time

wording courtesy of Wikipedia

for Moore's law to be practical you need Dennard scaling!

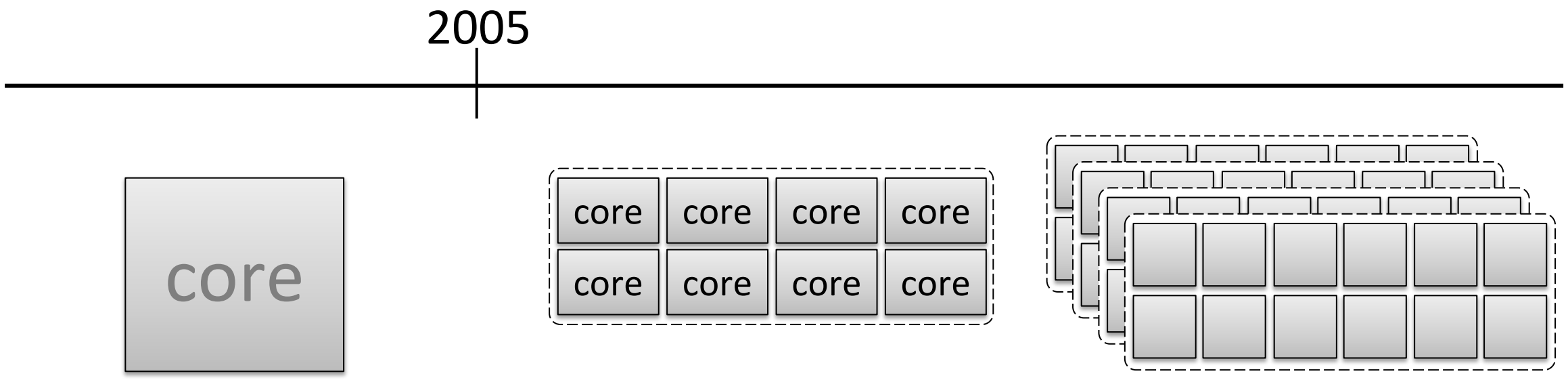
Moore's law

"… the observation that the number of transistors in a dense integrated circuit doubles approximately every two years."

Dennard scaling

" … as transistors get smaller their power density stays constant, so that the power use stays in proportion with area: both voltage and current scale (downward) with length."

central processing unit (CPU) evolution



single-core CPUs

multicore CPUs

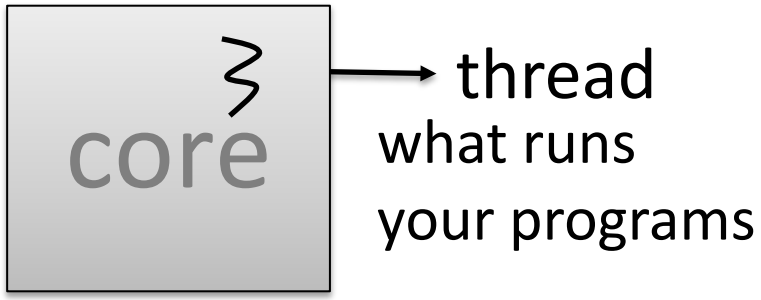
multisocket
multicore CPUs

around 2005, Dennard scaling stopped

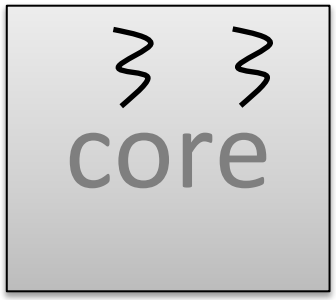
switching to multicores kept Moore's Law alive

types of hardware parallelism

implicit/vertical parallelism



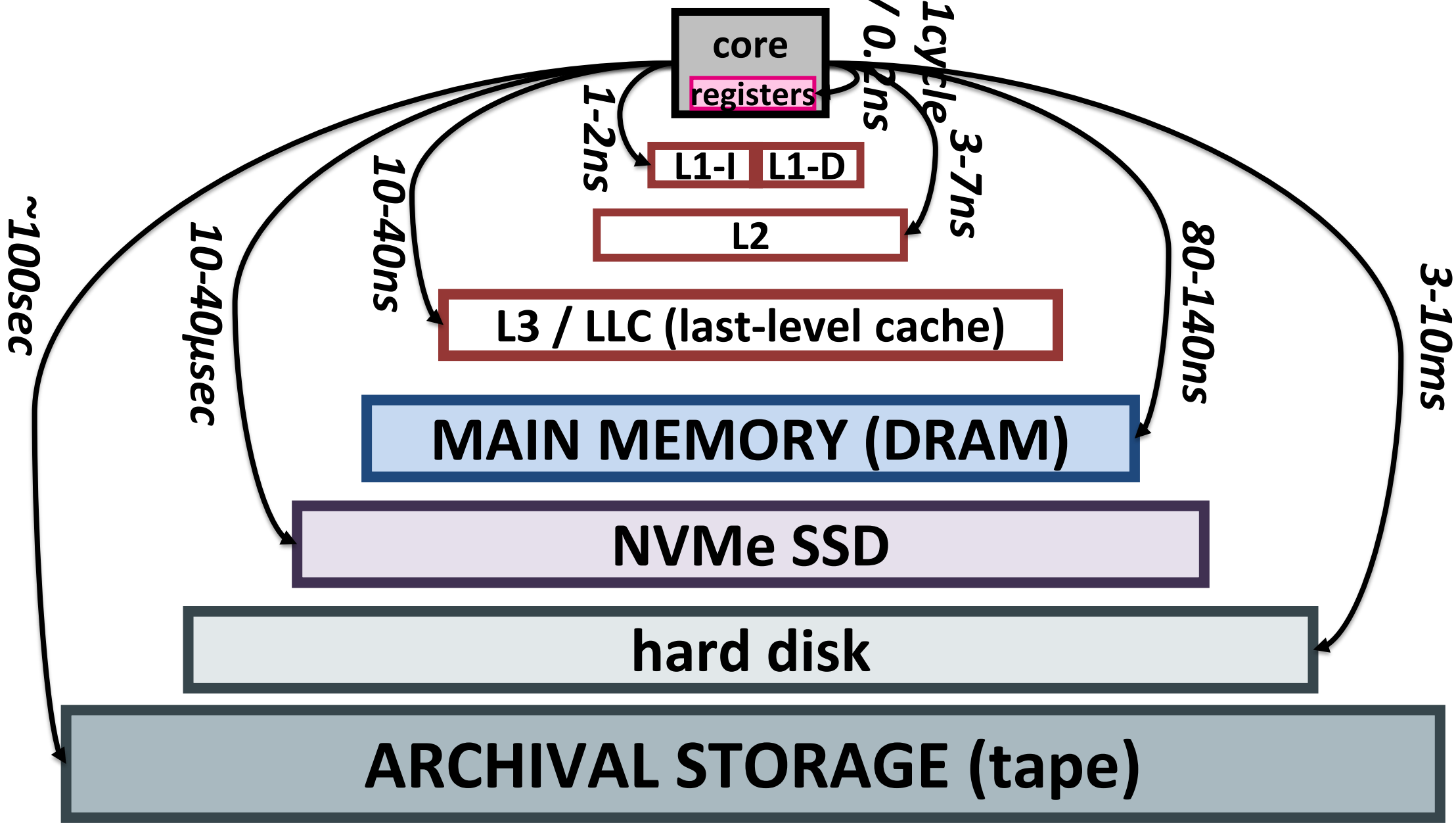
instruction & data parallelism
hardware does this automatically



multithreading
threads share
execution cycles
on the same core

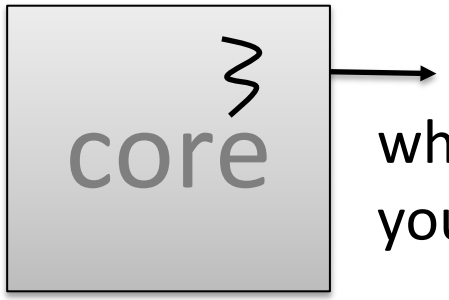
why do we need this?

(typical) storage hierarchy – access latency



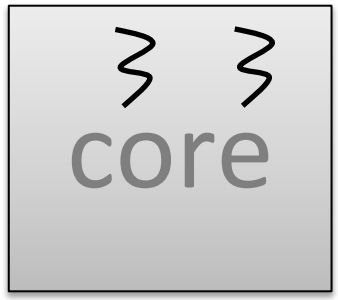
types of hardware parallelism

implicit/vertical parallelism



thread
what runs
your programs

instruction & data parallelism
hardware does this automatically

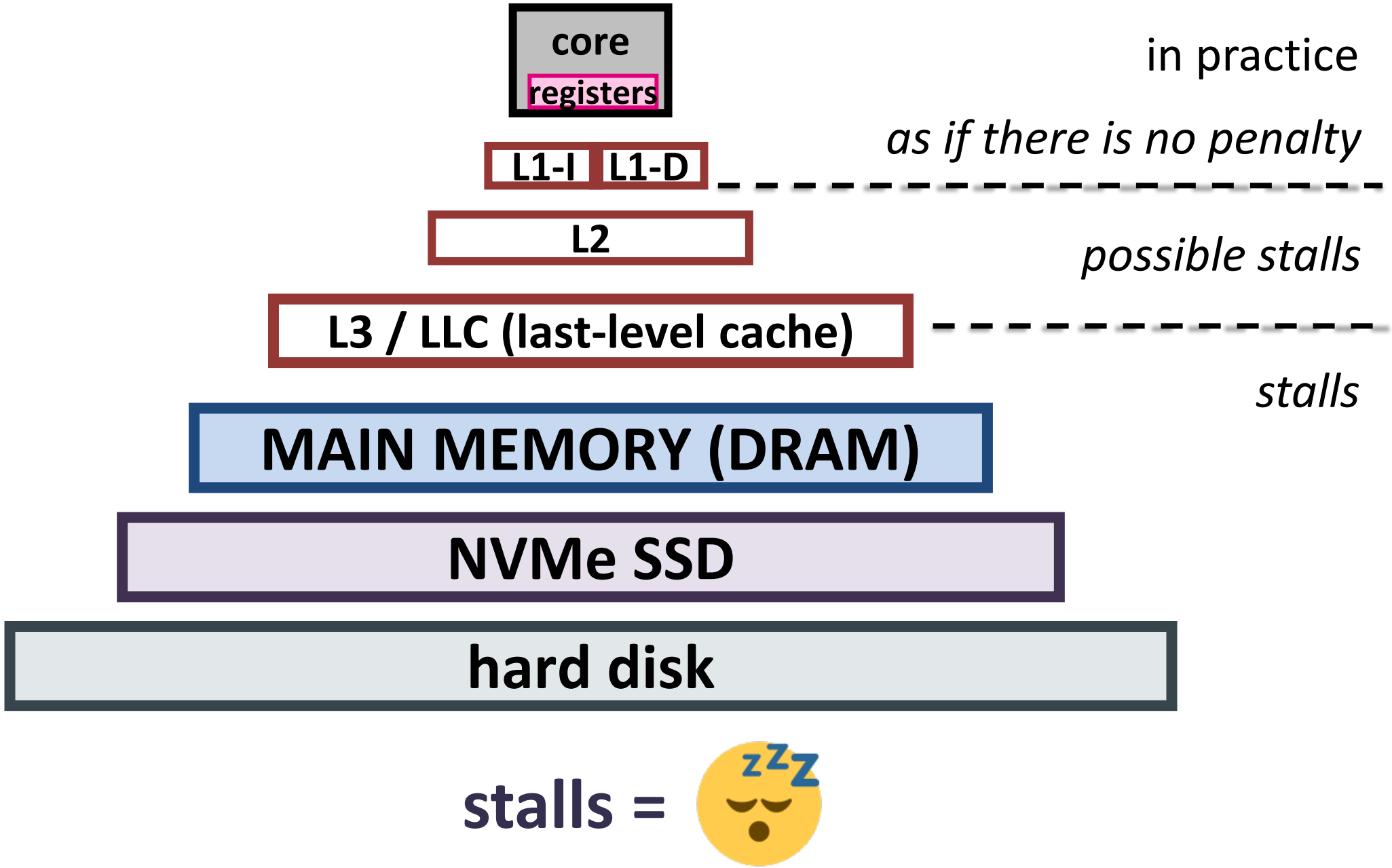


multithreading
threads share
execution cycles
on the same core

why?
we don't want cores to stay idle waiting for instruction/data accesses!

