

CS 610: A Quick Refresher on Cache Memory

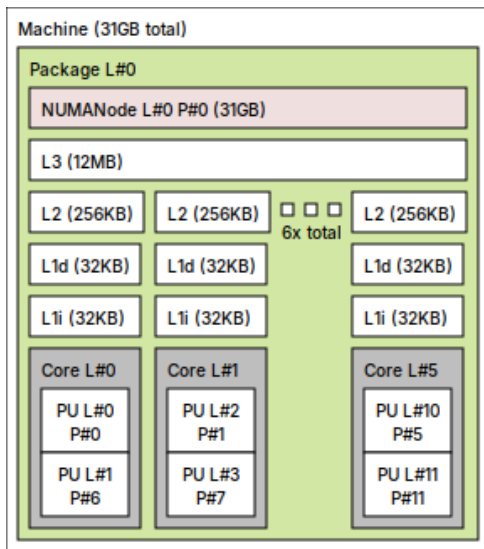
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I



```
lstopo --output-format png --no-io > cpu.png
```



Let us compare the performance!

```
#define T (1024 * 1024)
double A[N][N];
for (it = 0; it < T; it++)
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            A[i][j] += 1;
```

```
#define N 32
#define T (1024 * 1024)
```

100 ms

```
#define N 128
#define T (1024 * 1024)
```

150 ms

```
#define N 256
#define T (1024 * 1024)
```

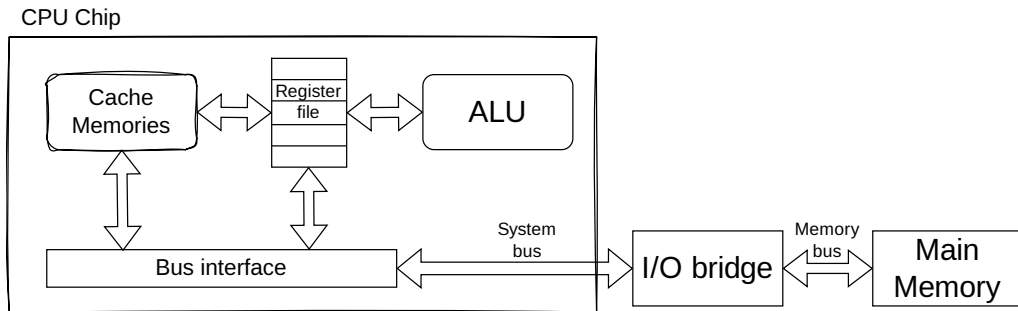
500 ms

```
#define N 4096
#define T (1024 * 1024)
```

750 ms

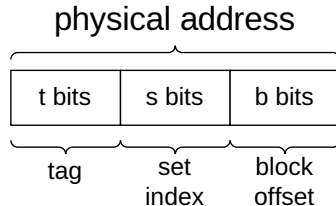
Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware and hold frequently accessed blocks of main memory
 - ▶ CPU looks for data in caches (e.g., L1, L2, and LLC) first, and then in main memory
 - ▶ Because of **locality**, programs tend to access the data at level k (higher) more often than they access the data at level $k + 1$



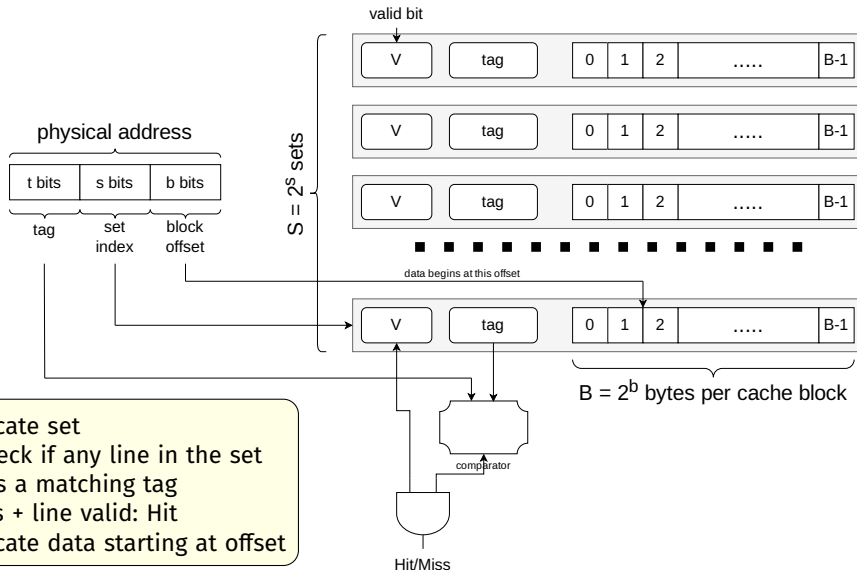
Cache Organization and Lookup

- Caches are organized as arrays of cache lines (or blocks) containing program data
- Each cache line holds contiguous bytes of data (e.g., 64 or 128 bytes)
- Let us assume for now that caches are addressed using physical addresses (e.g., 40 bits)



