Concurrent programming: From theory to practice

Concurrent Algorithms 2014

Vasileios Trigonakis
Georgios Chatzopoulos

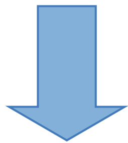
From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

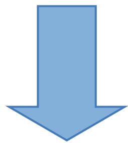


Design (pseudo-code)

From theory to practice

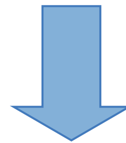
Theoretical (design)	Practical (design)	Practical (implementation)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs



Design (pseudo-code)

- System models
  - shared memory
  - message passing
- Finite memory
- Practicality issues
  - re-usable objects
- **Performance**

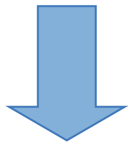


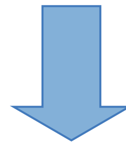
Design (pseudo-code, prototype)

From theory to practice

Theoretical (design)	Practical (design)	Practical (implementation)

• Impossibilities
• Upper/Lower bounds
• Techniques
• System models
• Correctness proofs
	• System models
	  • shared memory
	  • message passing
	• Finite memory
	• Practicality issues
	  • re-usable objects
	• **Performance**
		• **Hardware**
		• Which atomic ops
		• Memory consistency
		• Cache coherence
		• Locality
		• **Performance**
		• **Scalability**





Design (pseudo-code)	**Design (pseudo-code, prototype)**	**Implementation (code)**

Outline

- CPU caches
- Cache coherence
- Placement of data
- Memory consistency
- Hardware synchronization
- Concurrent programming techniques on locks

Outline

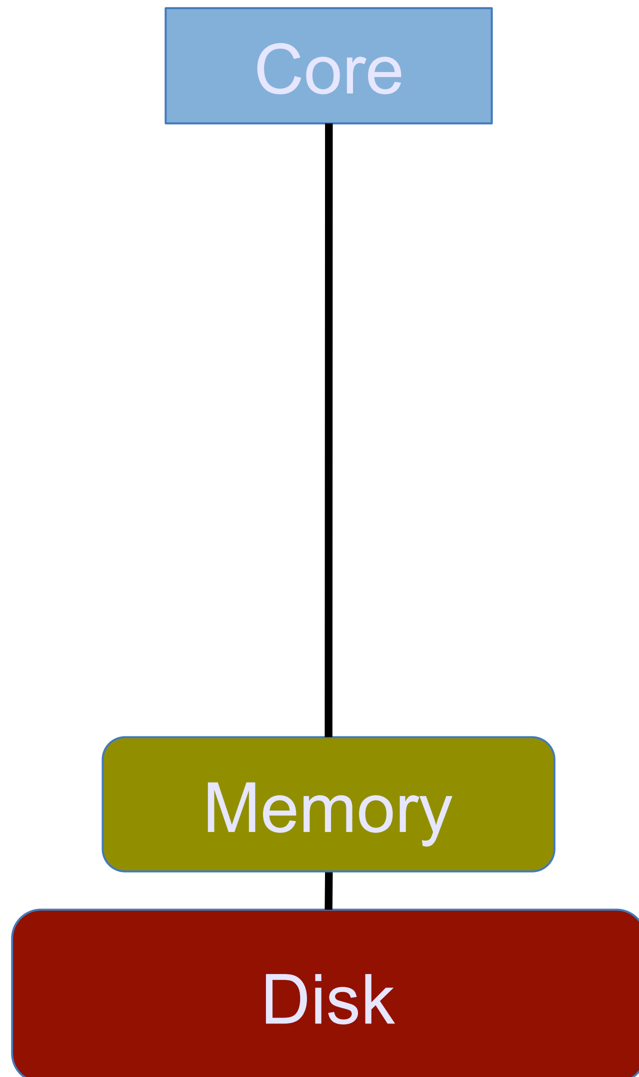
- **CPU caches**
- Cache coherence
- Placement of data
- Memory consistency
- Hardware synchronization
- Concurrent programming techniques on locks

Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms

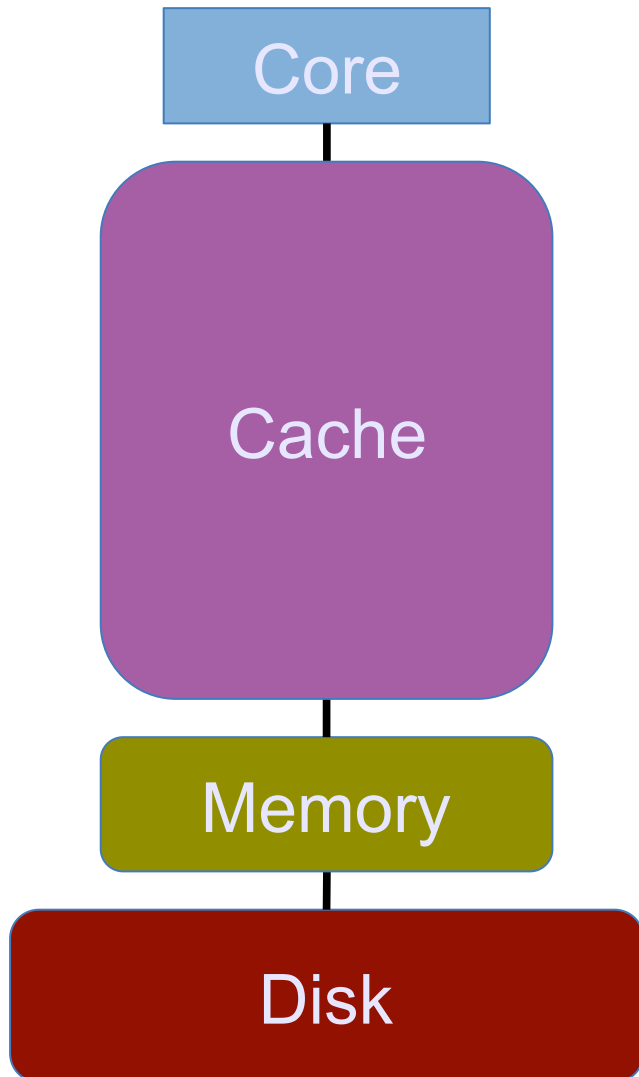
Disk

Why do we use caching?



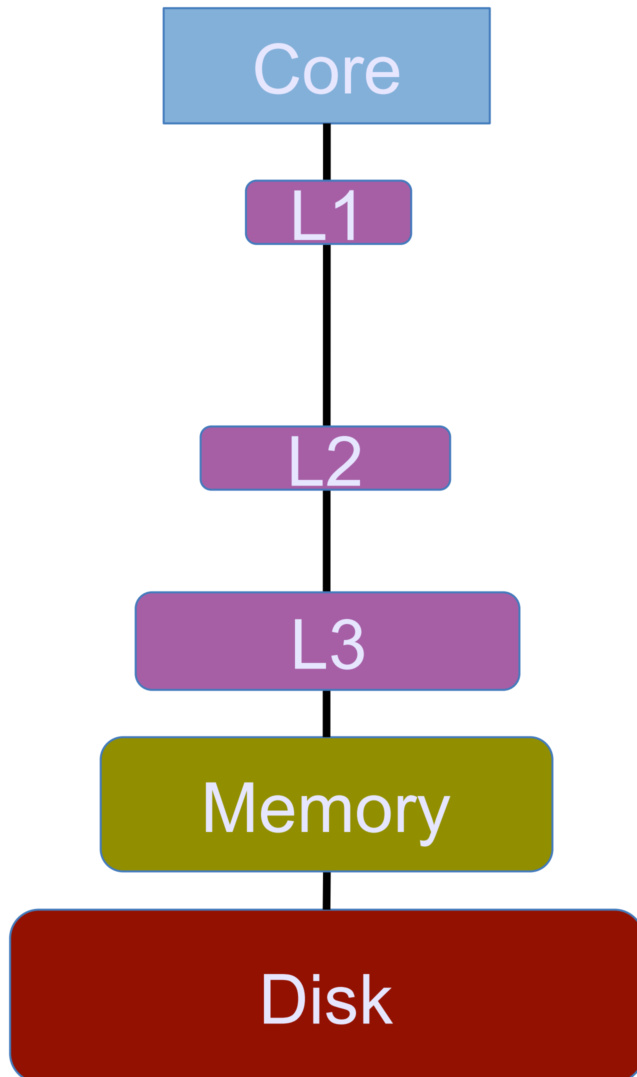
- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns

Why do we use caching?



- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Large = slow
  - Medium = medium
  - Small = fast

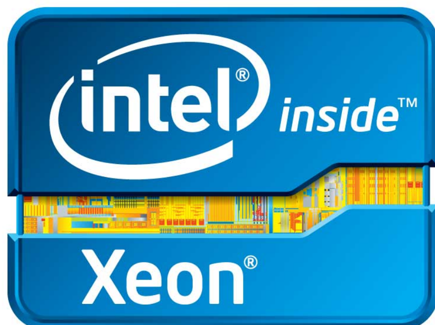
Why do we use caching?



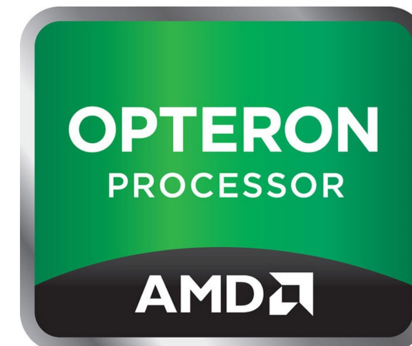
- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Core → L3 = ~20ns
  - Core → L2 = ~7ns
  - Core → L1 = ~1ns

Typical server configurations

- **Intel Xeon**
  - 10 cores @ 2.4GHz
  - L1: 32KB
  - L2: 256KB
  - L3: 24MB
  - Memory: 64GB



- **AMD Opteron**
  - 8 cores @ 2.4GHz
  - L1: 64KB
  - L2: 512KB
  - L3: 12MB
  - Memory: 64GB



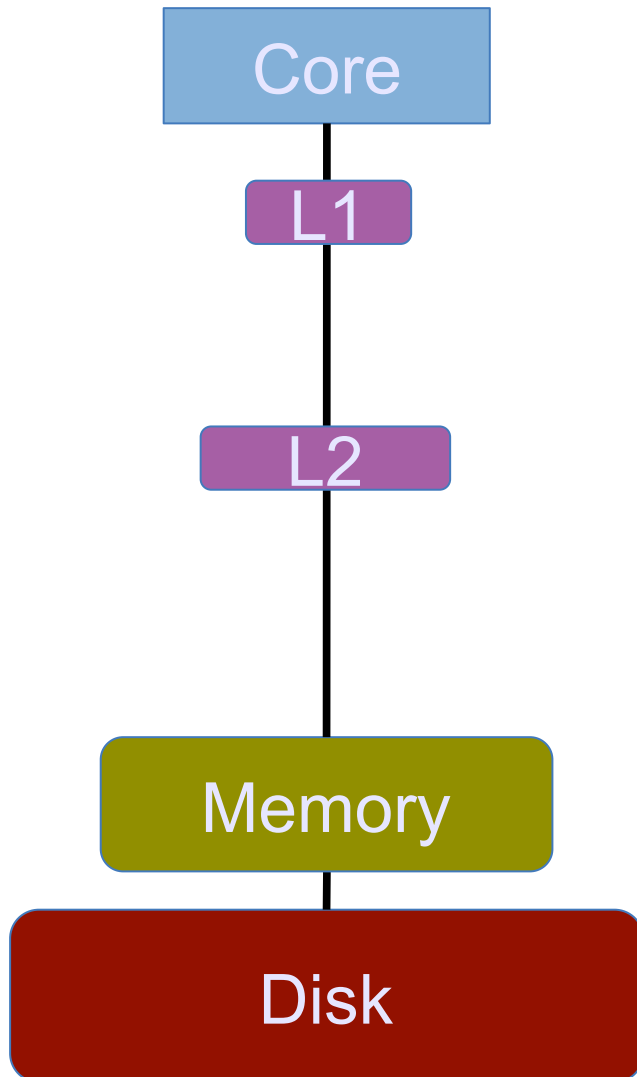
<u>Experiment</u>

Throughput of accessing some memory, depending on the memory size

Outline

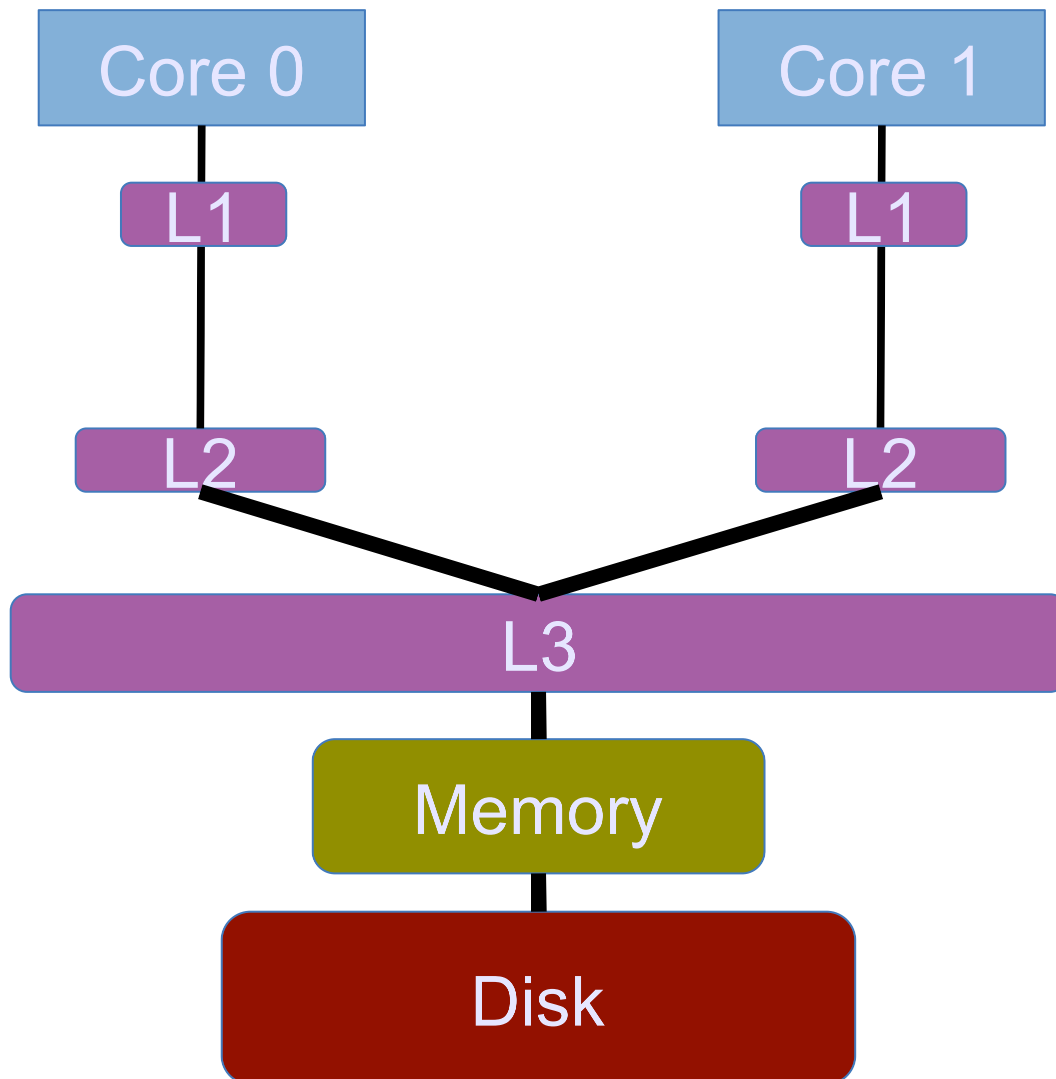
- CPU caches
- **Cache coherence**
- Placement of data
- Memory consistency
- Hardware synchronization
- Concurrent programming techniques on locks