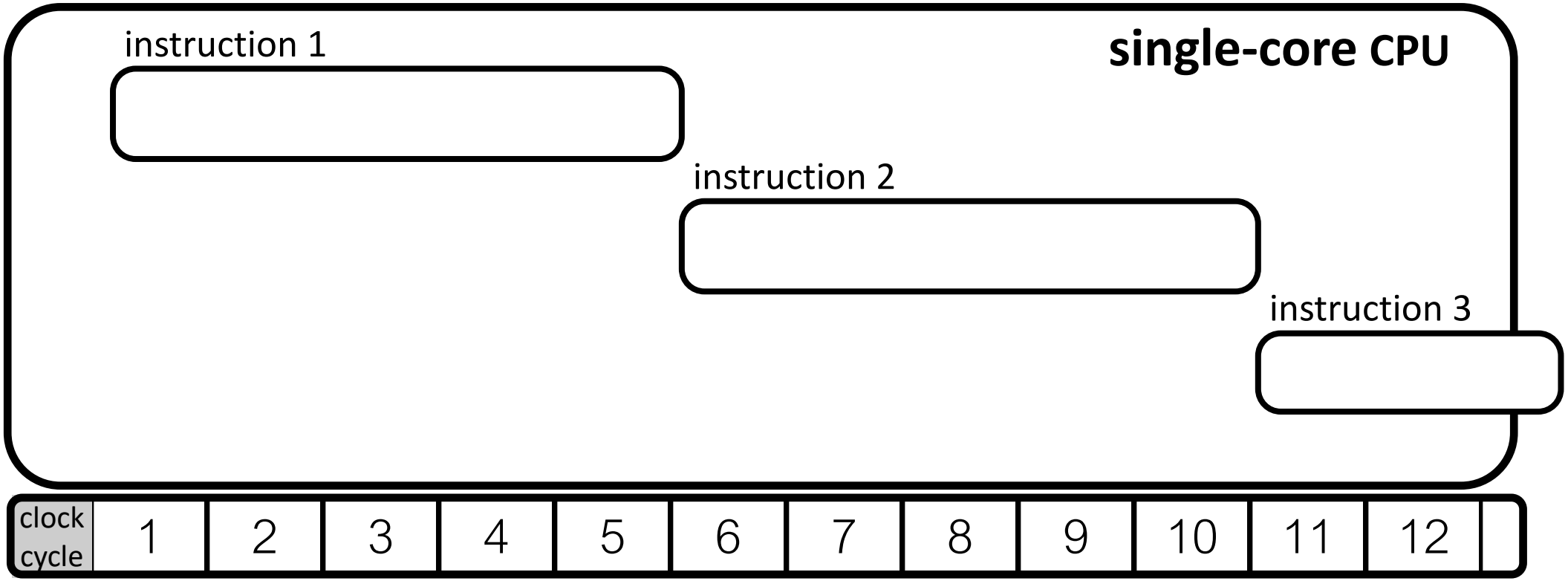
goal: minimize stall time due to cache/memory accesses overlapping access latency for one item with other work

(typical) storage hierarchy – access latency



subscalar CPU (i.e., no implicit parallelism)

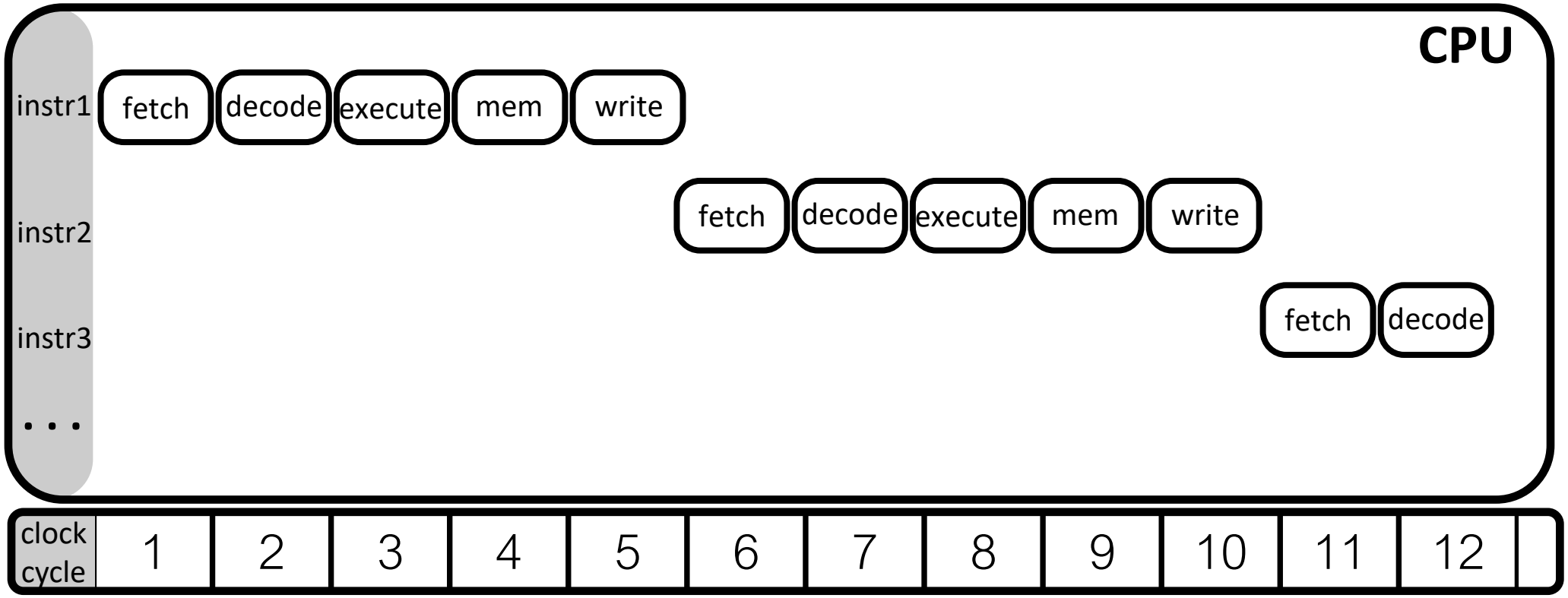
one instruction at a time



several cycles to complete two instructions, assuming no long-latency data/memory accesses

subscalar CPU (i.e., no implicit parallelism)

one instruction at a time



several cycles to complete two instructions, assuming no long-latency data/memory accesses

