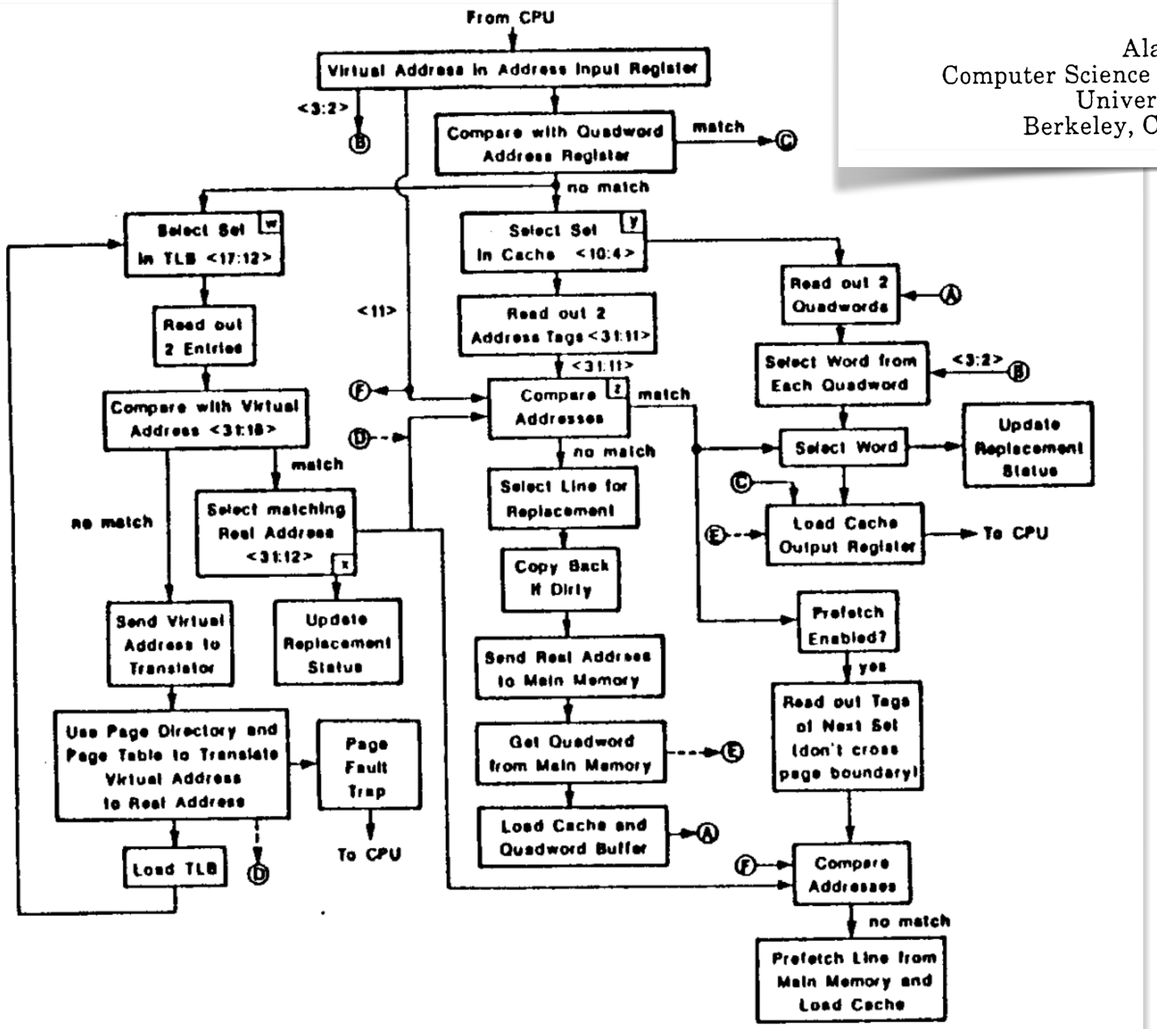
Understanding and Improving Cache Misses

Design of CPU Cache Memories†

Alan Jay Smith
Computer Science Division, EECS Department
University of California
Berkeley, California 94720, USA