Until ~2004: single-cores



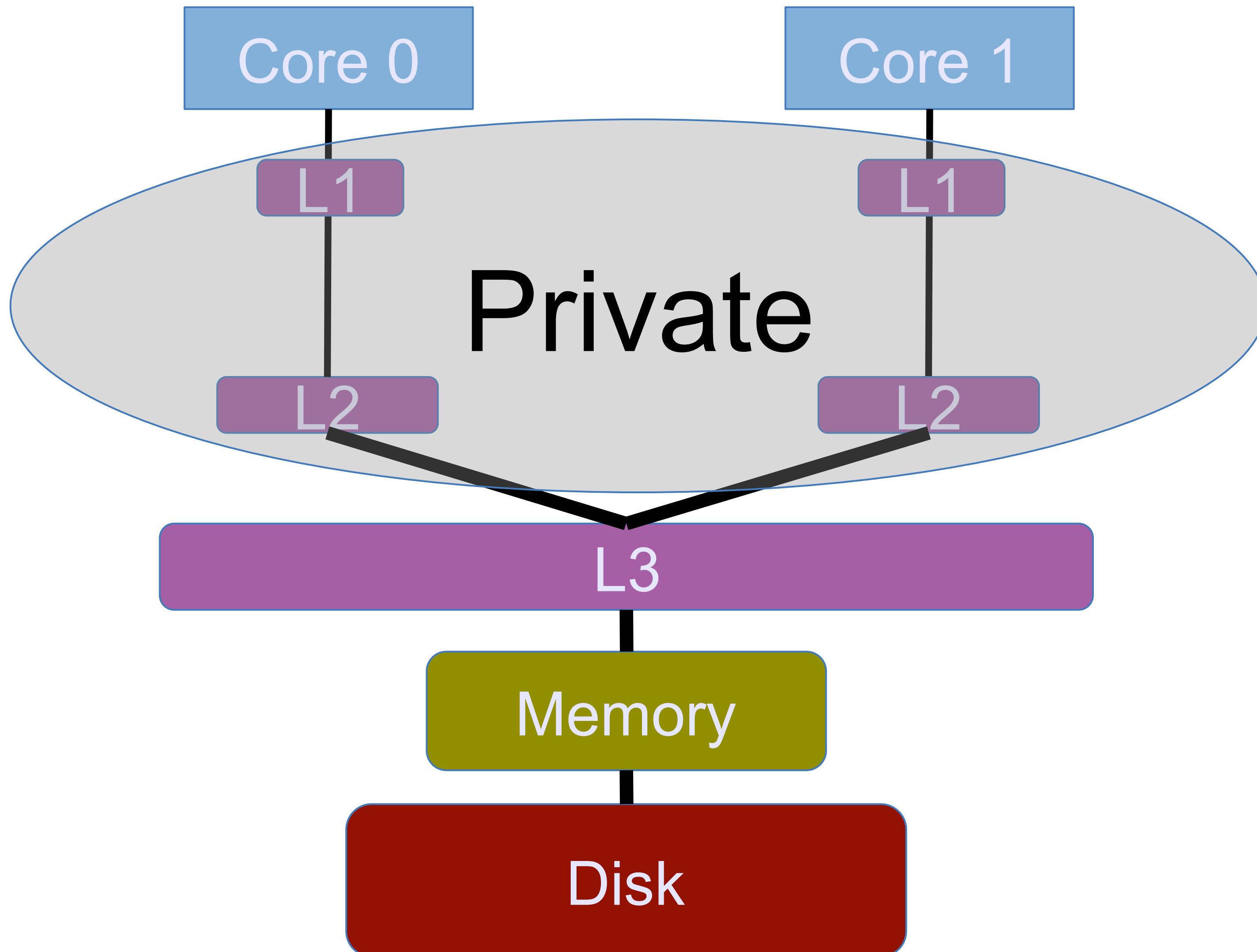
- Core freq: 3+GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → L3
  - Core → L2
  - Core → L1

After ~2004: multi-cores



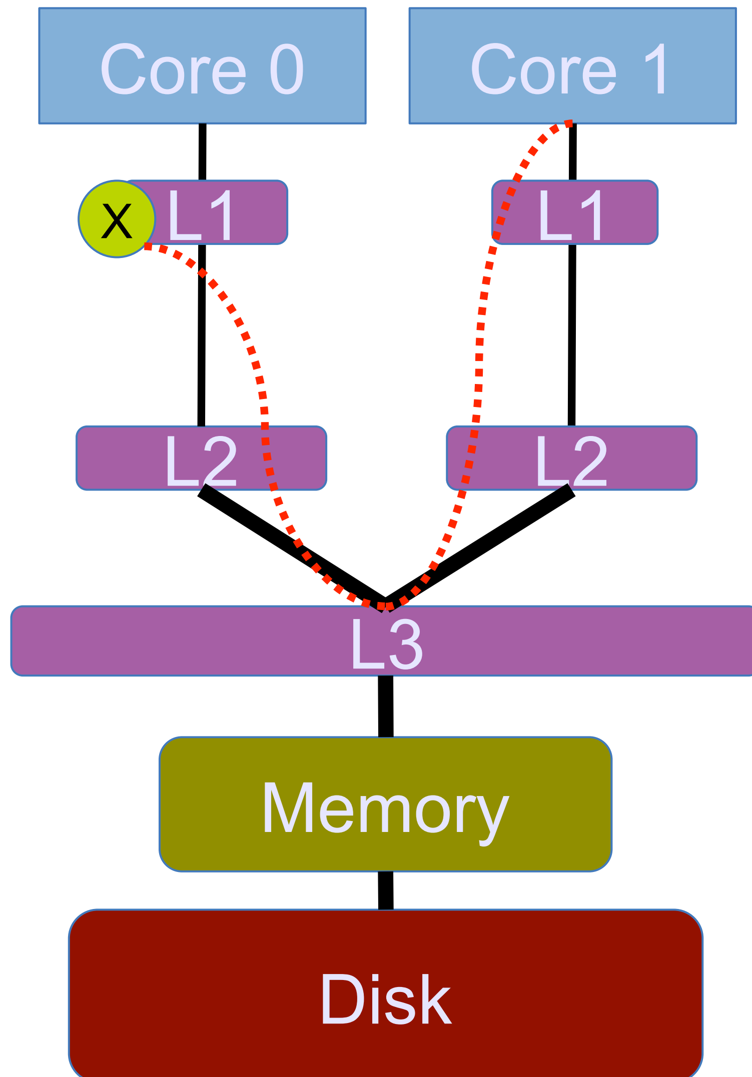
- Core freq: ~2GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → **shared** L3
  - Core → L2
  - Core → L1

Multi-cores with private caches



= multiple copies

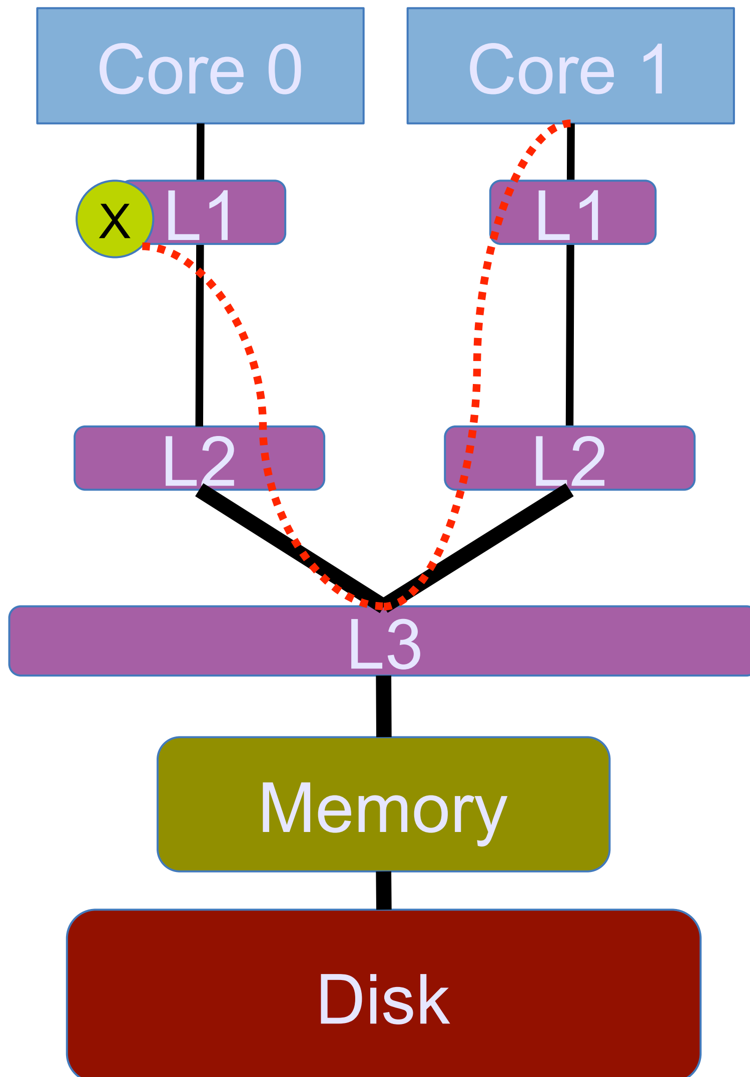
Cache coherence for consistency



Core 0 has X and Core 1

- wants to write on X
- wants to read X
- did Core 0 write or read X?