RISC instruction stages

fetch
the instruction from the cache

decode
specifying which operation the instruction performs
and inputs it needs

execute
the operation itself

memory
accessing the memory for inputs if needed

write
writing back the results into registers

micro-architecture of an OoO processor

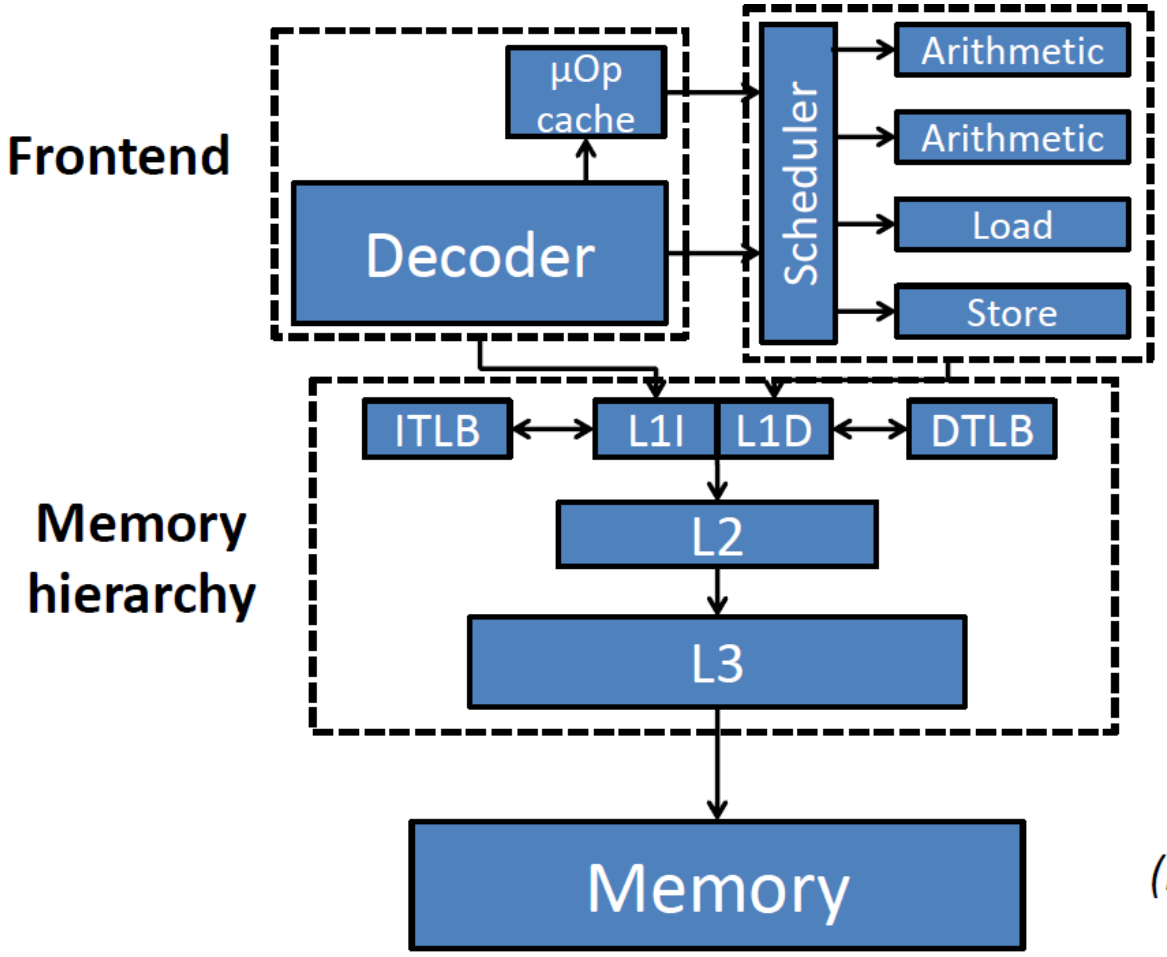


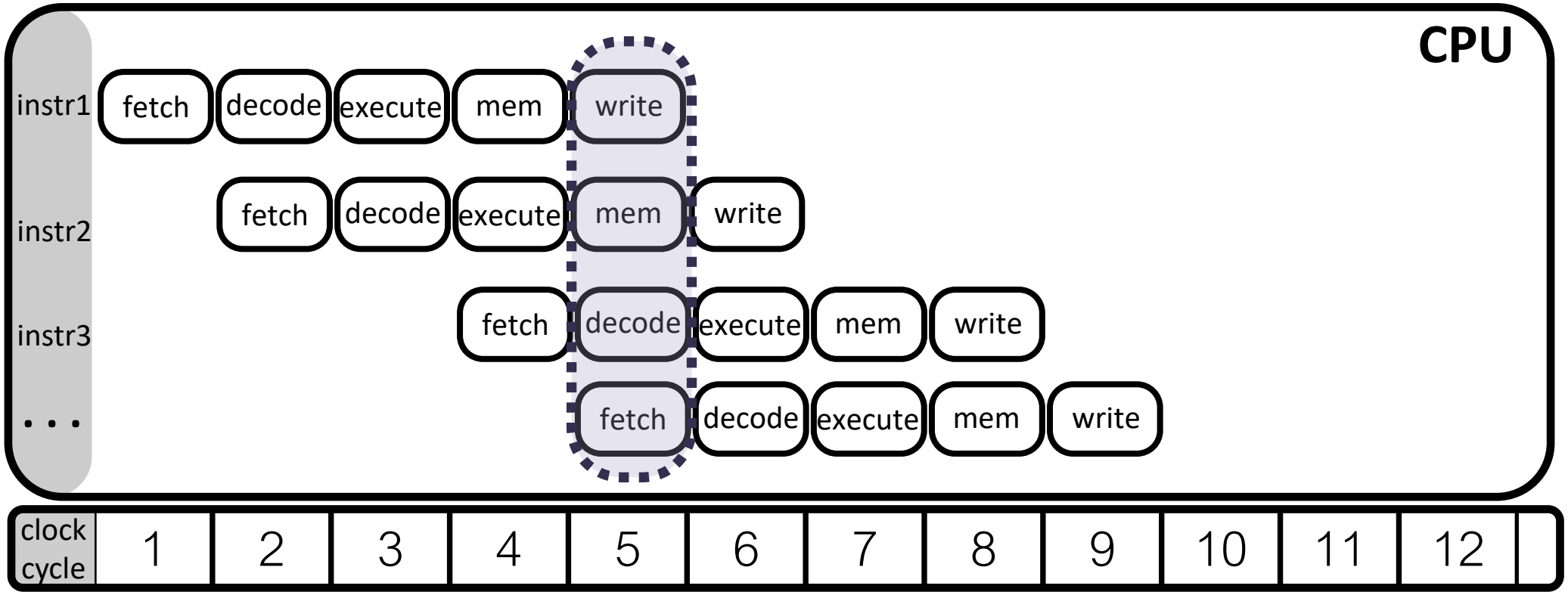
figure from Utku Sirin's DaMoN 2017 presentation

(Intel, SW opt. man., 2016)

delays in fetching, decoding, etc. an instruction cause *frontend stalls*, rest cause *backend stalls*

instruction pipelining

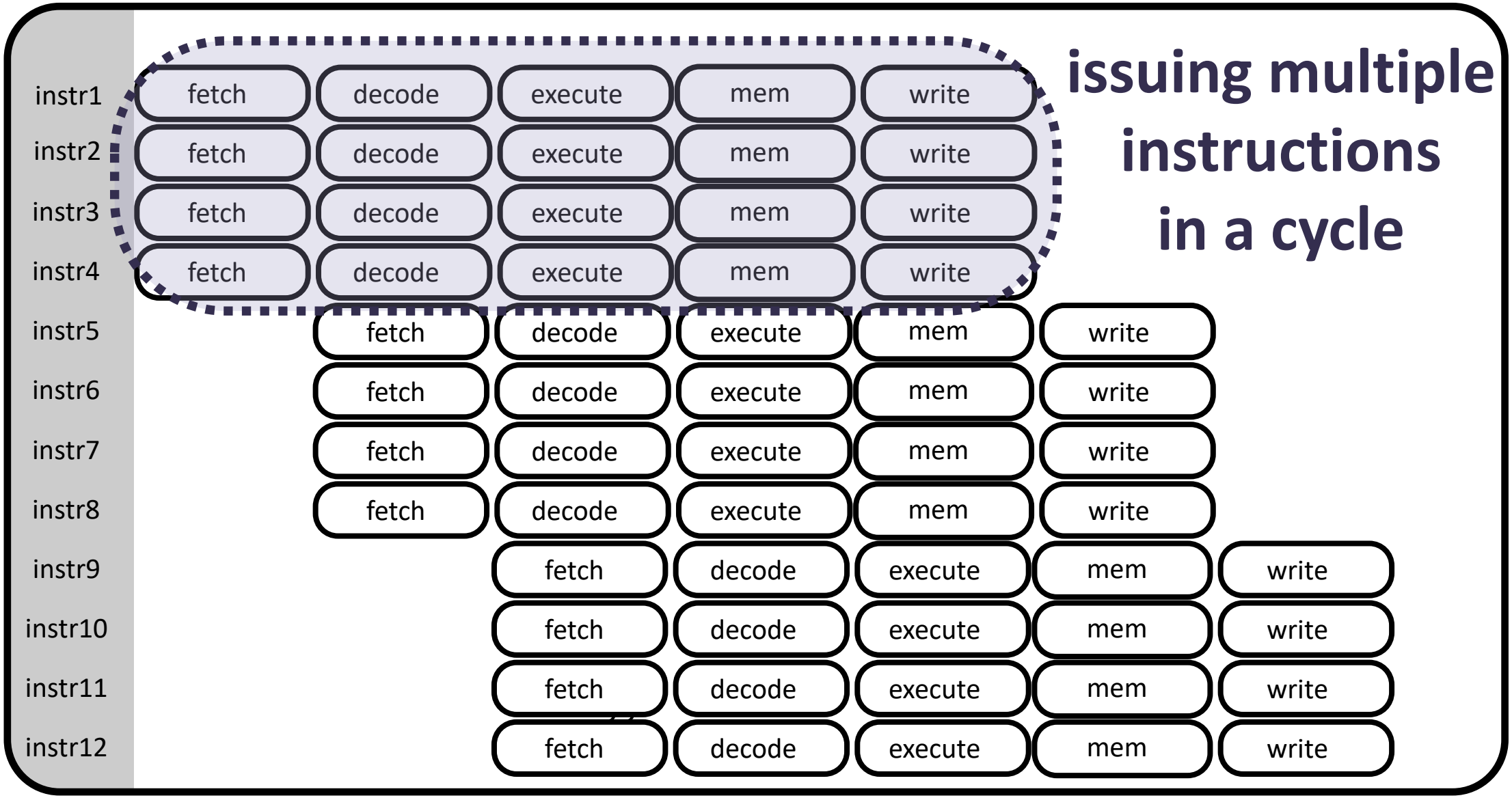
overlapping stages of different instructions



fundamental way to parallelize implicitly

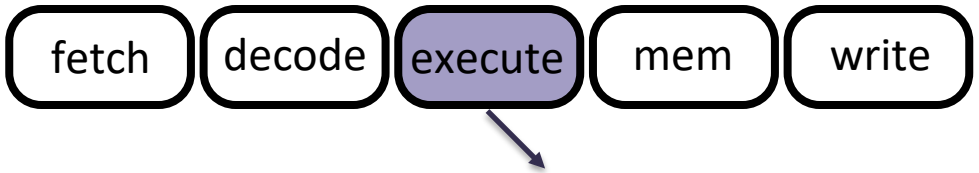
superscalar CPU

example of 4-way superscalar CPU



out-of-order (OoO) execution

allows a processor to execute instructions based on the availability of input data rather than strictly following the instruction ordering of a program



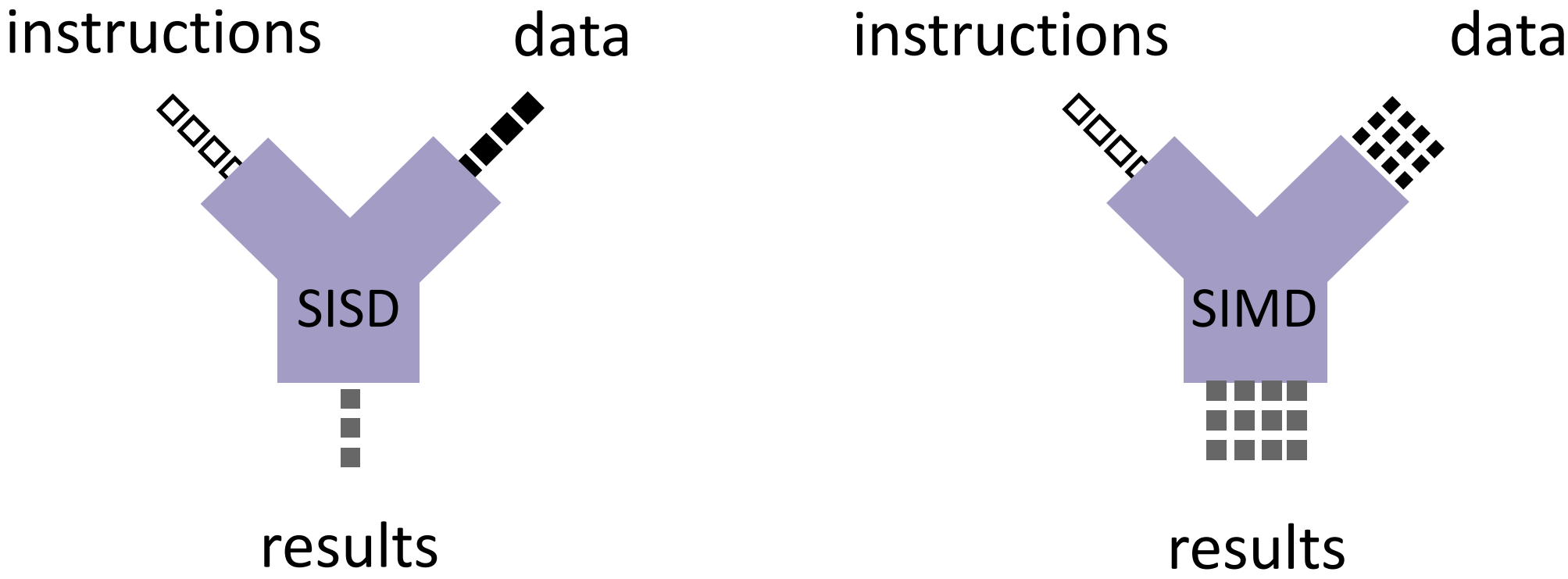
done here

example:

(1) r1 ← r2 / r3
(2) r4 ← r1 + r5
(3) r6 ← r7 * r8

independent from the previous two, can be executed independently in parallel or before 1 & 2

single-instruction multiple data (SIMD)



also, implicit data parallelism

but this one has to be managed/issued by the software

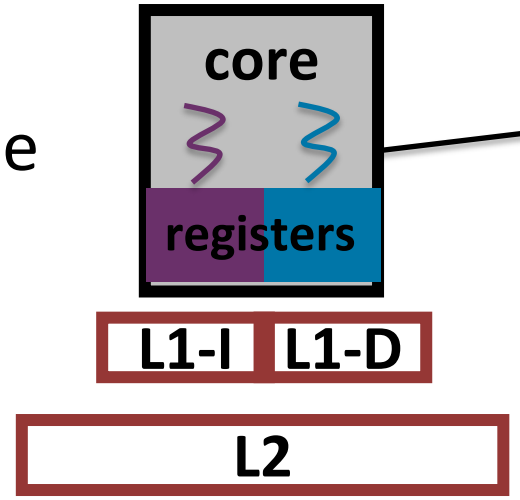
simultaneous multithreading (SMT)

today, also known as hyperthreading

keep state/context of multiple threads via more register space

each CPU cycle,
the threads are swapped

no need for a context switch
for this by the operating system



a core that supports
two hardware contexts
(or logical threads)