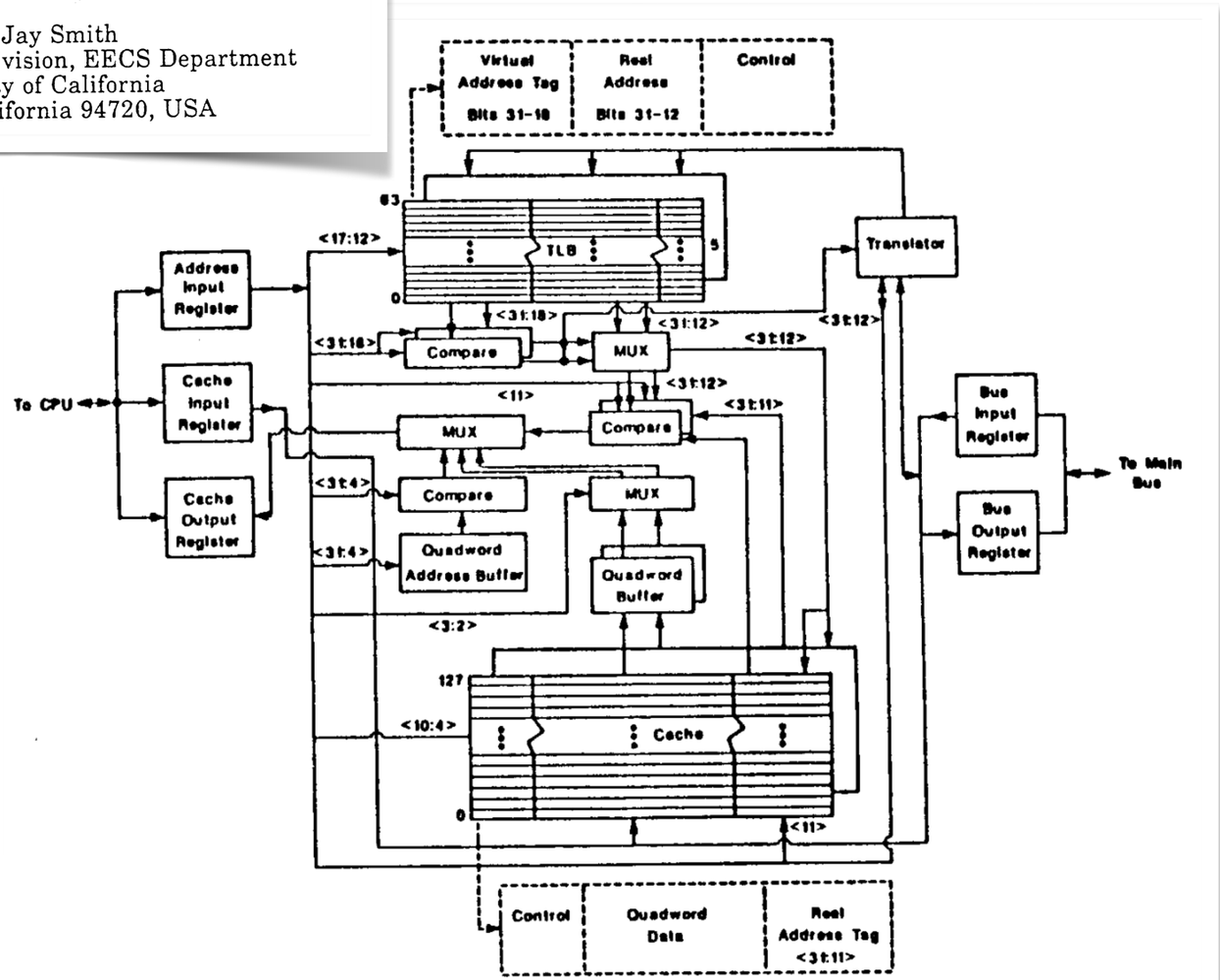


Image credit: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf

Outline

- Understanding cache misses
- Strategies to mitigate misses!
- Introducing the coherence problem…

