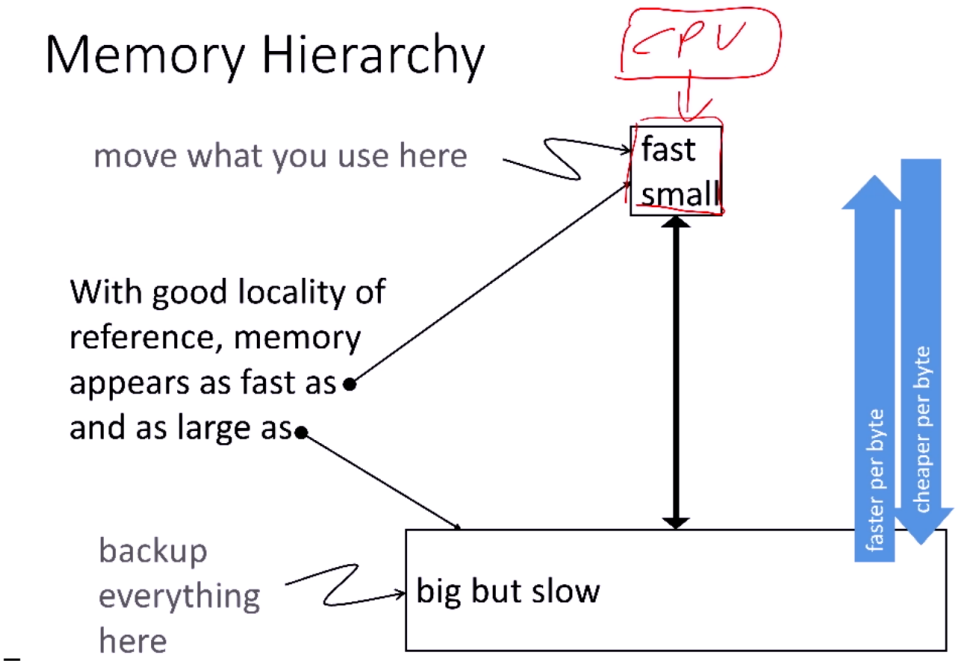
Speculative Execution

- If there is a conditional branch and it is not clear if the branch will be taken or not, the CPU can't fetch any more instructions
- speculate outcome and which branch will be executed
  - store results in ROB
  - trash result if prediction incorrect
- Significant effort is spent by CPUs on learning to predict the branches correctly in an executed program
  - Branch prediction on is done based on execution history: if a branch was taken before, it is likely to be taken again (think of loops!)
- side effects
  - Speculative execution does cause side effects on current CPUs; e.g. instructions that are executed speculatively and trashed affect the timing of actual instructions that are executed later on
  - Timing differences can be exploited in order to make trashed results visible

Memory Hierarchy

- ideal memory
  - Zero access time (latency)
  - Infinite capacity
  - Zero cost
  - Infinite bandwidth (to support multiple accesses in parallel)
  - requirements contradict each other
    - more capacity => slower
    - higher bandwidth => more expensive
    - lower latency => more expensive
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)
- create multiple levels of storage types
  - storage with different properties
    - register file
    - cache
    - DRAM
    - hard disk
  - use fast small as well as big slow memory

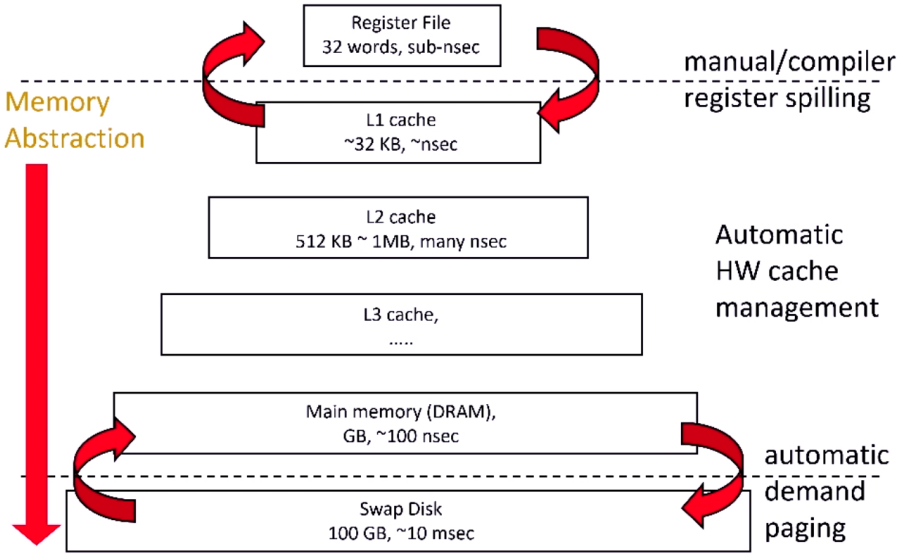


Locality

- use past behaviour to predict near future
  - Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
    - since you are here today, there is a good chance you will be here again and again regularly
  - Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
-
  - A "typical" program has a lot of locality in memory references
    - typical programs are composed of "loops"
  - Temporal: A program tends to reference the same memory location many times and all within a small window of time
  - Spatial: A program tends to reference a cluster of memory locations at a time
-

Automatic Memory Management

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon
- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- automatically manage/move data across levels
  - handled manually
    - registers
    - between DRAM and hard disk
  - modern memory hierarchy



Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i. The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

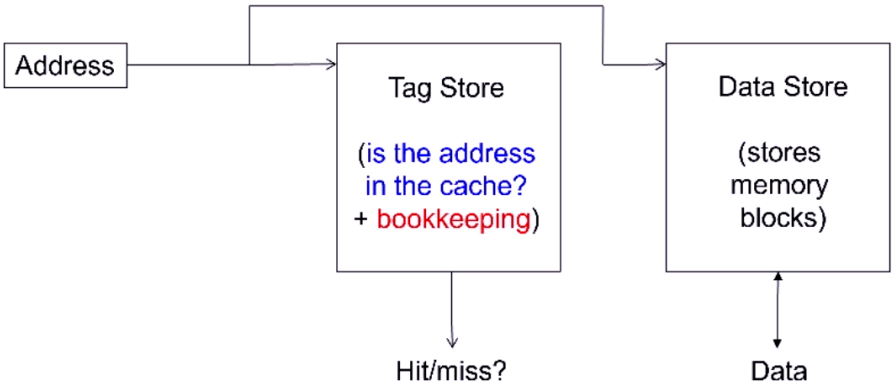
$$T_i = t_i + m_i \cdot T_{i+1}$$

h_i and m_i are defined to be the hit-rate and miss-rate

Cache

- Generically, any structure that "memorizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- design decisions
  - ❑ Placement: where and how to place/find a block in cache?
  - ❑ Replacement: what data to remove to make room in cache?
  - ❑ Granularity of management: large or small blocks? Subblocks?
  - ❑ Write policy: what do we do about writes?
  - ❑ Instructions/data: do we treat them separately?
- abstraction



- implementation
  - divide RAM into 2^n subsets
    - store one subset in cache
    - each subset can be identified with n-bit tag

        - first n-bits of RAM address equal to tag
    - repeated between cache layers
        - L1 cache
        - L2 cache
        - …
  - actually multiple subset in mutliple cache locations
    - map first k subsets to first cache location
        - repeated for following subsets
  - address converted to
    - tag
    - index
    - offset
  - actual workflow for data access
    - use cache location with index
        - check if its tag equal to address tag
        - if yes, use offset to access data
- example
  -