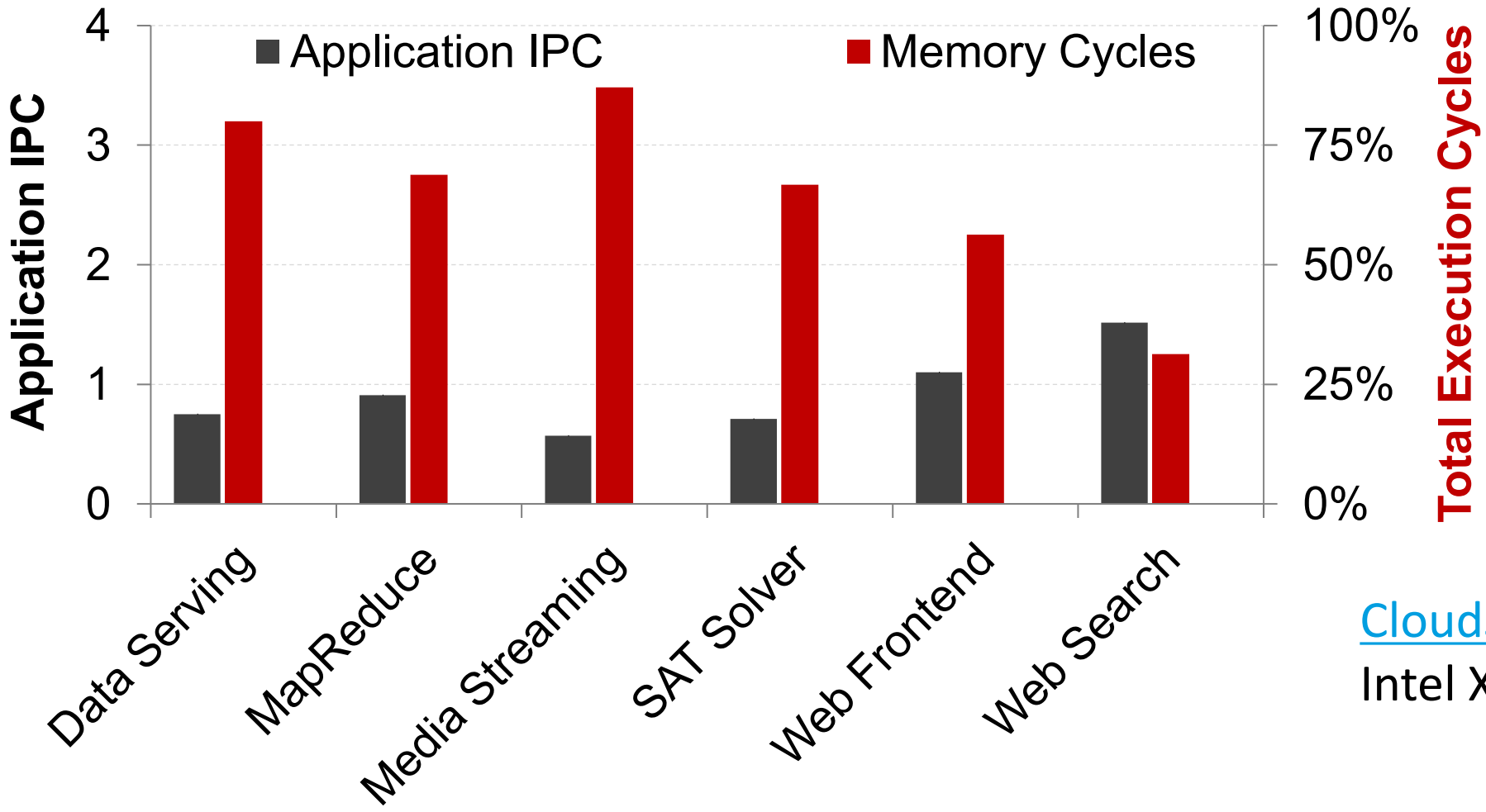
design trade-off: how to use the transistors in a core?

- to create higher per-thread speed by making the instruction pipeline more complex
- vs more threads

more pressure on shared hardware resources (e.g., caches)
if not used right, may even hurt performance

memory stalls in data-intensive systems

"Clearing the Clouds" [ASPLOS12]



CloudSuite on Intel Xeon X5670

data-intensive systems are known to suffer from memory stalls

agenda

lecture

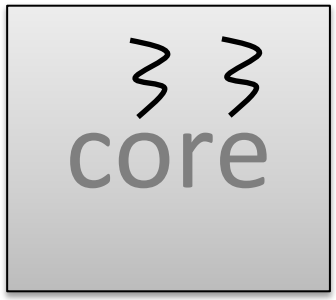
- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

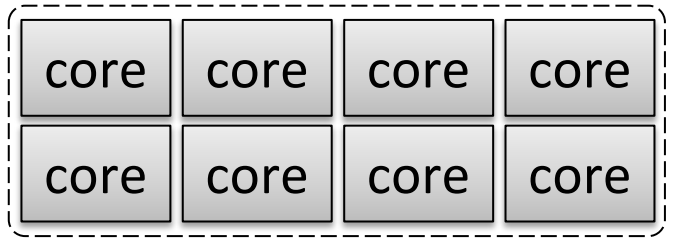
- paper discussion
- c/cpp examples
- setting core affinity

types of hardware parallelism

implicit/vertical parallelism ⟷ **explicit/horizontal parallelism**



single-core
instruction & data parallelism
simultaneous multithreading



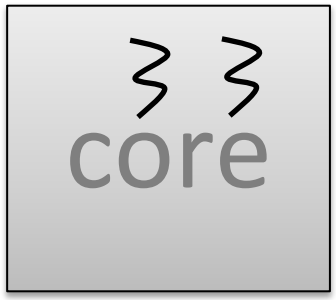
multicores
multiple threads run
in parallel on different cores

implicit parallelism → (almost) free lunch

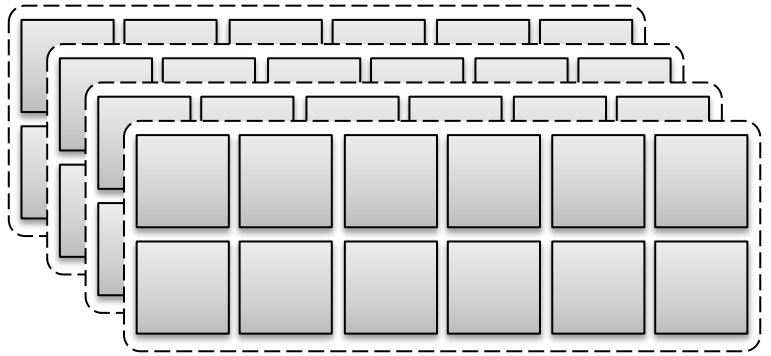
explicit parallelism → must work hard to leverage it

types of hardware parallelism

implicit/vertical parallelism ⟷ **explicit/horizontal parallelism**



single-core
instruction & data parallelism
simultaneous multithreading



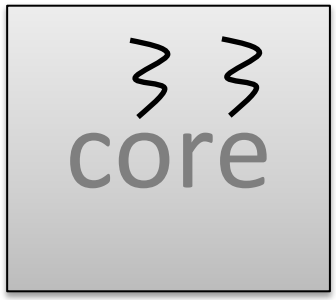
multisocket multicores
multiple processors/ CPUs
in one machine

implicit parallelism → (almost) free lunch

explicit parallelism → must work hard to leverage it

types of hardware parallelism

implicit/vertical parallelism ⟷ **explicit/horizontal parallelism**



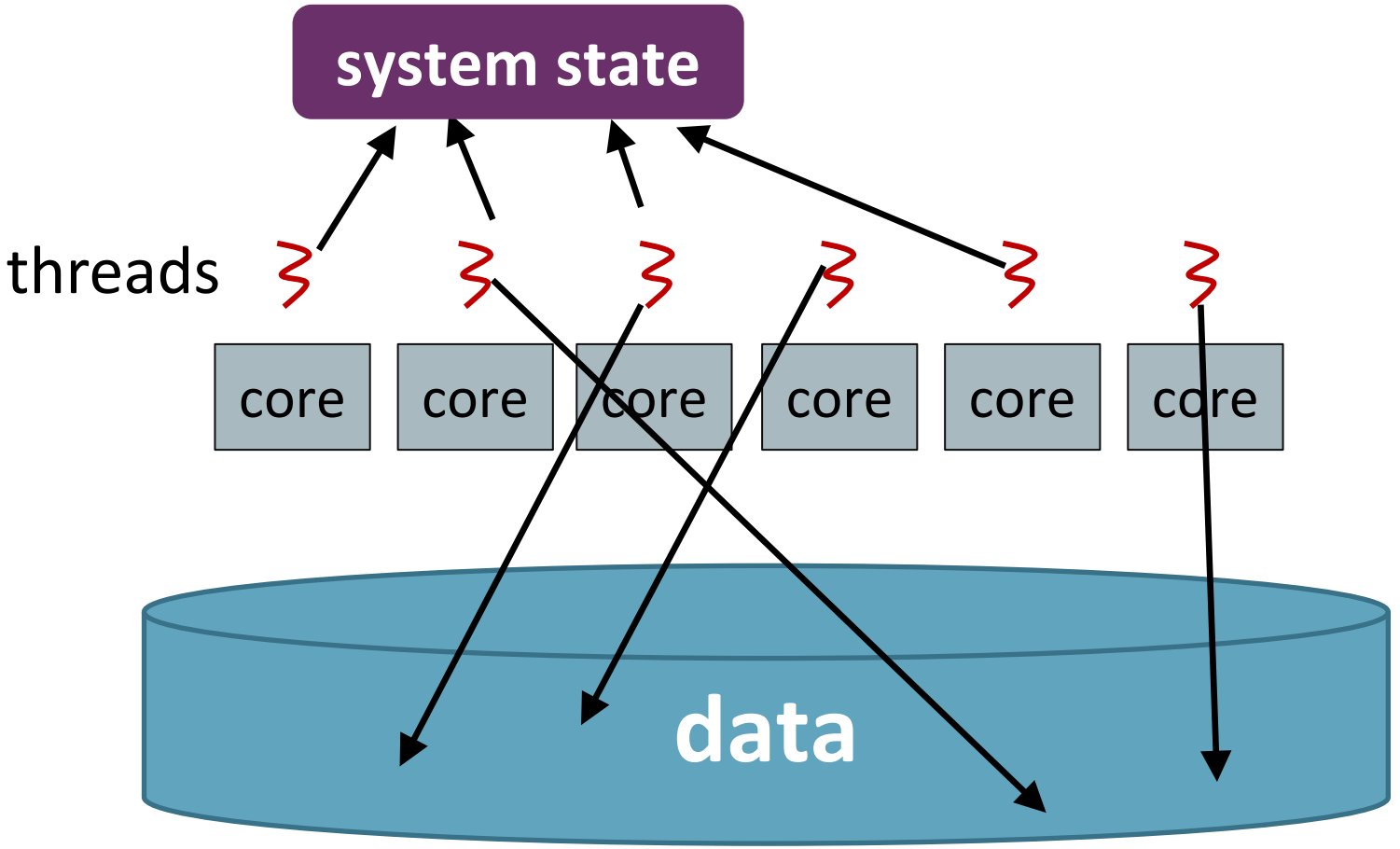
single-core
instruction & data parallelism
simultaneous multithreading

distributed systems
running a program over
multiple machines

implicit parallelism → (almost) free lunch

explicit parallelism → must work hard to leverage it

why explicit parallelism is hard to leverage?