Understanding Cache Misses

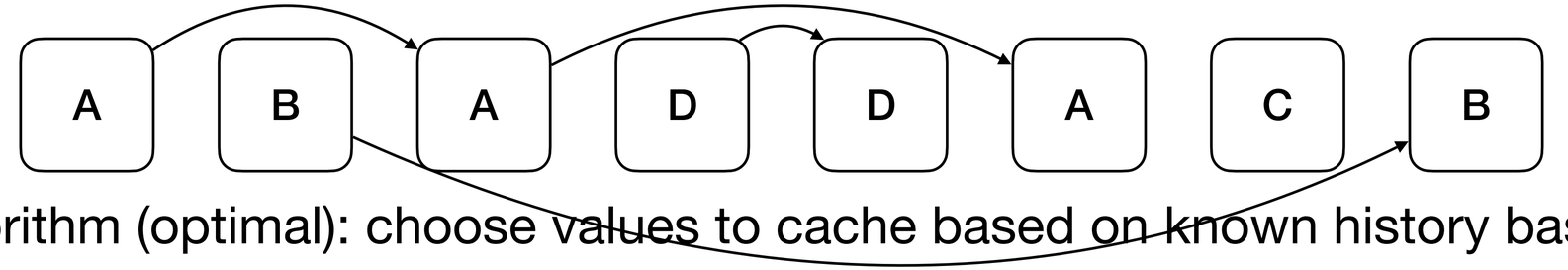
- *Compulsory* misses occur on the first access to a block, so the data cannot yet be in the cache ➡️ these misses are unavoidable!
- *Capacity* misses occur if the cache cannot contain all of the blocked needed during the execution
- *Conflict* misses are a subset of capacity misses in which a set within the cache cannot contain all of the blocks needed during the execution that can be mapped to that set
- The mapping of addresses to cache set is statically defined (i.e., fixed at manufacturing), so cache placement is inflexible ➡️ choosing which blocks to *replace* an incoming block with is dictated by behaviors at runtime!
- Expected memory operation latency: L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * (L3 hit time + L3 miss rate * memory latency))

Principles of Locality

- Applications tend to exhibit *temporal* and *spatial* locality when they are deployed!
- *Temporal locality* describes the likelihood of an application to reuse data within similar periods of time
- *Spatial locality* describes the likelihood of an application to reuse data at similar addresses to one another

Cache Replacement Policies

- Find an eviction target from the candidates within a set
- If an invalid block exists within the set, *greedily* fill that block
- If no valid block exists, choose a target to evict that makes sense relative to the application behavior

