18-747 Lecture 11: High-Performance Memory Hierarchies

James C. Hoe
Dept of ECE, CMU
October 3, 2001

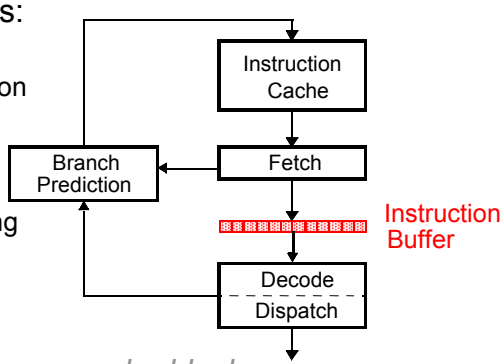
Reading Assignments: *S&L Ch 3 82-107*

Announcements: Midterm Exam on Monday 10/15

Handouts: *Handout #7: HW1 Solution*
Handout #8: Project 0 Solution
Graded HW1

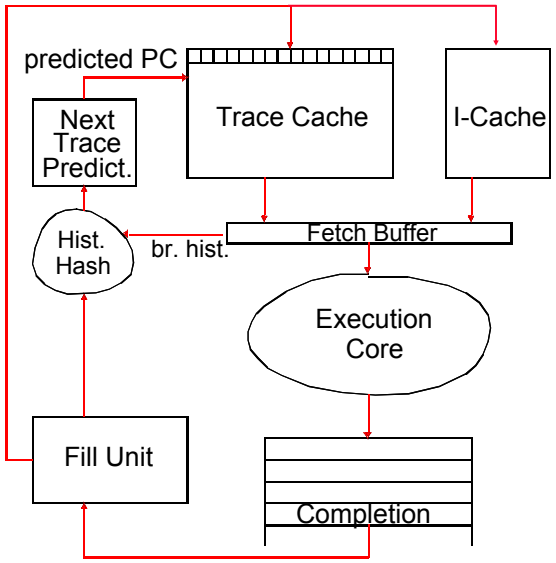
Wide Instruction Fetch Issues

- Average Basic Block Size
  - integer code: 4-6 instructions
  - floating-point code: 6-10 instructions
- Three Major Challenges:
  - Multiple-Branch Prediction
  - Multiple Fetch Groups
  - Alignment and Collapsing



Cannot be solved with just longer cache blocks

A Typical Trace Cache Organization



Trace Fill Unit

- Observe the dynamic execution sequence
- Gather instructions into a trace segment (or trace cache block)
- Some simple heuristics for forming trace segments
  - stop after collecting up to *N* instructions
    (*N* is the trace cache block size)
  - stop after *B* conditional branches
    (*B* is the limit of the multi-branch predictor)
  - stop after seeing an register-indirect jump
  - Don't split basic blocks
  - In some designs, unconditional and conditional branches can be dropped from the traces
- Can include pre-decoded dependence information
- Can even dynamically re-order instructions (don't need an out-of-order core!!)

Trace Selection/Prediction

- Basic
  - find the trace that starts at the predicted next-PC
- Multiple cached traces may have the same starting PC
  - difference is in the internal branch decisions

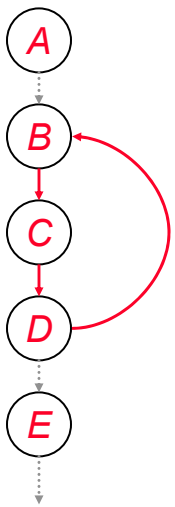
  ⇒ *need multi-branch predictors*
- Partial Traces
  - predicted next-PC points to the middle of a cached trace (cached ABC, but predicted BC)
  - multi-branch prediction may say not to use the entire length of a cached trace (cached ABC, but only needs AB)