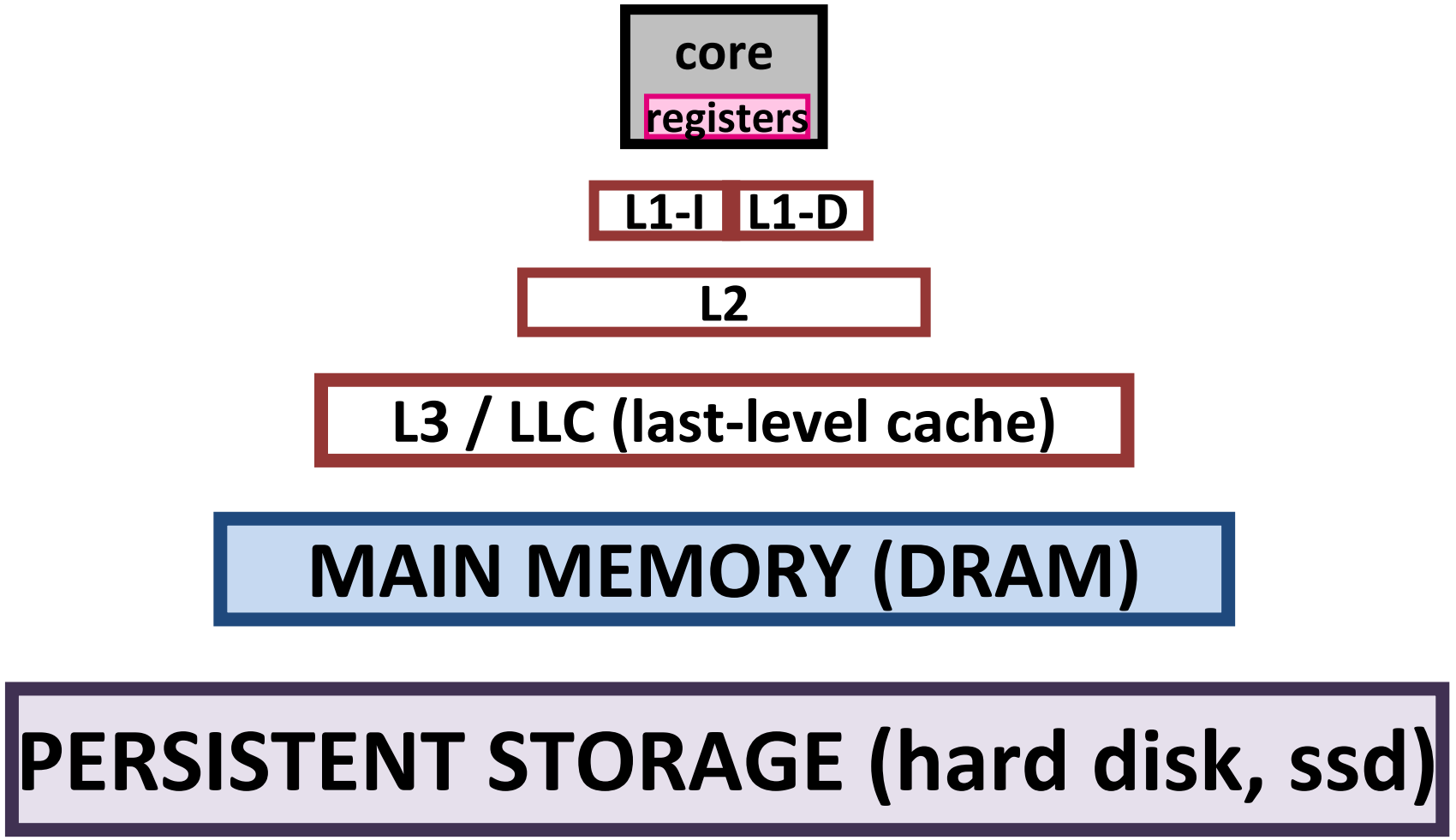


synchronization

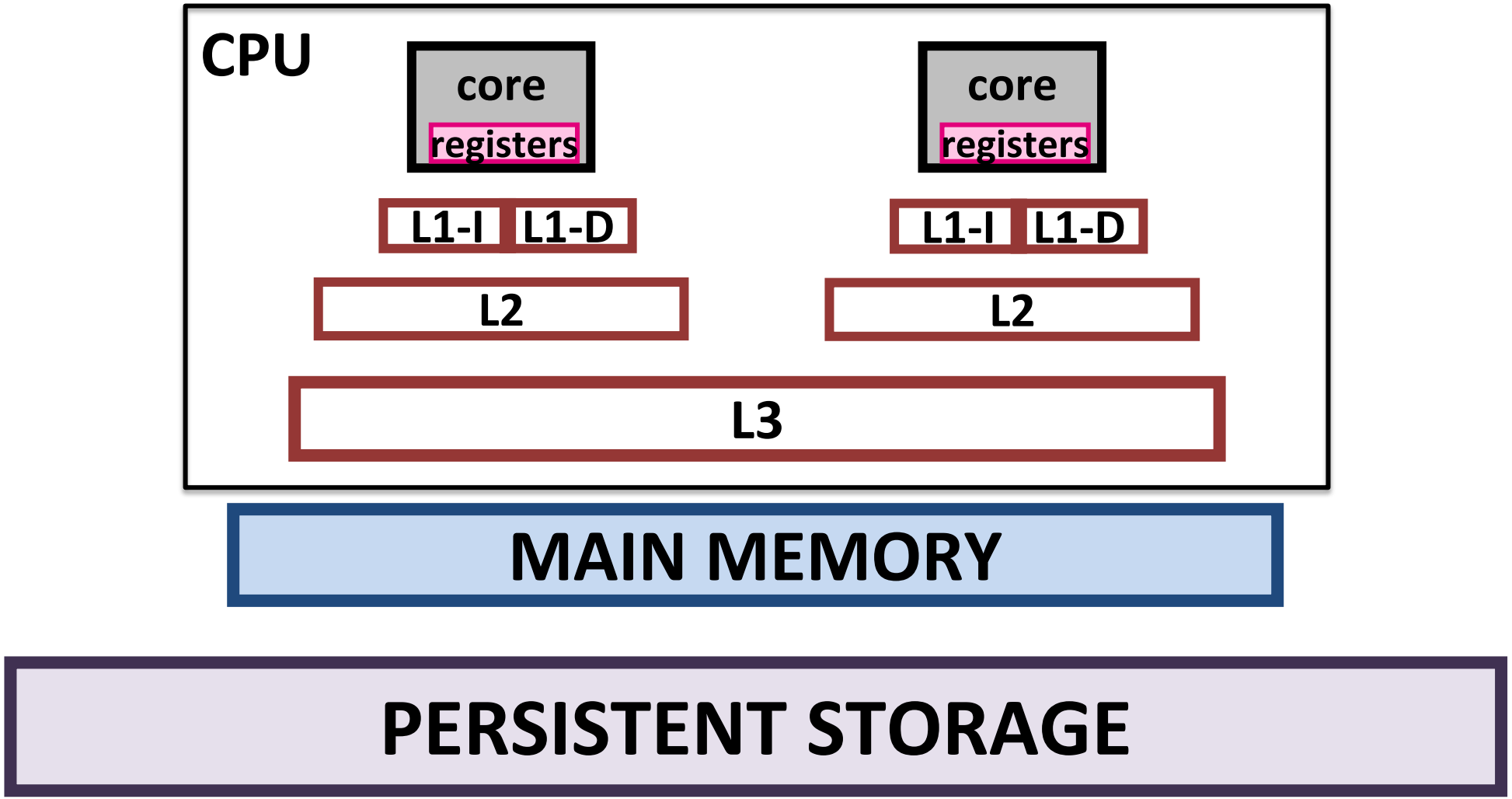
too many locks ➔ low performance

if a system has many unpredictable accesses to shared data, need to synchronize the threads while accessing to data