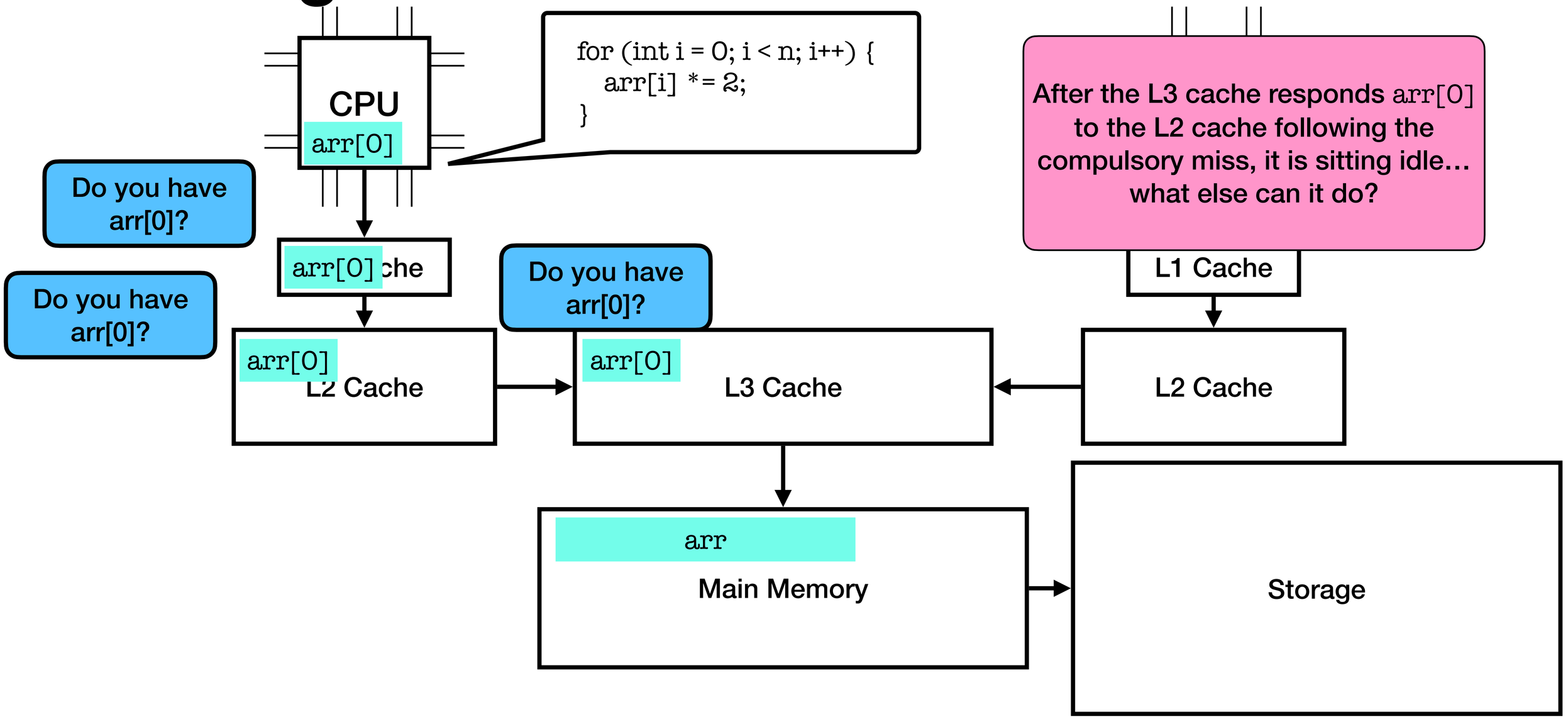


- Belady's algorithm (optimal): choose values to cache based on known history based on *reuse distance*
- In practice, *least recently used* is a heuristic that works well ➡️ you will explore alternatives in HW2 (written)!

Chat with your neighbor(s)!

What mechanisms of our cache construction reap the benefits of application temporal locality? Spatial locality?

Handing Cache Misses

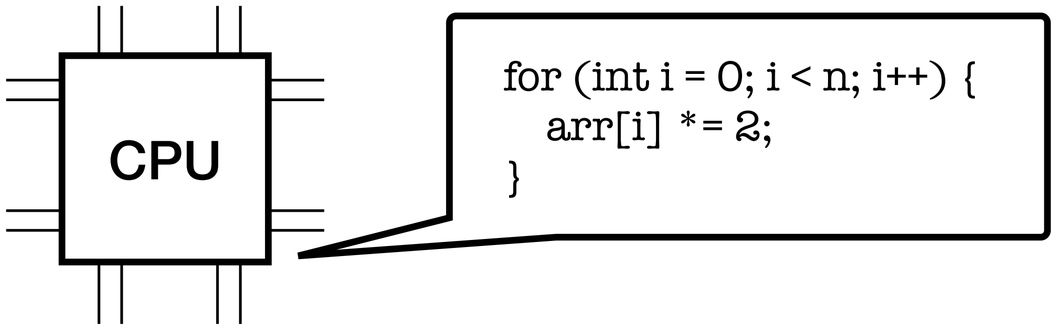


Cache Prefetching

- Given that applications tend to exhibit spatial locality beyond the granularity of a cache block, caches can try to *predict* what the next blocks to use are to avoid misses on requests!
- The notion of fetching some data before it is explicitly requested by the processor is referred to as *cache prefetching*
- Prefetching can either be compiler directed (software prefetching) and/or implemented in the cache logic (hardware prefetching)
- Software prefetching injects more instructions into the software to execute but can help avoid cache misses, so it is likely to benefit performance overall!