  ⇒ *need alignment and collapsing buffer*

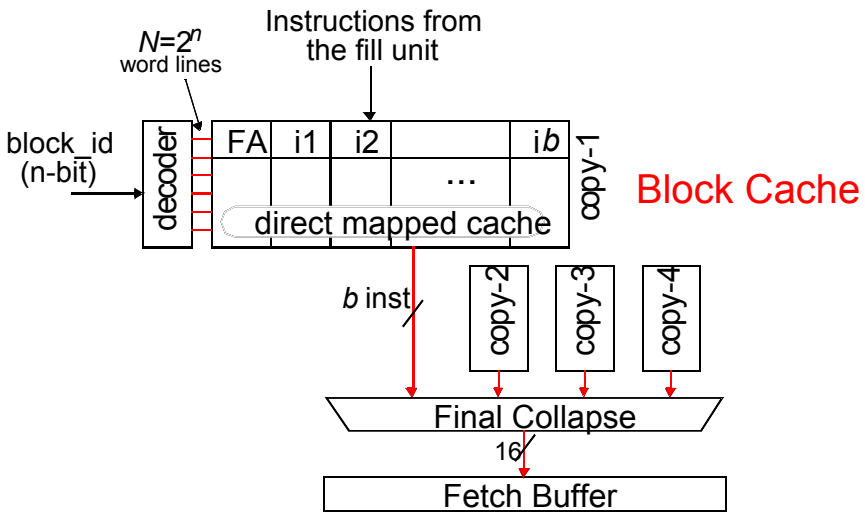
So how is this better?

Redundant Traces

- Suppose B,C and D are instructions in a loop
  - 3 different traces of 3-instructions are possible
  - Which one should we keep in the trace cache?
  - How do we detect the beginning and the end of basic blocks?
- Suppose A,B,C,D and E are basic blocks
  - don't cache BC if BCD is cached
  - what about CDB and CDE?
  - what about ABC and DBC?
  - How to cut down on redundant instruction storage?

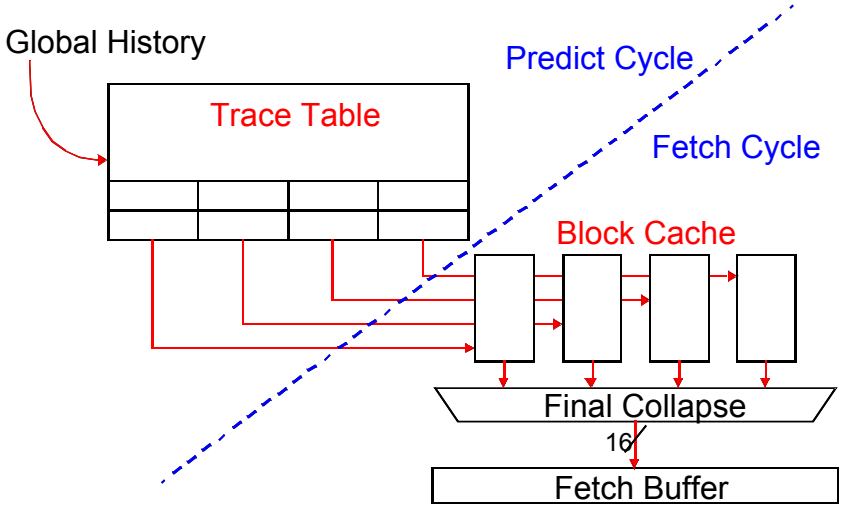


Replicated Block Cache



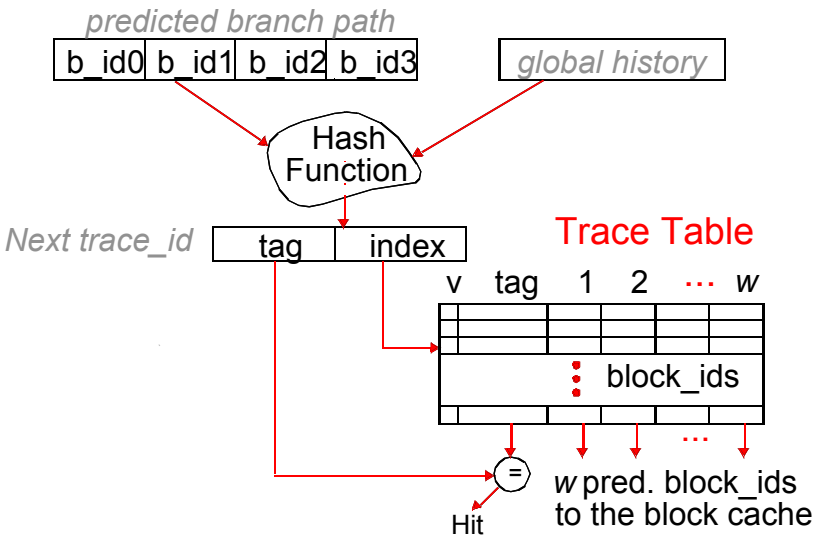
What about fragmentation?