- **Set** index bits identify the desired line(s) we should search for looking up the data
- **Block** offset bits identify the starting location of the requested data in a cache line
- **Tag** bits check for an exact match with the address to be looked up

Direct-Mapped Cache



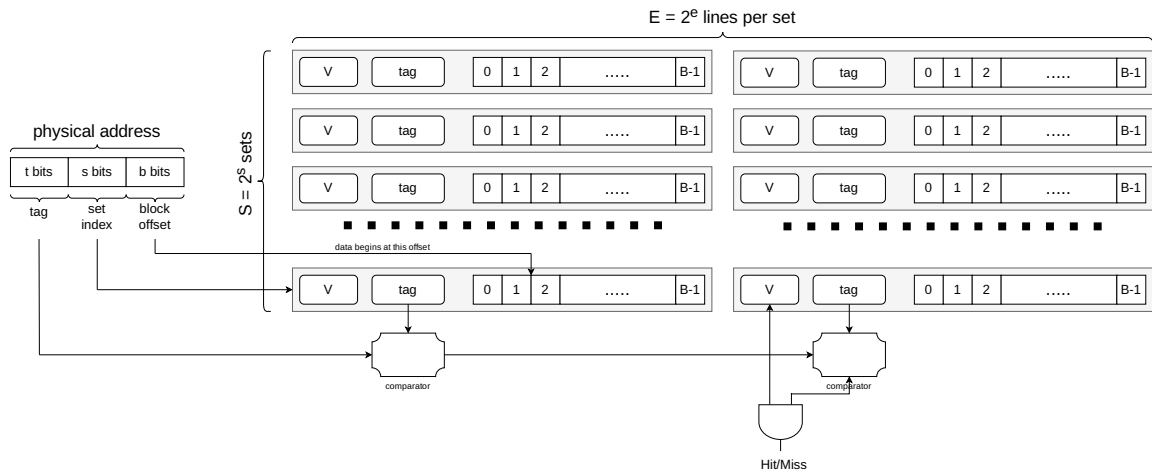
- (i) Locate set
- (ii) Check if any line in the set has a matching tag
- (iii) Yes + line valid: Hit
- (iv) Locate data starting at offset

Addressing a Cache

- Consider a system with 32-bit physical address, 32 KB direct-mapped cache with 64 byte blocks
- $b = 6$ bits, 512 cache lines or sets, $s = 9$ bits
- Hence, number of tag bits $t = 17$
- Each cache line contains 64 byte data, 17-bit tag, one valid/invalid bit, and additional state bits (e.g., dirty)
- Tag and index bits have been extracted from the physical address, so the cache is physically-indexed physically-tagged (PIPT)

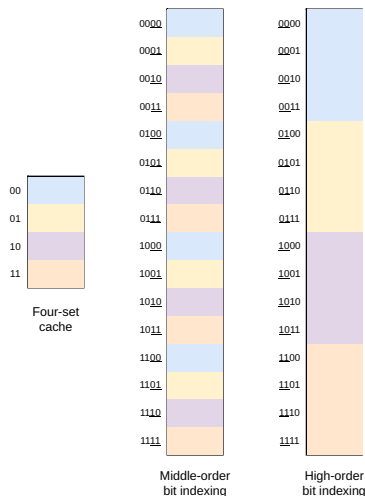
E-Way Set Associative Cache (E=2)

Set-associative caches reduce conflict misses by maintaining E lines per set



Why are index bits not the high order bits?