Prefetching Strategies

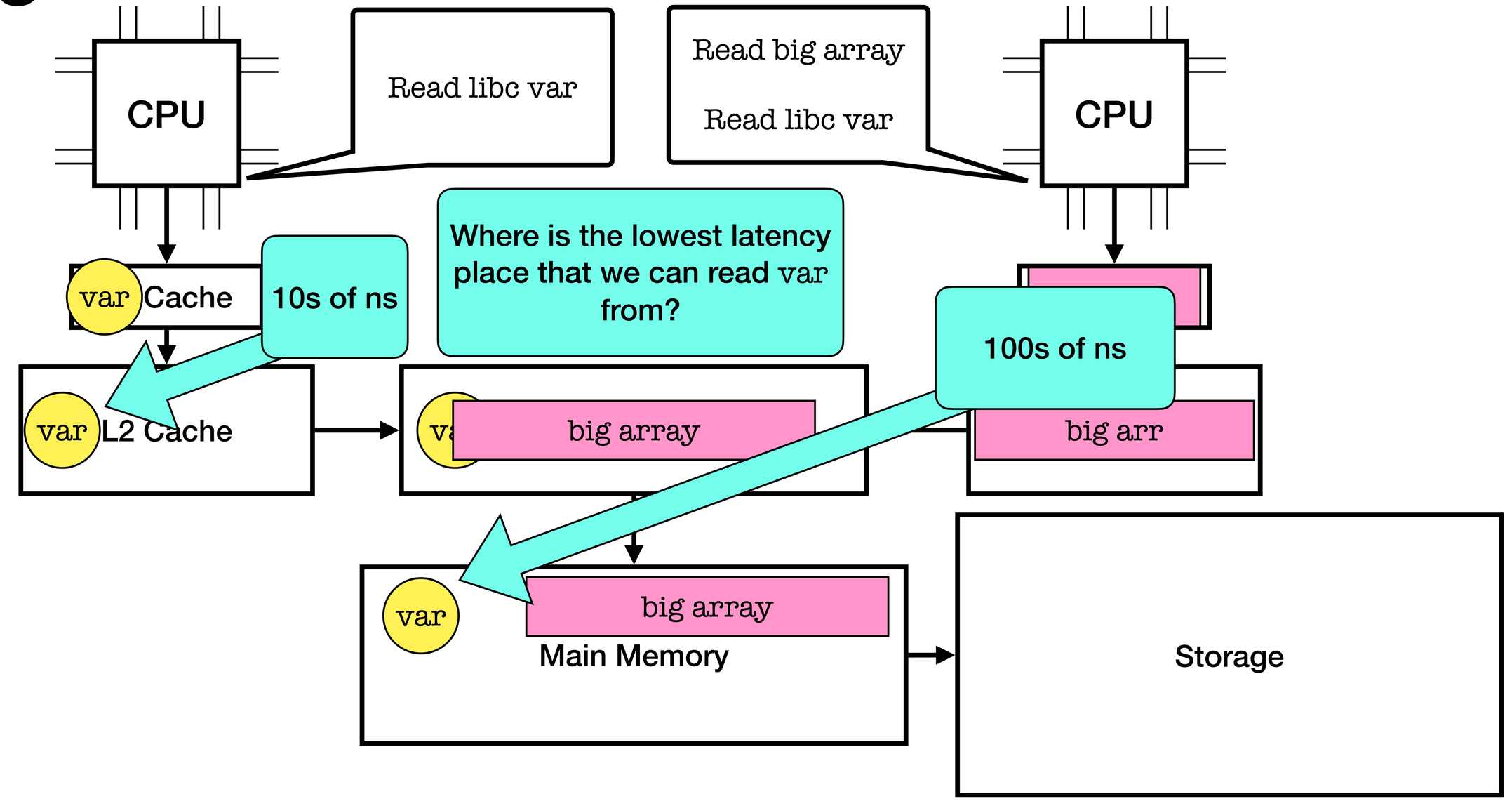
- There are two fields that we likely want to prefetch: the *instructions* and the *data*
- Note that the instructions and data are characterized by their field size ➡️ each data access happens on a *stride length* of the size of an integer (two bytes) whereas instruction *stride lengths* will depend on the ISA
- When prefetching to the caches, the caches make note of patterns of regular stride lengths to inform what to prefetch next



Chat with your neighbor(s)!

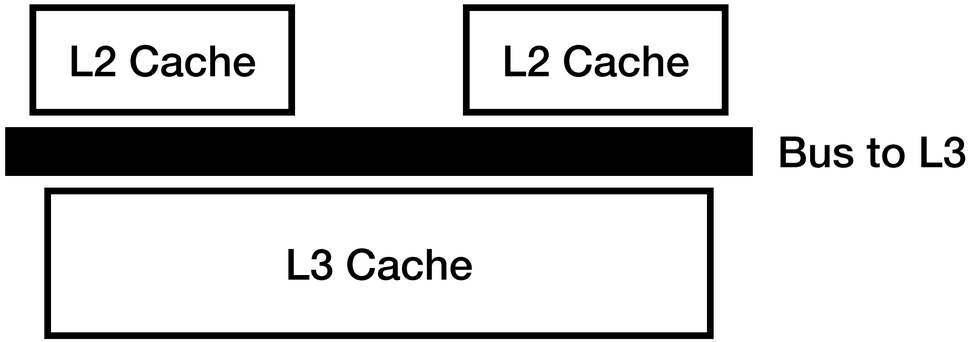
Suppose I have a data storage application where the central storage structure is based on a linked-list. Ideally, what would a cache prefetcher be retrieving from memory in advance of a compulsory miss? Are there any potential pitfalls?

Using Shared Caches



Snooping to Accelerate Lookups

- Caches are connected to one another using collections of wires between ports (i.e., buses)



- Buses between caches can support different kinds of memory commands: caches can query the lower-level memory component directly via a *request* or to other peers on the processor-side of the bus via a *broadcast*

- Broadcasts can be used to implement optimistic *snooping* requests in which a cache asks a peer if they have a shared data value to avoid the longer latency lookup of the lower levels

Chat with your neighbor(s)!

Suppose processors *A* and *B* share a L3 cache. Processor *A* reads data *X* at time t0. Processor *B* reads data *X* at time t1. Processor *A* writes data *X* = *X+1* at time t2, and processor *B* reads data *X* again at time t3. What is the data read by *B* at t3?

Exit ticket!

https://forms.cloud.microsoft/r/z4yfLHSng2