Predict and Fetch Trace

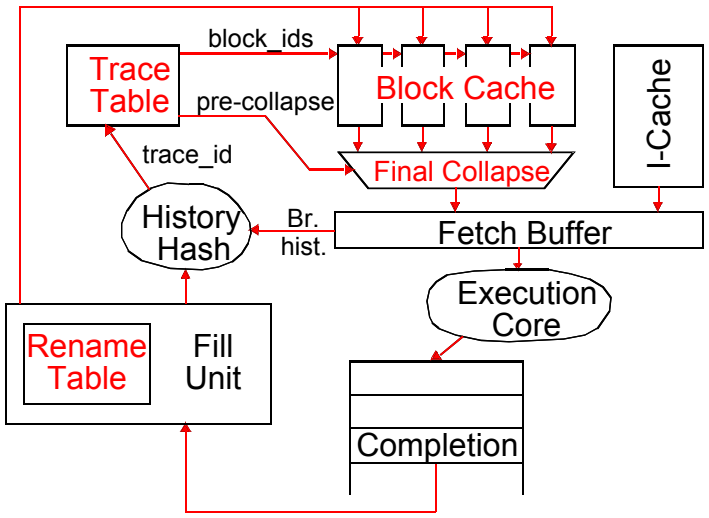


More efficient: redundancy is in the trace table and not the block cache

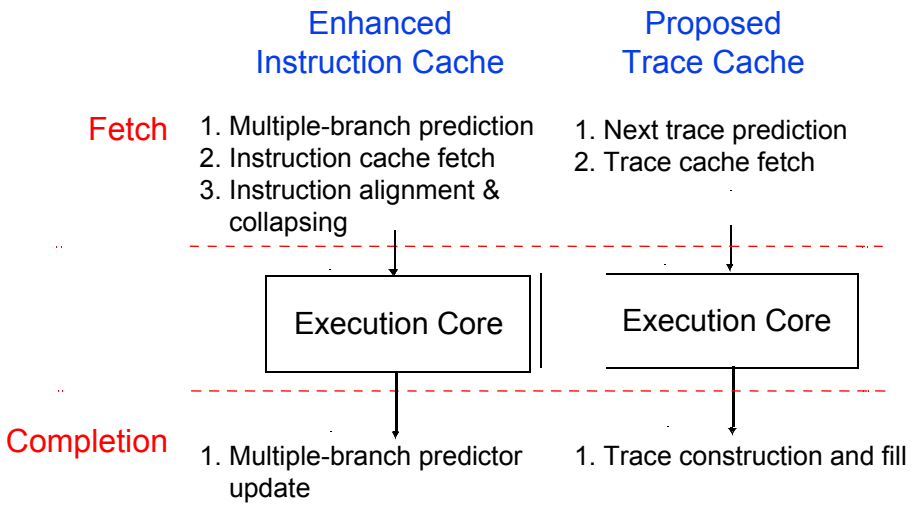
Next Trace Prediction



The Block-Based Trace Cache



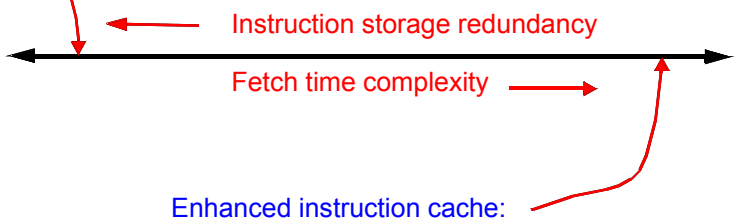
Wide-Fetch I-cache vs. T-cache



Trace Cache Trade-offs

Trace cache:

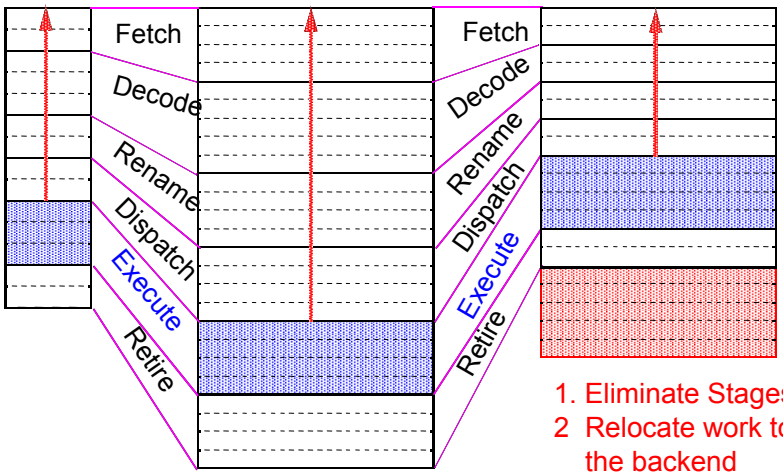
Pros → Moves complexity to backend
Cons → Inefficient instruction storage