- Using the high-order address bits for indexing is **inefficient**
 - ▶ A larger address range needs to be stepped over for the high-order index bits to change
 - ▶ Will imply collision among closely-located memory blocks



Dealing with Writes

Possibilities on a hit

Write through Write immediately to memory

Write back Defer write to memory until replacement of the line

- Need a dirty bit to indicate that the line has been updated

Possibilities on a miss

Write allocate Load into the cache and then update the line

- Good if more writes to the location follow

No-write allocate Writes straight to memory, does not load into cache

Typical setup

- Write back + Write allocate
- Write through + No-write allocate

Evaluating Cache Performance

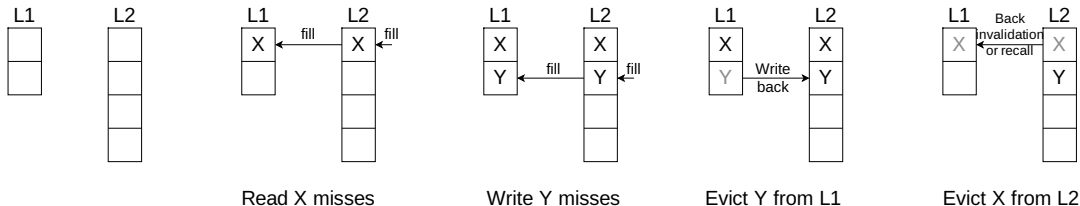
- Hit time**
 - Time to deliver a line in the cache to the processor, including the time to determine whether the line is in the cache
 - Reduce hit time: small direct-mapped structures, overlap or avoid address translation when indexing the cache, and way prediction
- Miss rate**
 - Fraction of memory references not found in the cache (misses/access)
 - Reduce miss rate: larger caches, larger block sizes, greater associativity, and compiler optimizations
- Miss Penalty**
 - Time taken for a cache miss to complete
 - Reduce miss penalty: multilevel caches, nonblocking caches, victim cache, early restart, critical word first, fill before spill, prefetching (hardware or software)

Average Memory Access Time

$$AMAT = time_{hit} + prob_{miss} * penalty_{miss}$$

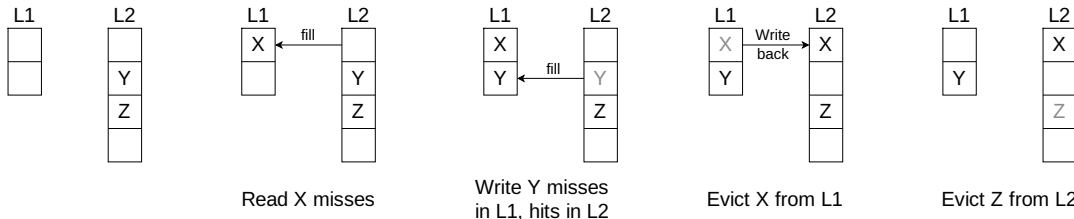
Inclusion Policy in Cache Hierarchy

- In an **inclusive** hierarchy, all cache lines present at level i are also present in level $i + 1$
 - ▶ A L1 and L2 miss will require fetching the line in both the caches
 - ▶ Eviction of a L2 line will fetch the latest copy from L1 and invalidate the L1 line (if present)
 - ▶ Inclusion lengthens miss handling but simplifies write back
 - ▶ Intel Core i7-6700 (codename Skylake-S) uses inclusive private caches



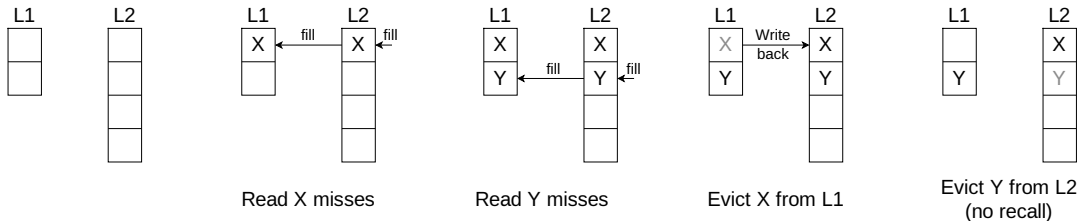
Inclusion Policy in Cache Hierarchy

- In an **exclusive** hierarchy, the lower level cache contains only blocks that are not present in the higher level cache
 - ▶ L2 cache is filled when a L1 line is evicted
 - ▶ L2 in AMD Opteron is exclusive



Inclusion Policy in Cache Hierarchy

- In a **non-inclusive non-exclusive** (NINE) cache, the contents of the lower-level cache are neither strictly inclusive nor exclusive of the higher-level cache
 - ▶ The L3 in AMD Opteron is NINE