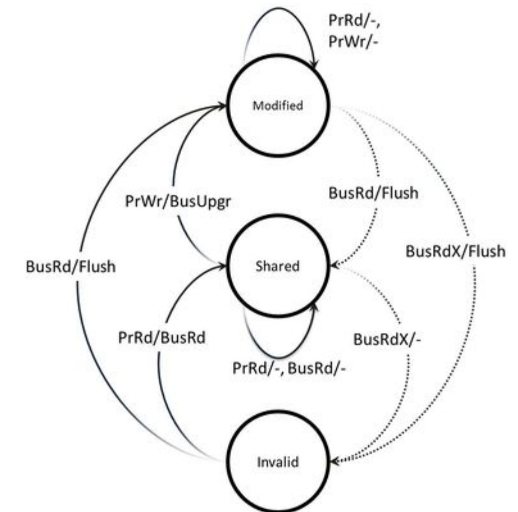
Cache coherence principles



- To perform a write
  - invalidate all readers, or
  - previous writer
- To perform a read
  - find the latest copy

Cache coherence with MESI

- A state diagram
- State (per cache line)
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data



The ultimate goal for scalability

- Possible states
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data

- **Which state is our "favorite"?**

The ultimate goal for scalability

- Possible states
  - **M**odified: the only dirty copy
  - **E**xclusive: the only clean copy
  - **S**hared: a clean copy
  - **I**nvalid: useless data

= threads can keep the data close (L1 cache)

= faster

Experiment

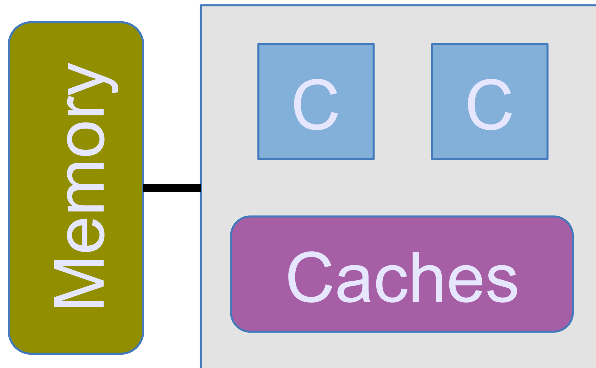
The effects of false sharing

Outline

- CPU caches
- Cache coherence
- **Placement of data**
- Memory consistency
- Hardware synchronization
- Concurrent programming techniques on locks