Enhanced instruction cache:

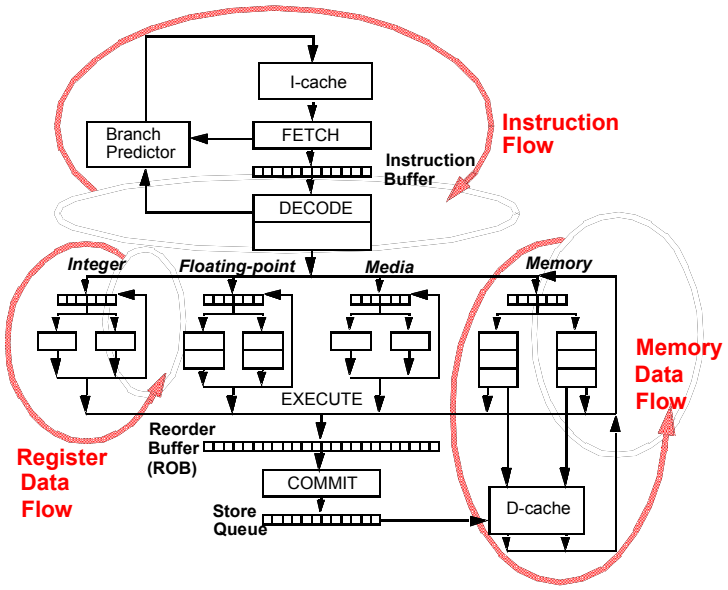
Pros → Efficient instruction storage
Cons → Complexity during fetch time

As Machines Get Wider (… and Deeper)

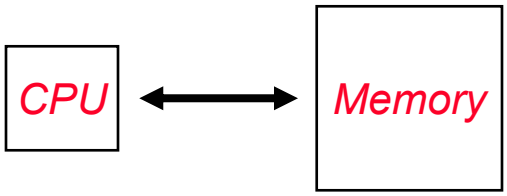


1. Eliminate Stages
2. Relocate work to the backend

Flow Path Model of Superscalars

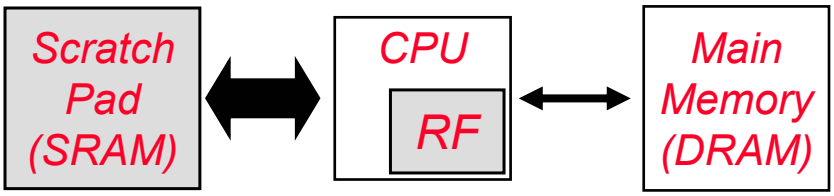


CPU-Memory Bottleneck



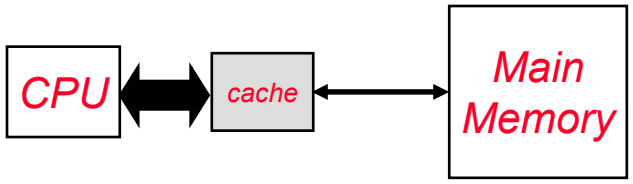
- Performance of high speed computers is usually limited by memory performance, *bandwidth & latency*
- Main memory access time >> Processor cycle time
  over 100 times difference!!
- if *m* fraction of instructions are loads and stores
  then average '1+*m*' references per instruction
  suppose *m*=40%, IPC=4@1GHz ⇒ 22.4 GByte/sec

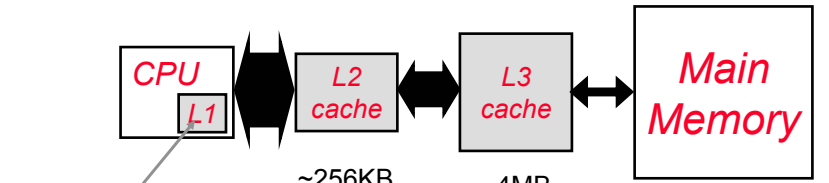
How to Incorporate Faster Memory



- SRAM access time << Main memory access time
- SRAM bandwidth >> Main memory bandwidth
  ⇒ *SRAM is expensive*
  ⇒ *SRAM is smaller than main memory*
- Programs exhibit temporal locality
  - frequently-used data can be held in the scratch pad
  - the cost of the first and last memory access can be amortized over multiple reuse
- Programs must have a small *working set (aka footprint)*