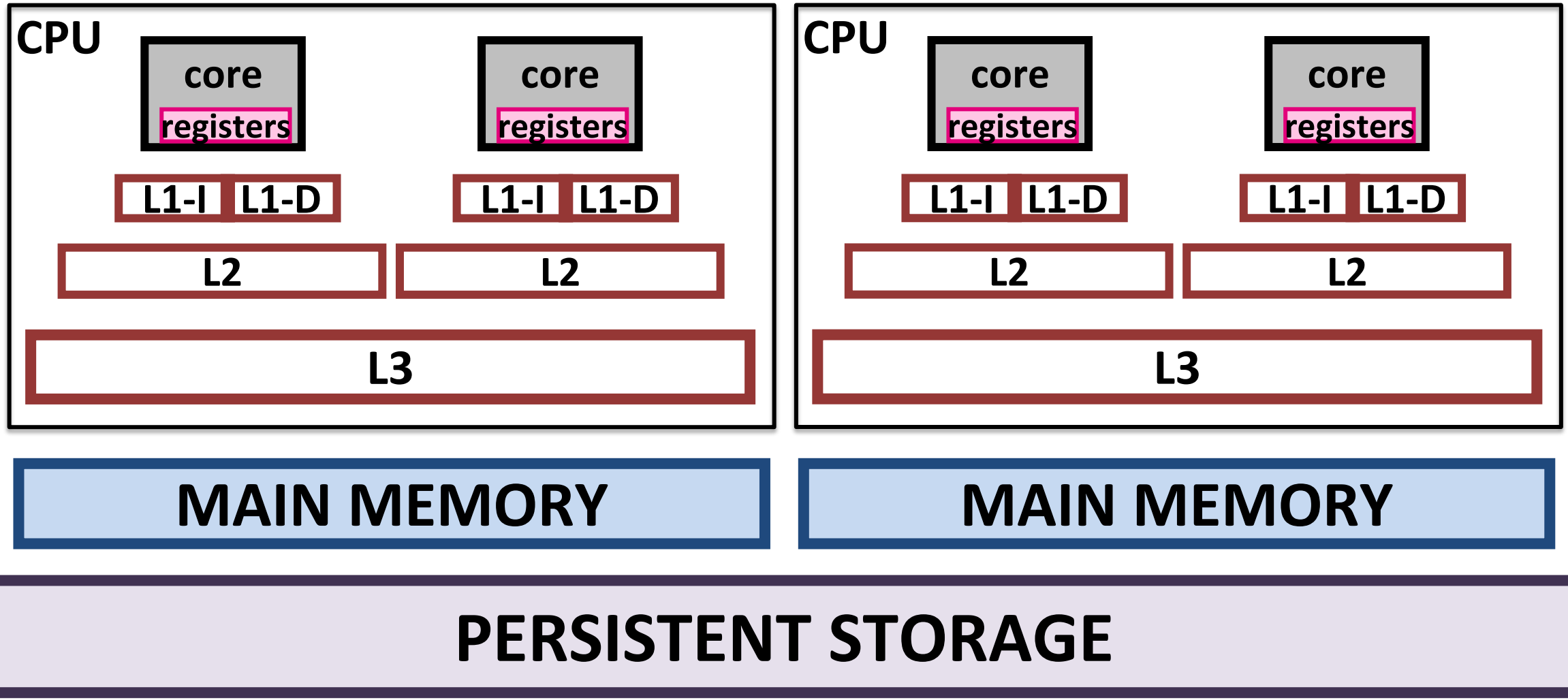
single-core CPU storage hierarchy



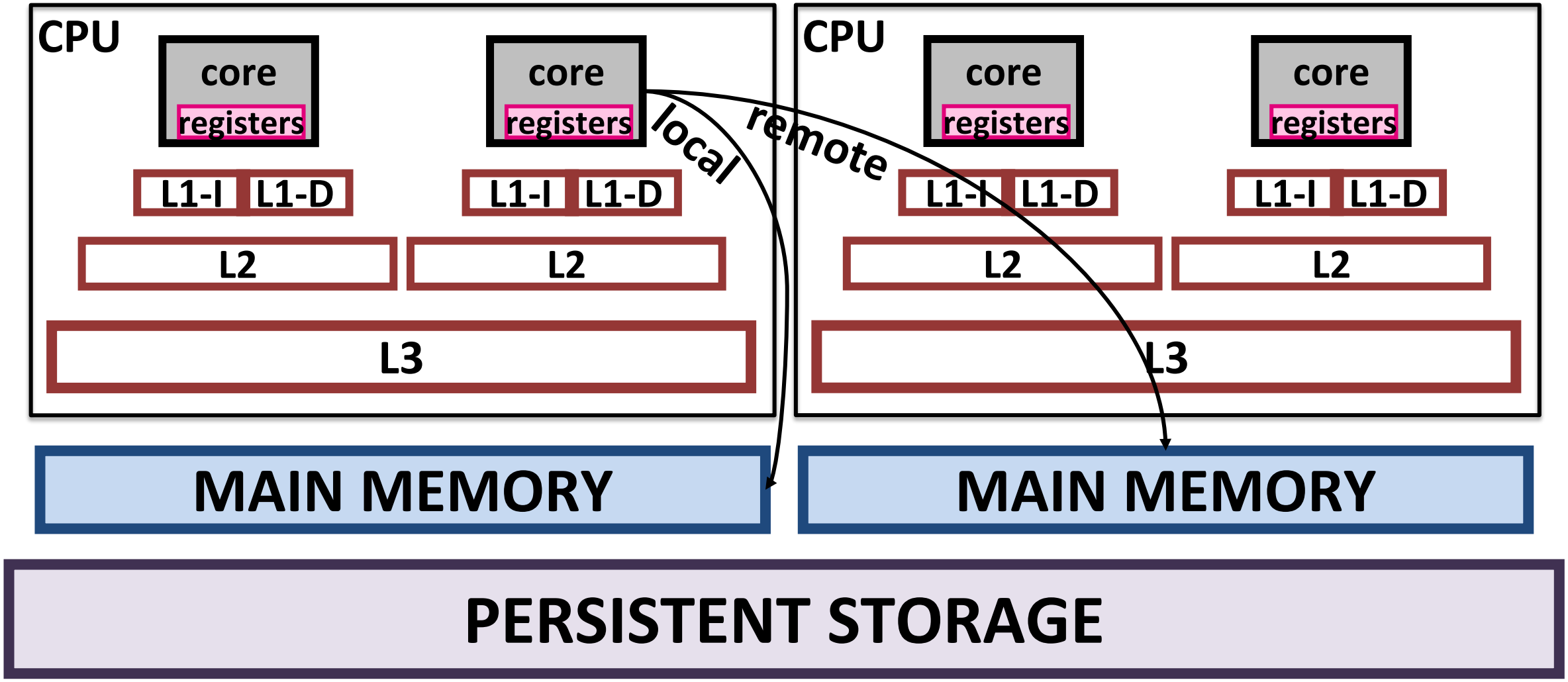
multicore CPU storage hierarchy



multi-socket multicore CPU storage hierarchy

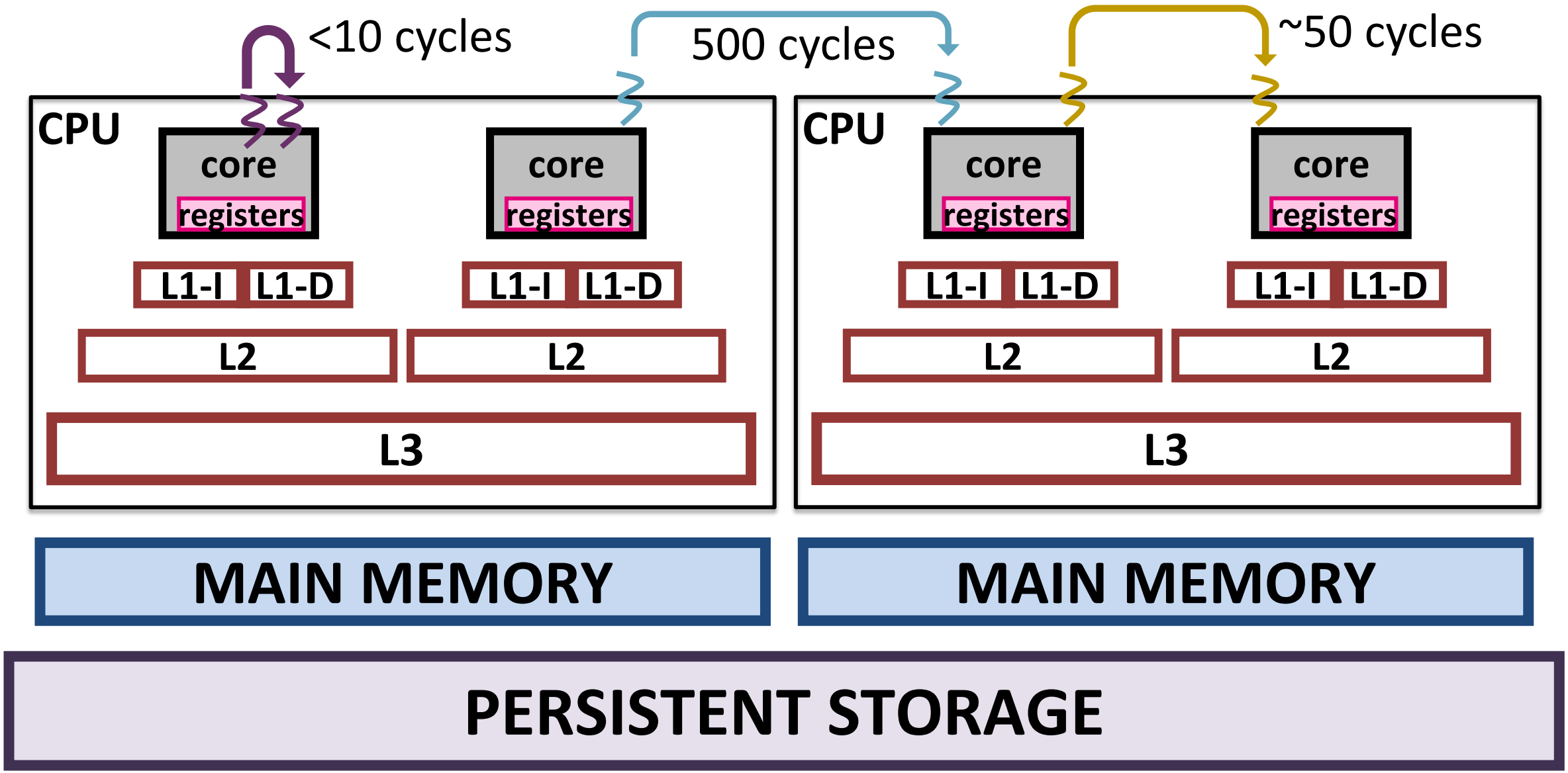


NUMA = non-uniform memory access



local memory access is faster than remote one!

NUMA impact


<10 cycles
500 cycles
~50 cycles
CPU
core
registers
L1-I
L1-D
L2
L3
CPU
core
registers
L1-I
L1-D
L2
L3
MAIN MEMORY
MAIN MEMORY
PERSISTENT STORAGE

agenda

lecture

- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

- paper discussion
- c/cpp examples
- setting core affinity

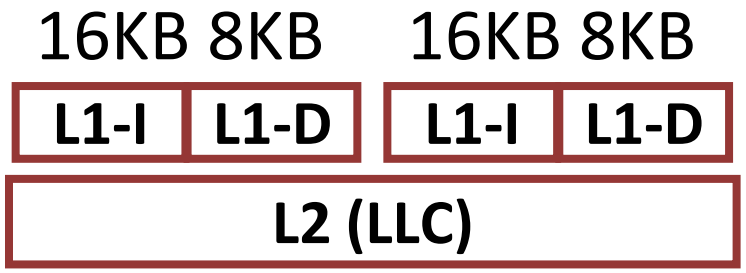
Sun SPARC CPUs

discontinued!

data-intensive workloads have

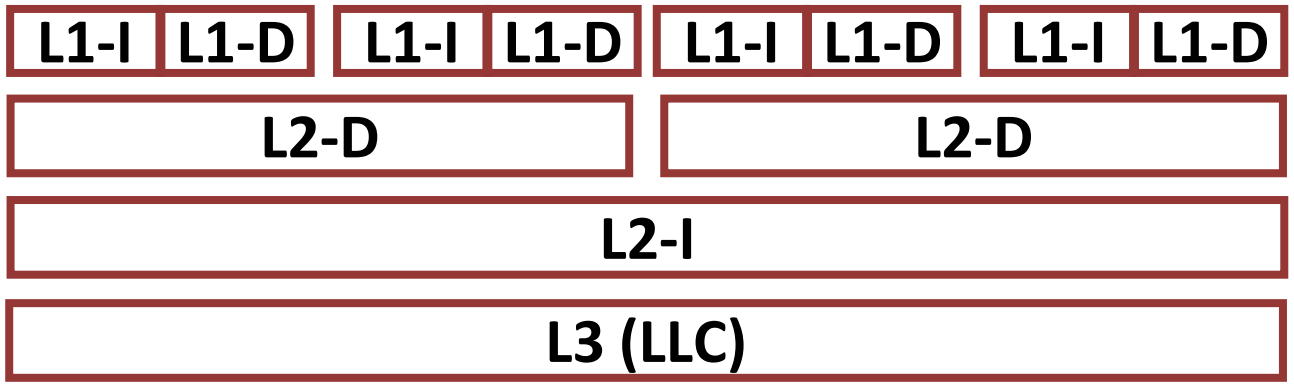
1. **large instruction footprint**
2. **low temporal data locality**

UltraSPARC T2 (Niagara 2)



used in “data partitioning on chip multiprocessors” paper

SPARC M7 & M8



let’s check some hardware topology!