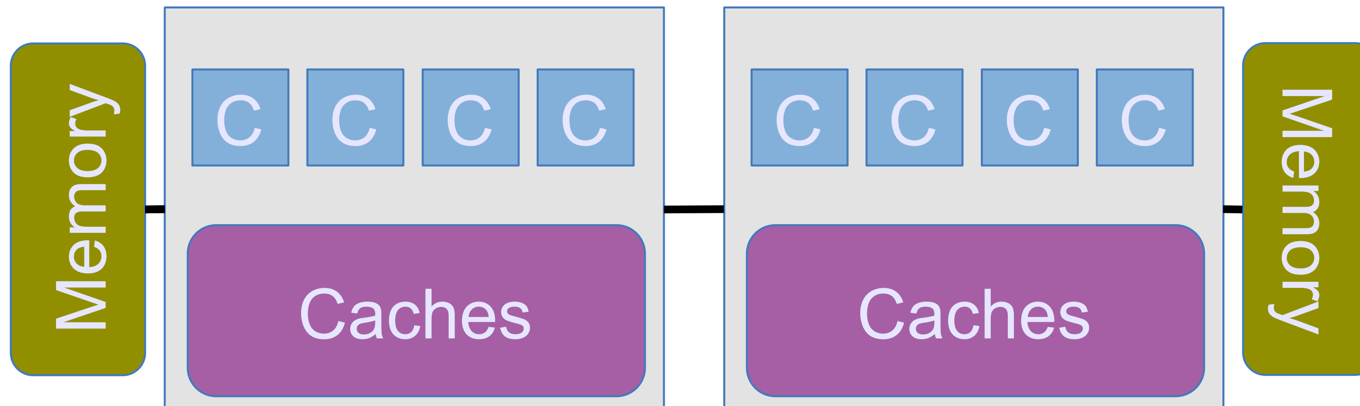
Uniformity vs. non-uniformity

- Typical **desktop** machine



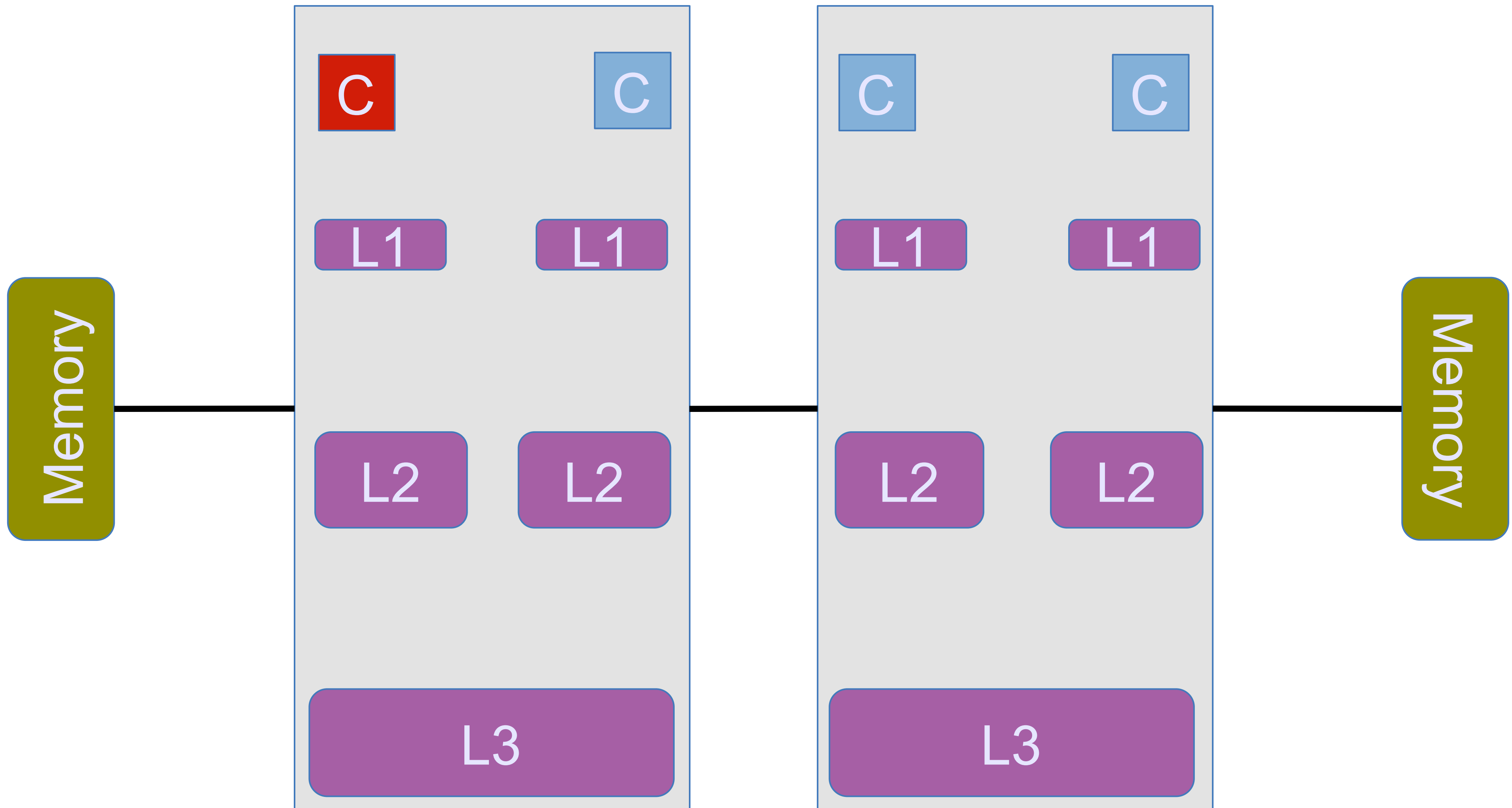
= Uniform

- Typical **server** machine

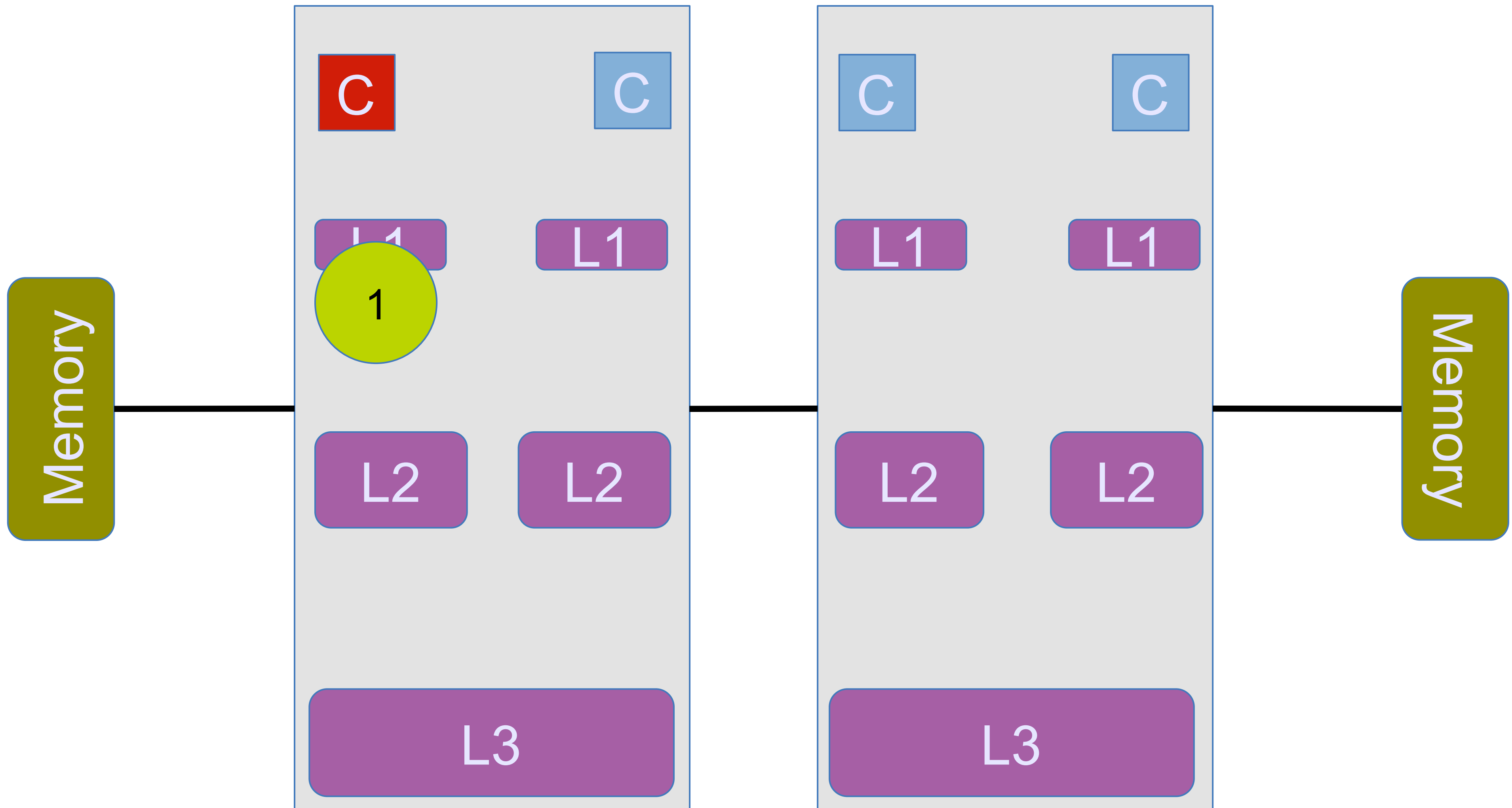