Caches: Automatic Management of Fast Storage





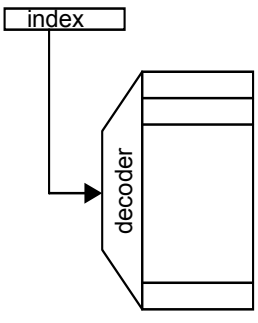
16~32KB
1~2 pclk latency

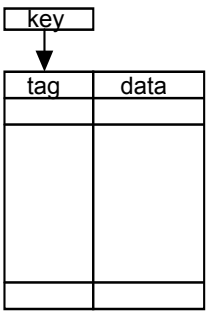
~256KB
~10 pclk latency

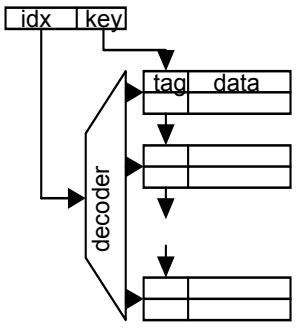
~4MB

~50 pclk latency

Cache Memory Structures







Indexed Memory

k-bit index
2^k blocks

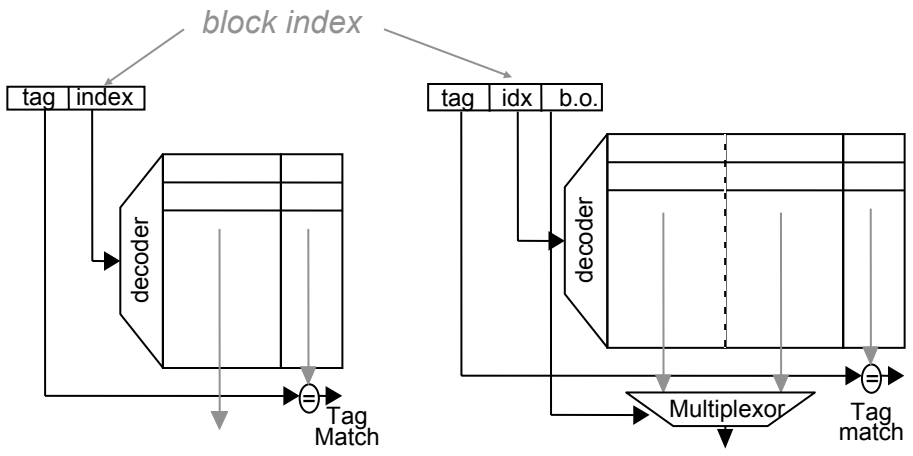
Associative Memory (CAM)

no index
unlimited blocks

N-Way Set-Associative Memory

k-bit index
$2^k \cdot N$ blocks

Direct Mapped Caches



Cache Block Size

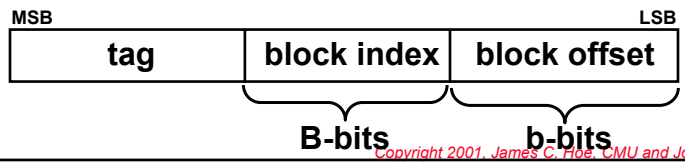
- Each cache block or (cache line) has only one tag but can hold multiple "chunks" of data
  - reduce tag storage overhead

    In 32-bit addressing, an 1-MB direct-mapped cache has 12 bits of tags

    4-byte cache block ⇒ 256K blocks ⇒ ~384KB of tag

    128-byte cache block ⇒ 8K blocks ⇒ ~12KB of tag
  - the entire cache block is transferred to and from memory all at once

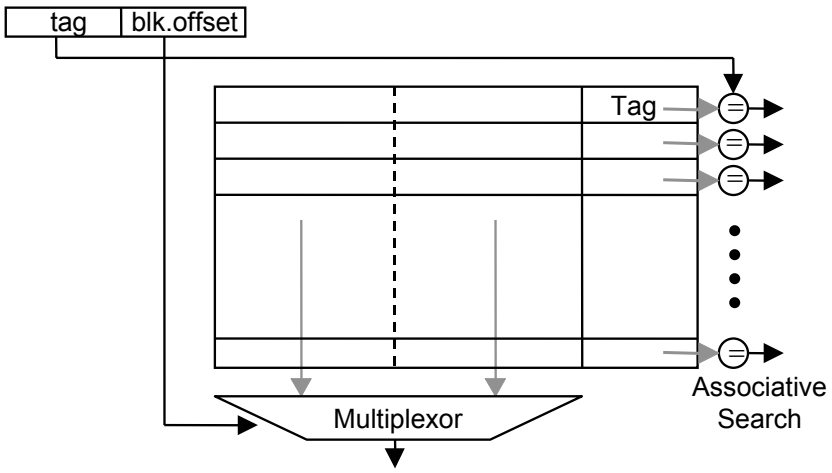
    good for spatial locality since if you access address i, you will probably want i+1 as well (prefetching effect)
- Block size = 2^b; Direct Mapped Cache Size = 2^{B+b}