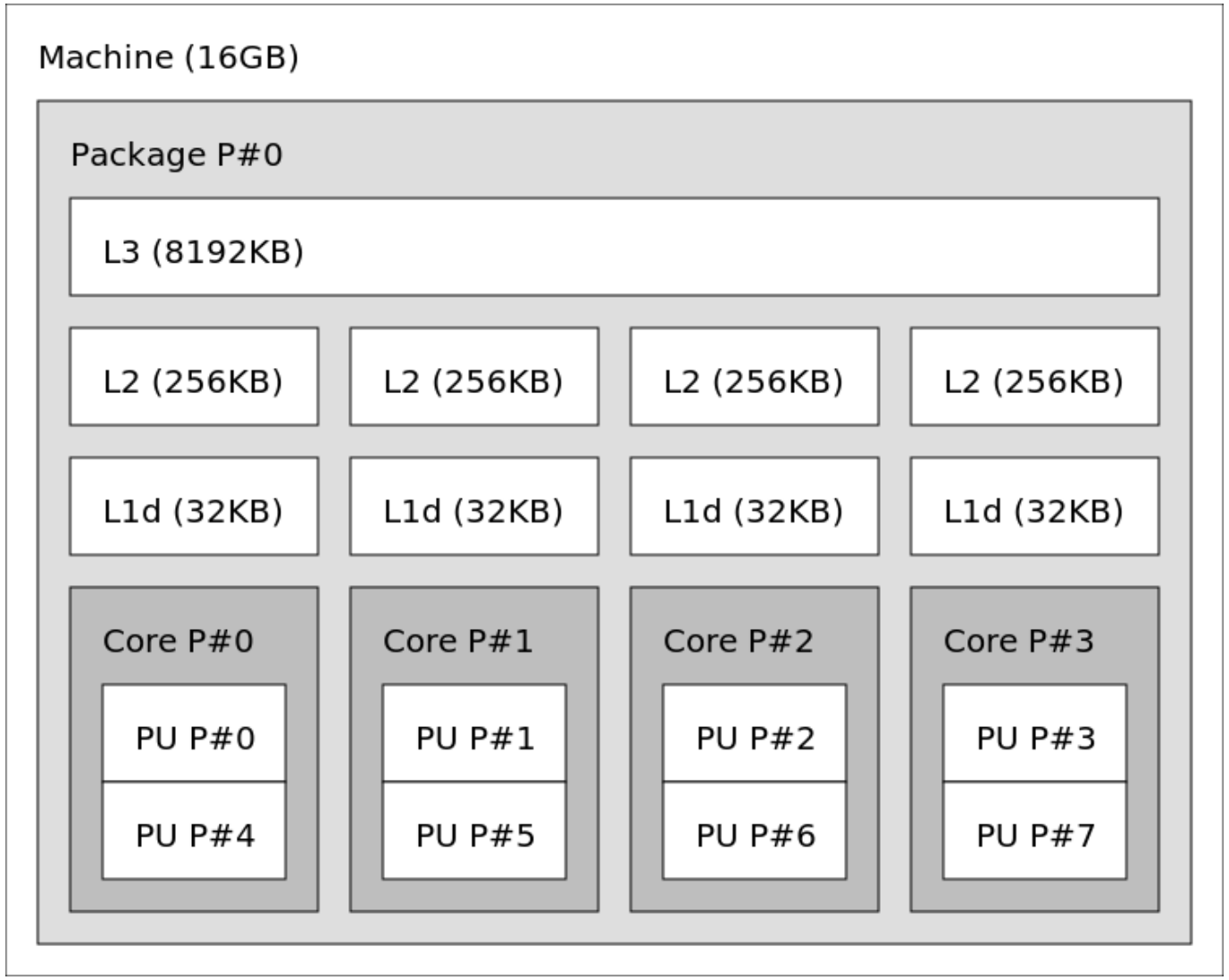
$ lstopo

or for better visuals

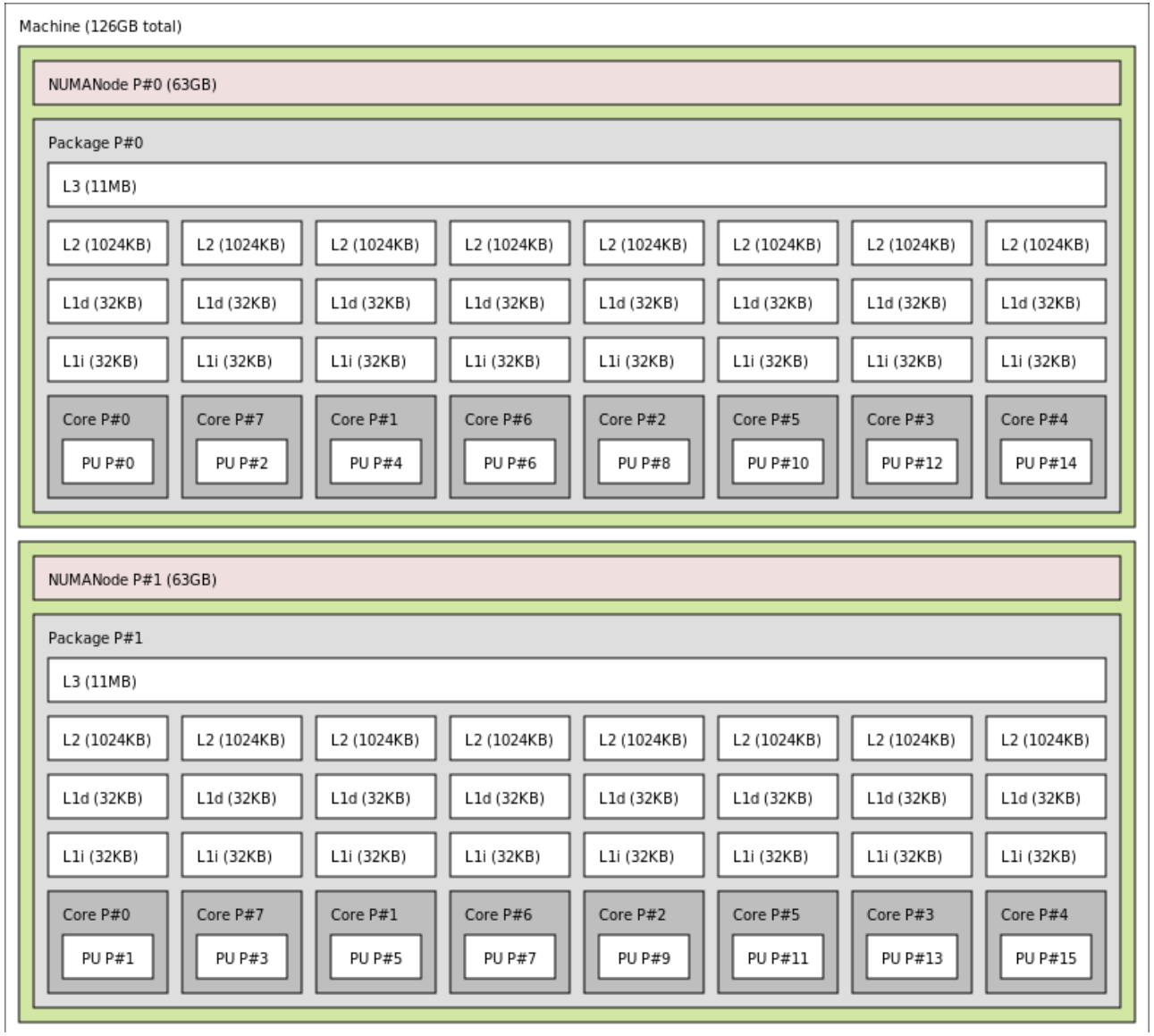
$ lstopo --output-format png -v --no-io > cpu.png

- try this out on your laptop
- report what you see
- does it look like the hierarchy we have been discussing?
- if not, what is different?

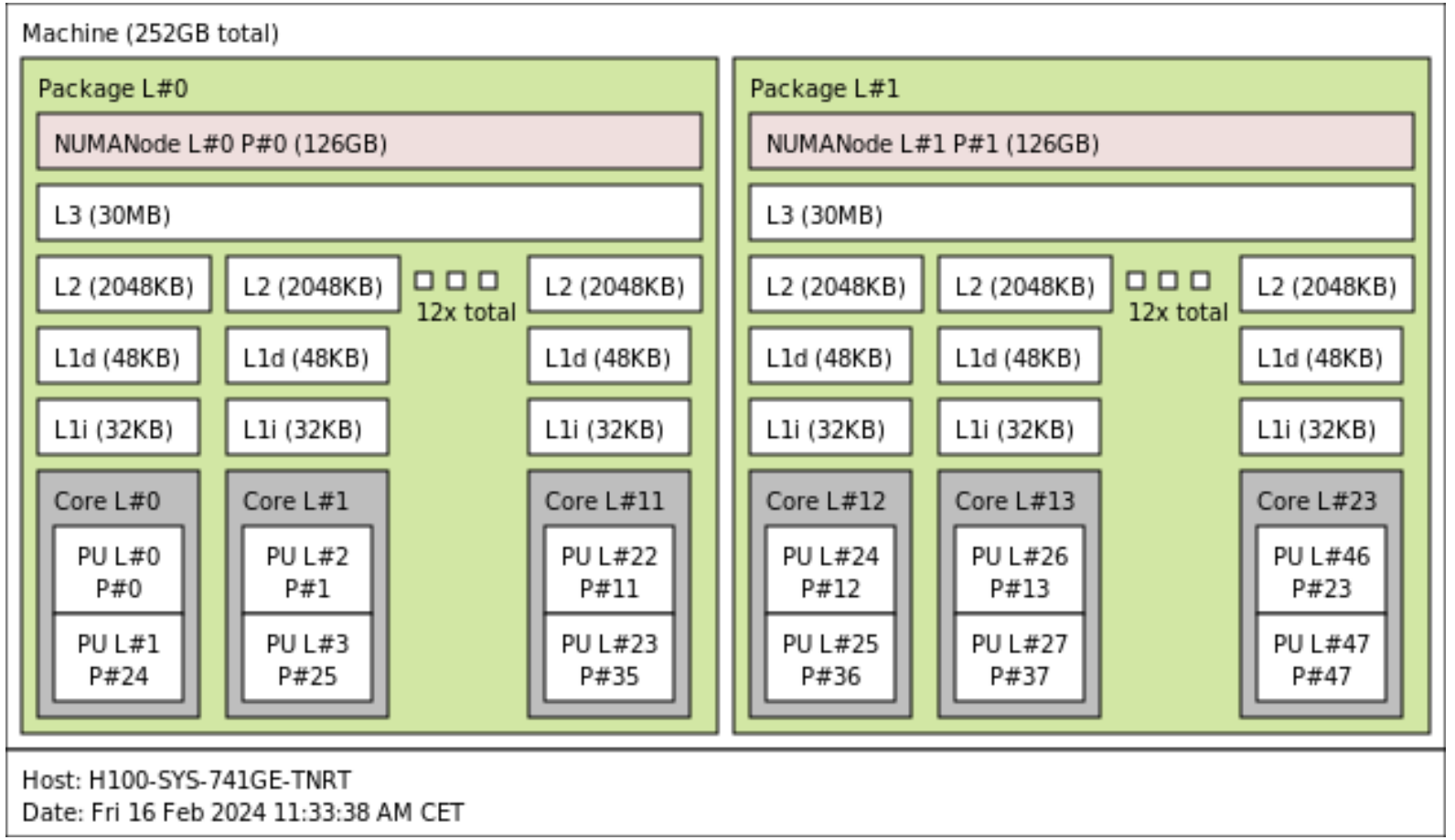
laptop with Intel CPU – hyper-threading on