= non-Uniform

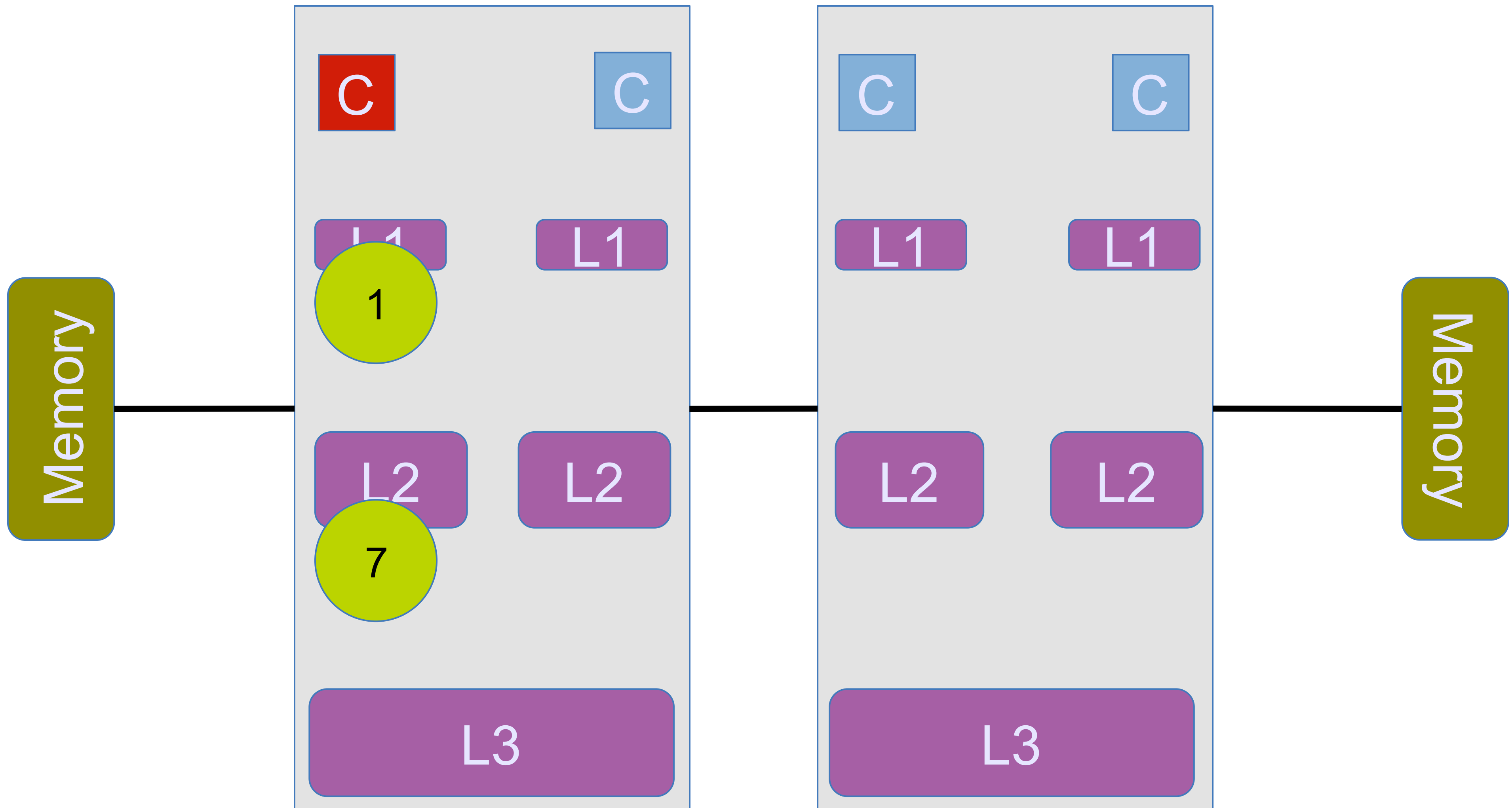
Latency (ns) to access data



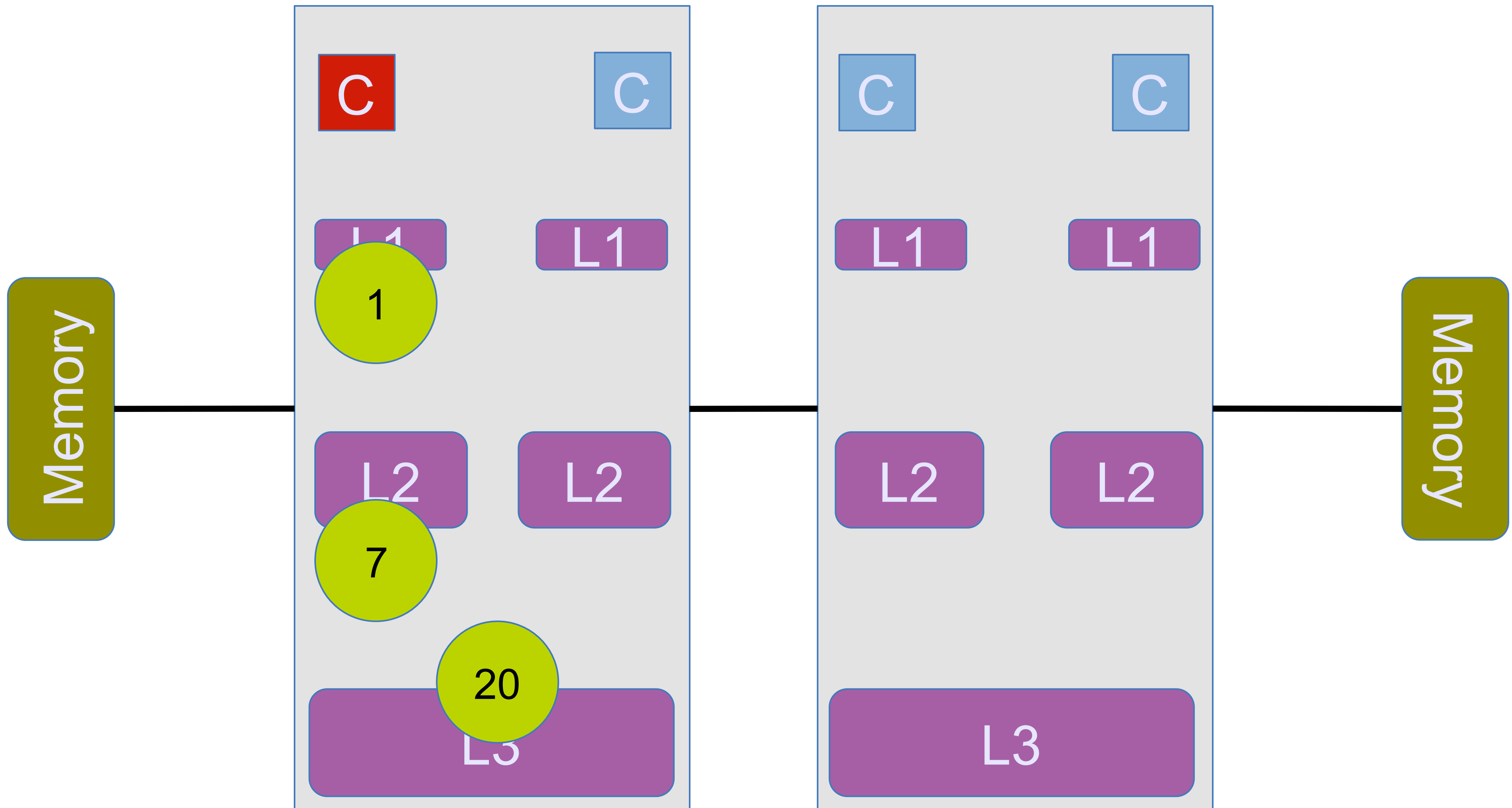
Latency (ns) to access data



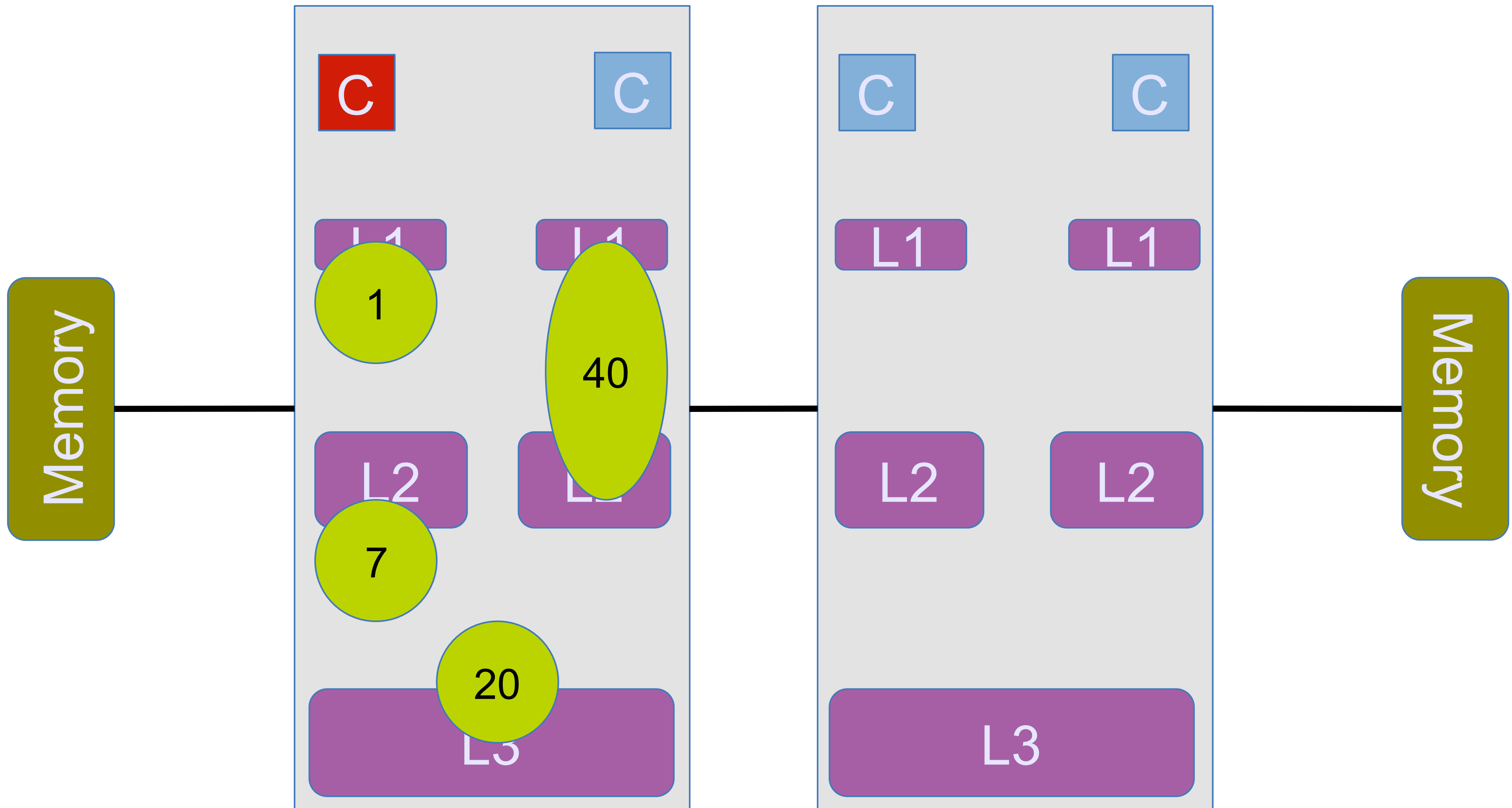
Latency (ns) to access data