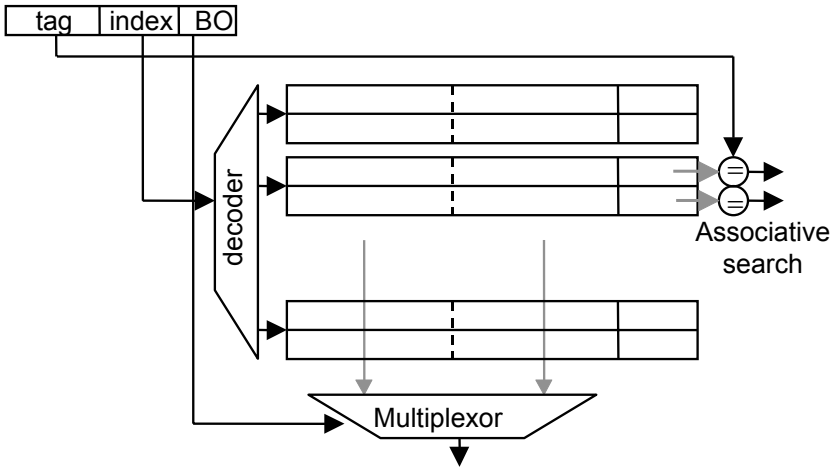
Large Blocks and Subblocking

- Large cache blocks can take a long time to refill
  - refill cache line *critical word first*
  - restart cache access before complete refill
- Large cache blocks can waste bus bandwidth if block size is larger than spatial locality
  - divide a block into subblocks
  - associate separate valid bits for each subblock.

Fully Associative Cache

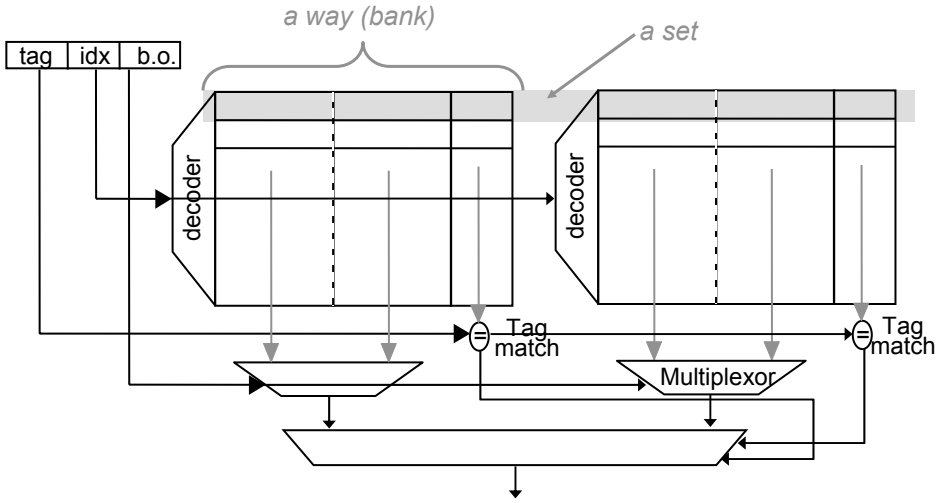


N-Way Set Associative Cache



Cache Size = N x 2^{B+b}

N-Way Set Associative Cache



Cache Size = N x 2^{B+b}

Principle Behind Hierarchical Storage

- Each level memoizes values stored at lower levels
- Instead of paying the full latency for the "furthermost" level of storage each time

  Effective Access $T_i = h_i \cdot t_i + (1 - h_i) \cdot T_{i+1}$

  - where h_i is the 'hit' ratio, the probability of finding the desired data memoized at level i
  - t_i is the raw access time of memory at level i
- Given a program with good locality of reference

  $S_{working\text{-}set} < s_i \Rightarrow h_i \approx 1 \Rightarrow T_i \approx t_i$

- A balanced system achieves the best of both worlds
  - the performance of higher-level storage
  - the capacity of lower-level low-cost storage.