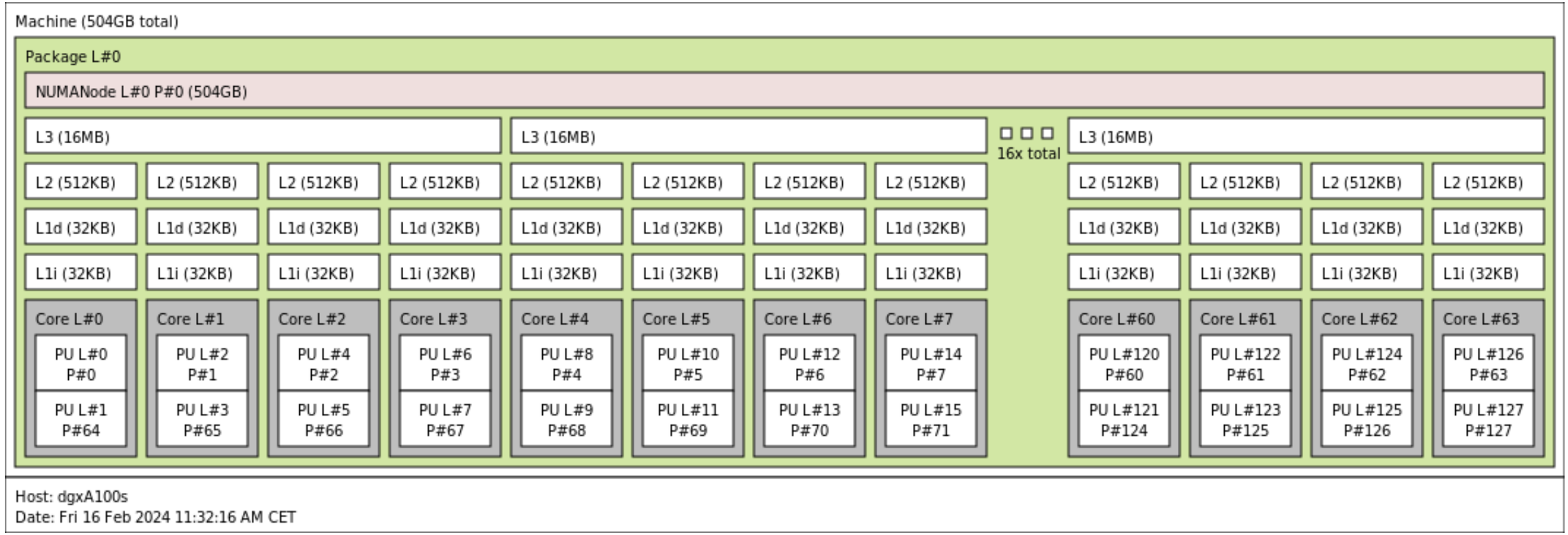
class server – hyper-threading off



newer intel CPUs – hyper-threading on

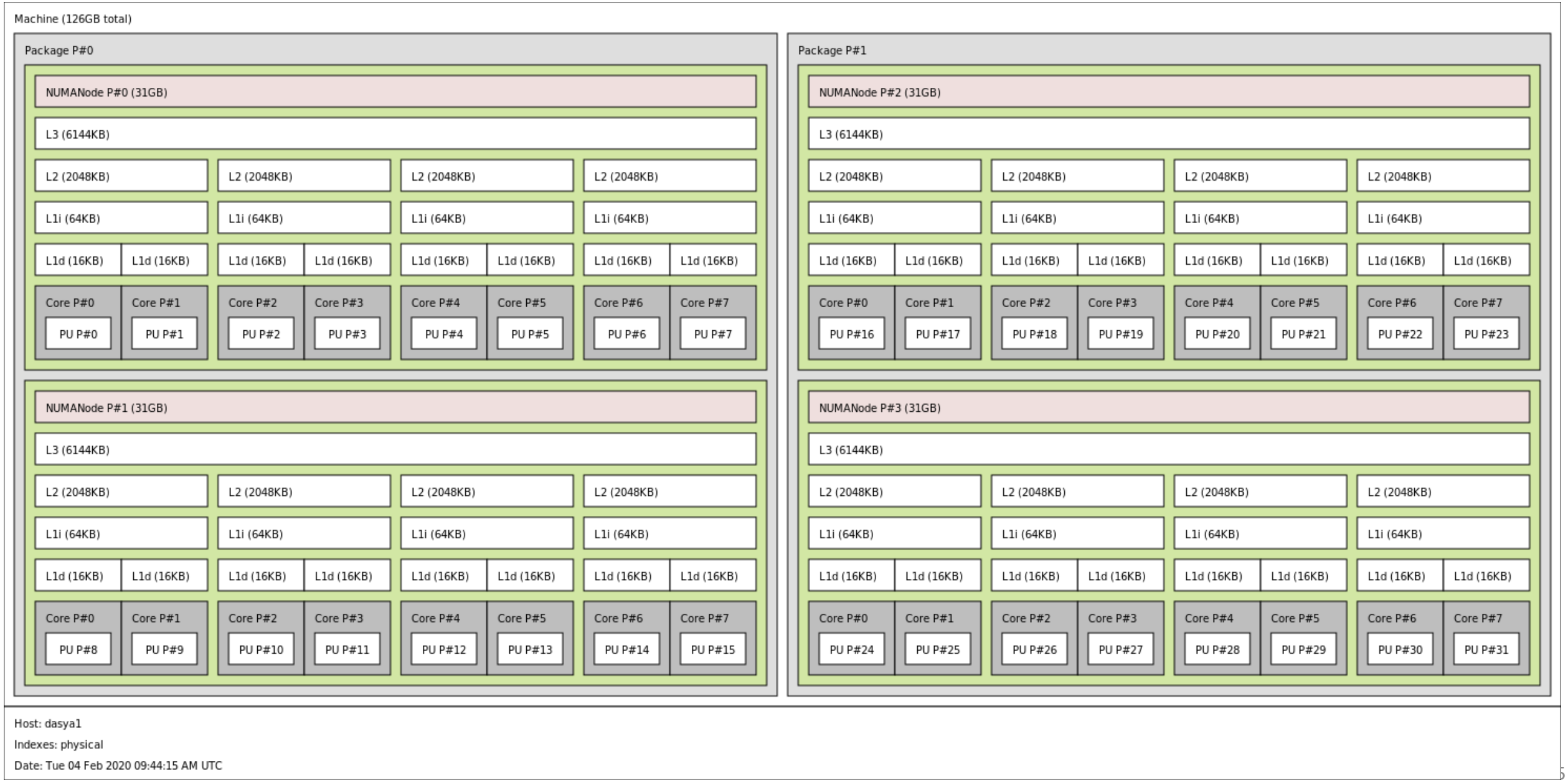


NVIDIA DGX A100 Station – AMD CPU



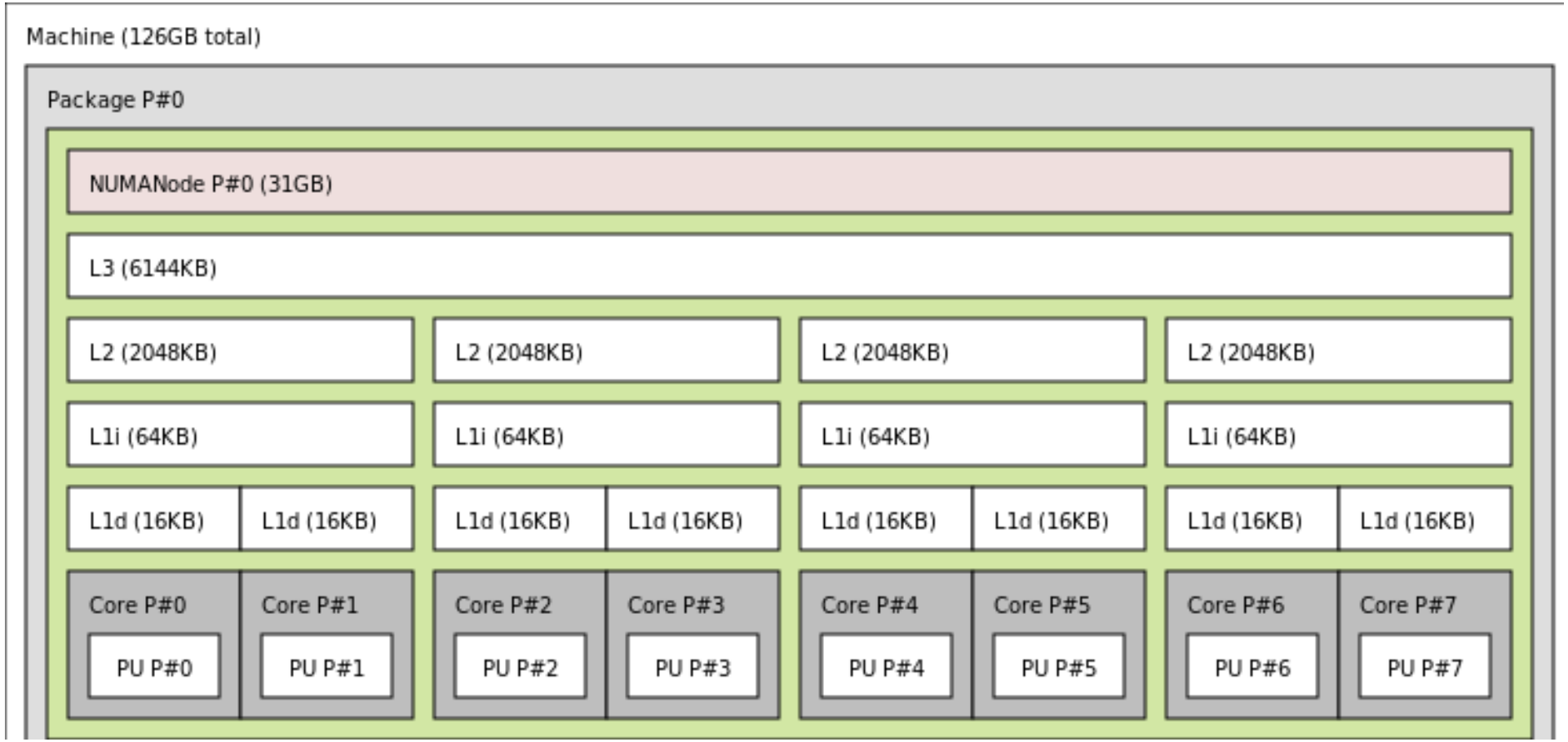
- **AMD 7742 (Rome) CPU, 64 cores in total**
- **hyper-treading on, 4 A100 GPUs attached**
- **8 dies & 16 core complexes**
- **creates additional layers in the NUMA hierarchy**

AMD 6386 (older) – clustered multi-threading



AMD 6386 (older) – clustered multi-threading

zoom in



agenda

lecture

- storage hierarchy overview
- local memory hierarchy: DRAM & caches
- hardware parallelism: implicit & explicit
- different memory hierarchies

exercises

- paper discussion
- c/cpp examples
- setting core affinity

experimental design case study

"Data Partitioning on Chip Multiprocessors"

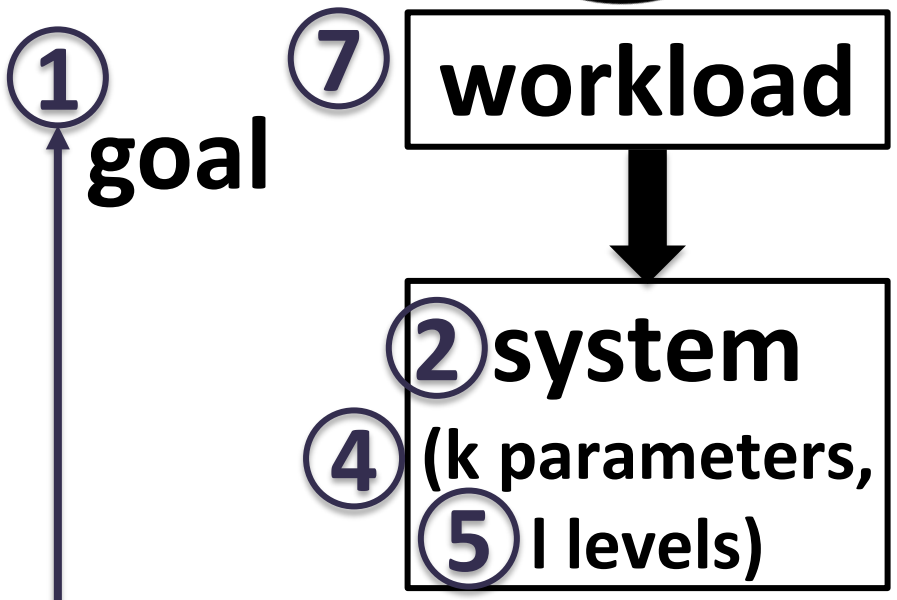
analyze the experiments in this research paper

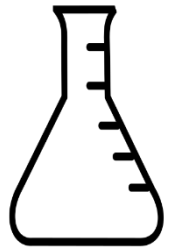
focusing on the following questions:

- what is the goal?
- what is/are the system(s)?
- what are the metrics?
- how many runs for the experiments?
- what are the good or bad points about the experiments?

also relevant for your first project!!

performance analysis – reminder





experiment

6 **decide on technique (model, simulate, measure)**

8 **actual experiment**

9 **interpret results**

10 **present results**

keep the big picture in mind at all times!

goal: investigating how existing data partitioning mechanisms behave on multicore hardware. do old findings still hold?
system:
application-level: 4 partitioning mechanisms
OS-level: general purpose OS
hardware-level: multicore hardware
metrics: tuples per second (throughput), data TLB misses, data cache misses, metadata overhead
parameters (k): page size, hash bits (or #partitions), #threads,
levels(l): page size: 8KB, 64KB, 4MB, 256MB
hash bits: 1 … 18, #threads: 1, 2, 4, 8, 16, 32
type of experiment: measurements (main), analytical approach to cache/tlb misses
workloads: with uniform data (8byte keys – 8byte payloads), 2^{24} tuples
experimental runs: 8 trials

useful commands

```
$ lscpu                    # cpu topology

$ cat /proc/meminfo        # checking hugepage size

$ cat /proc/cpuinfo        # core by core info

$ cpuid                    # as above, but also TLB info
```