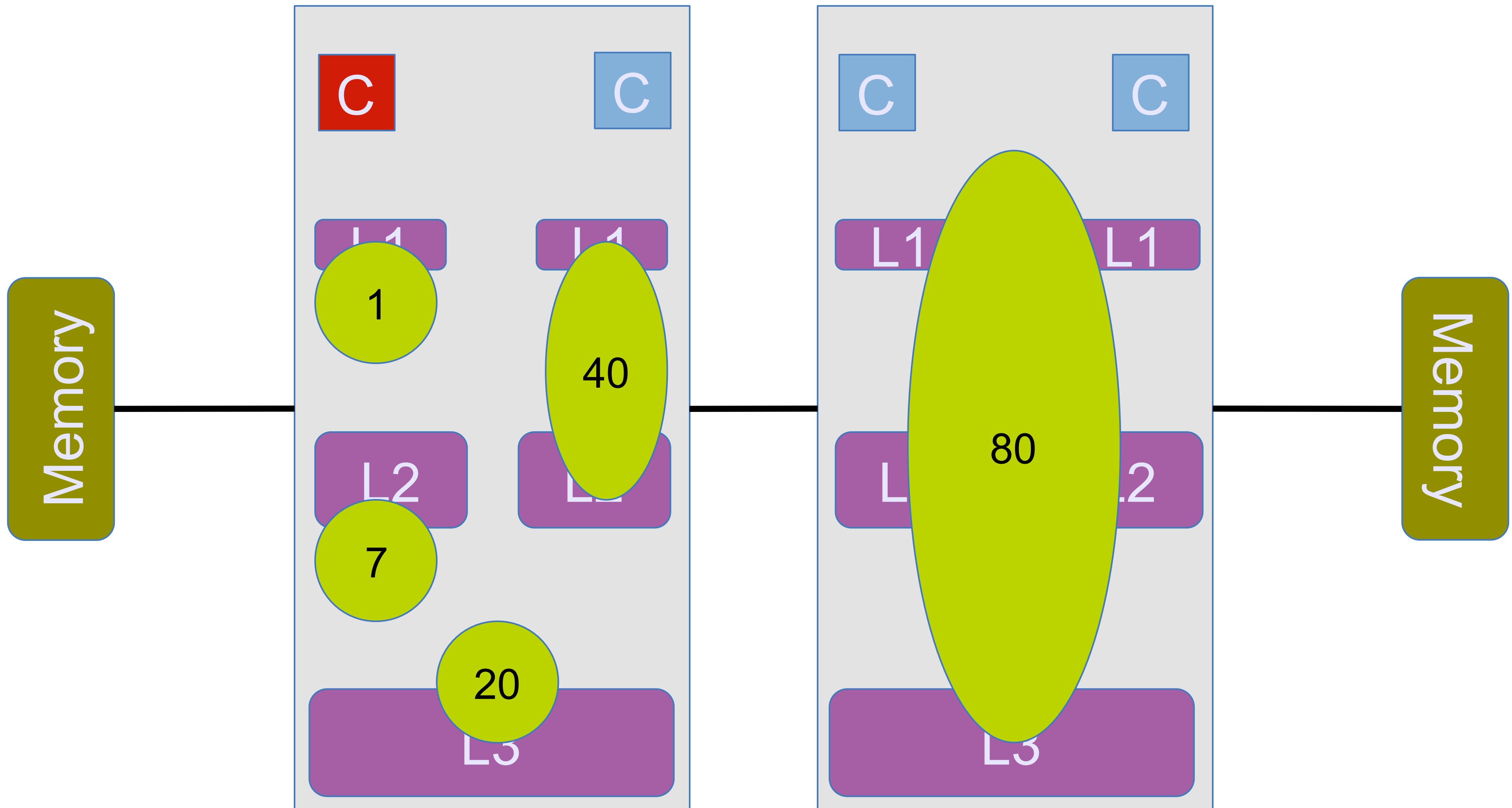
Latency (ns) to access data



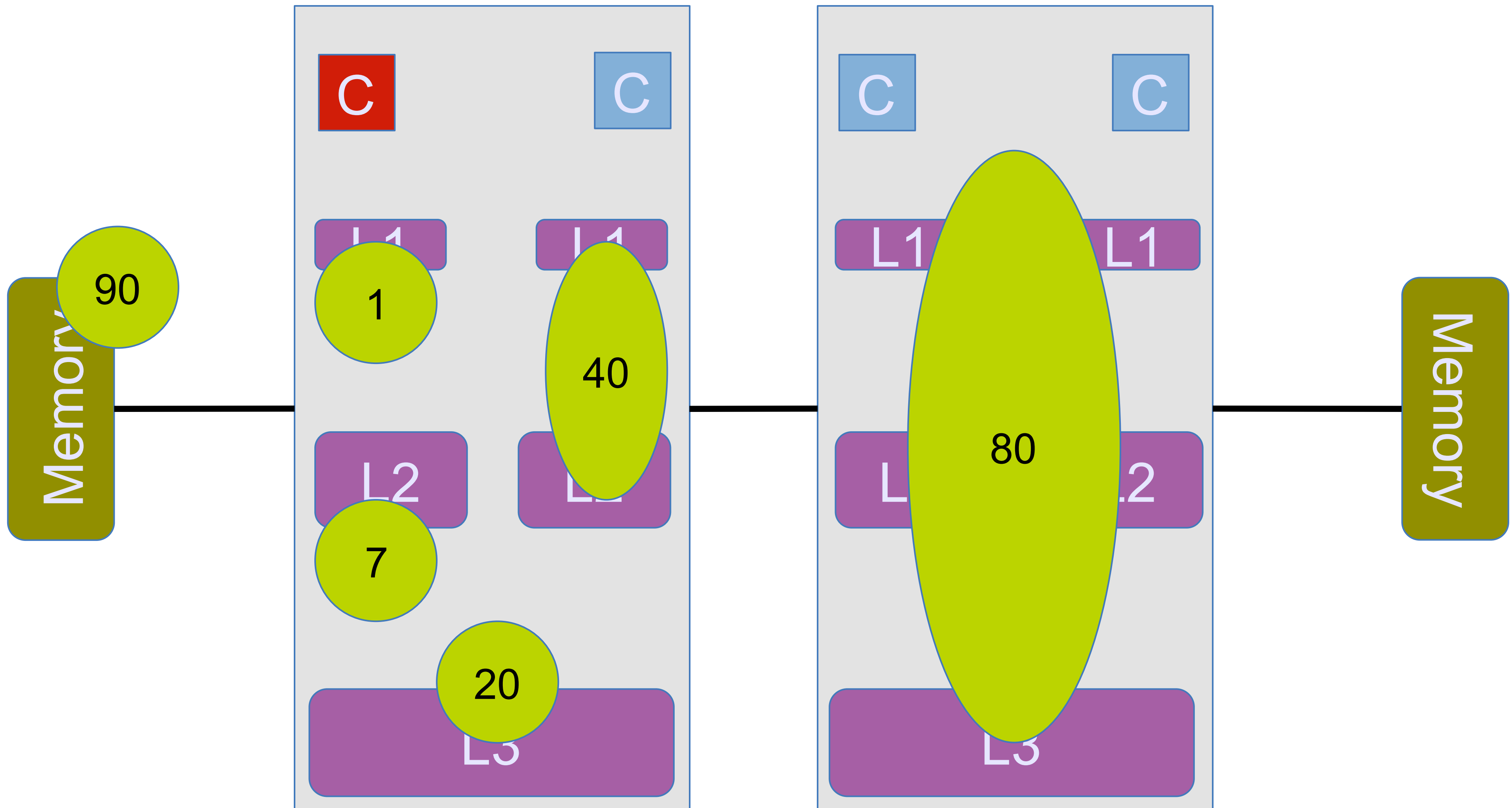
Latency (ns) to access data



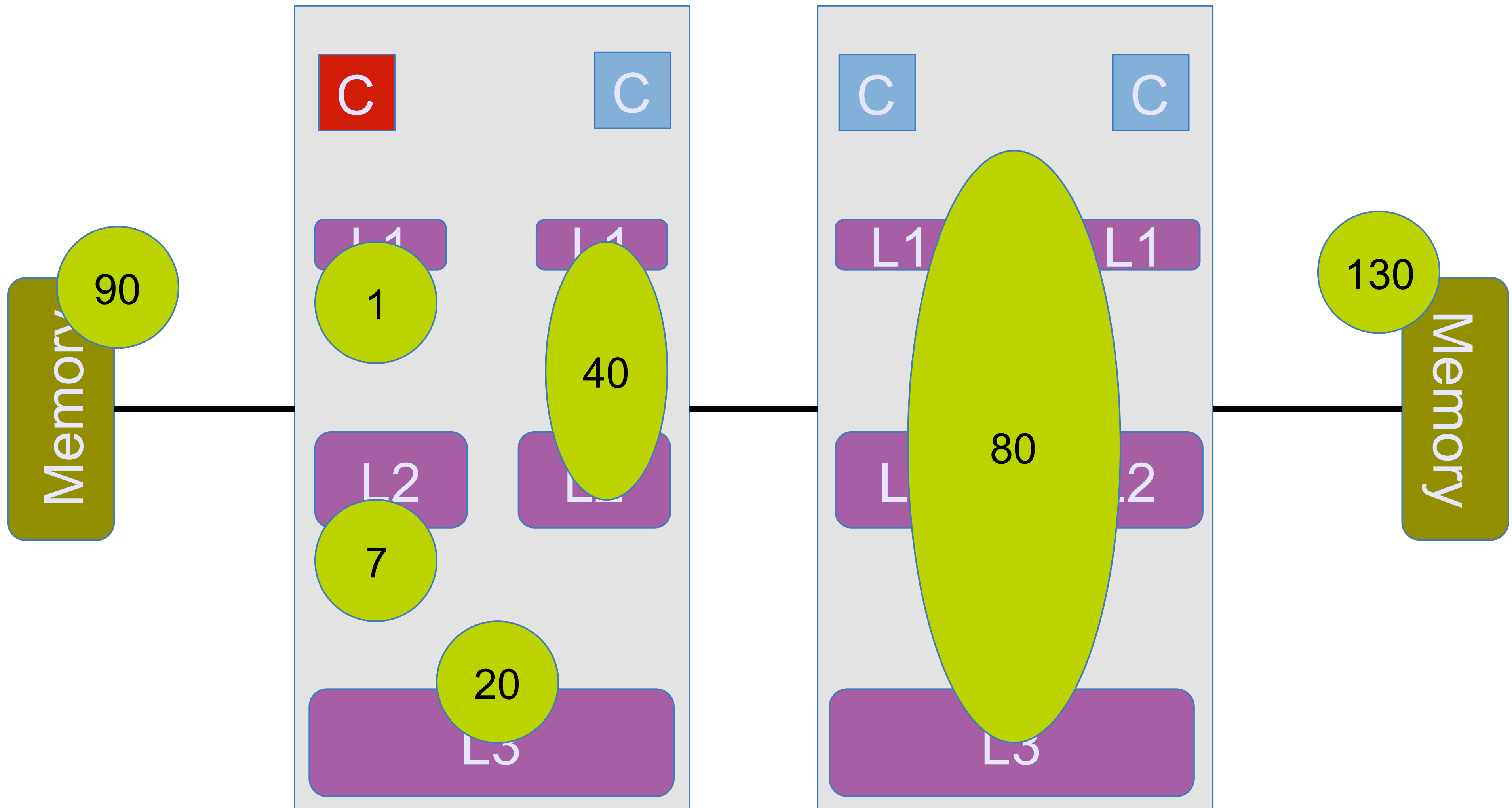
Latency (ns) to access data



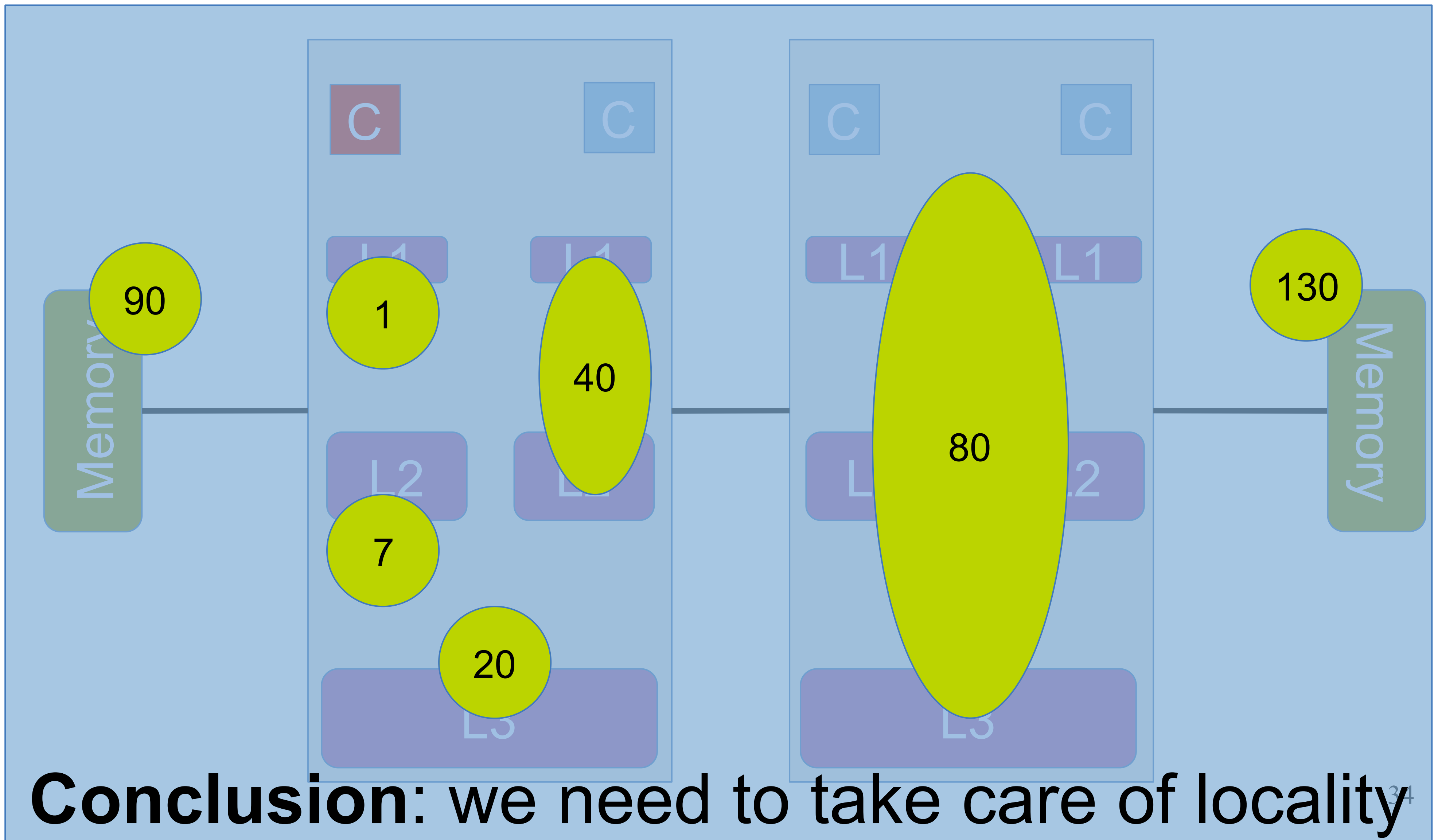
Latency (ns) to access data



Latency (ns) to access data



Latency (ns) to access data



Conclusion: we need to take care of locality

Experiment

The effects of locality