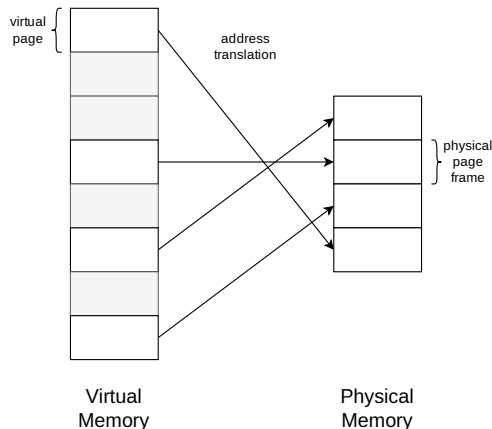


Virtual Memory

Need for Virtual Memory

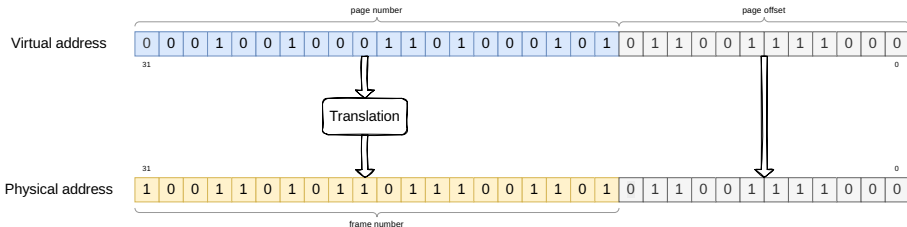
- + Provides an illusion of larger memory
- + Reduces application start-up time
- + Supports multiprocessing by allowing for per-process privileges

- A processor generates virtual addresses while memory is physically addressed
 - ▶ Requires a virtual-to-physical address translation



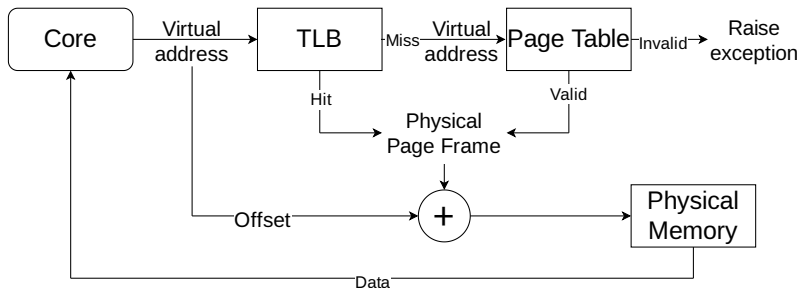
Address Translation

- Virtual address VA is split into two parts: page number (VPN) and the page offset
 - ▶ Given a 4 KB page with 32-bit virtual addresses, the low 12 bits are the page offset, and the remaining 20 bits are the VPN
- The VPN is mapped to a physical page frame number (PFN) using the **page table**
 - ▶ Each page table entry (PTE) includes valid, access permissions, and other state bits
 - ▶ A **page fault** occurs when the valid bit is reset (i.e., no physical page in memory)
 - The kernel allocates a new physical page frame (may involve running a replacement algorithm), moves data from the disk to the new page frame, and updates the page table with the new mapping
- The physical address PA is obtained by concatenating the PFN and page offset



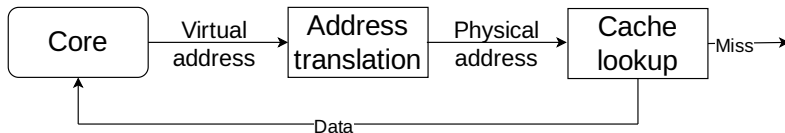
Translation Lookaside Buffer (TLB)

- A TLB is used to cache recent address translations
 - ▶ TLBs are usually fully associative and contain a mapping from VPNs to PTEs
- On a TLB miss,
 - (i) Hardware implementation will walk the page table
 - (ii) Software implementation will trap to the kernel, fill the TLB with the desired translation, and resume execution



Physical Caches

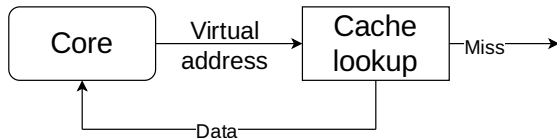
- Every memory operation **requires accessing** the TLB **first** to translate a virtual to a physical address
 - ▶ Called a physically-indexed physically-tagged (PIPT) cache
- Address translation is performed **before** cache access in physically addressed caches
 - Increases the cache hit time
- PIPT caches are popularly used for lower-level caches



Virtual Caches

The cache is accessed using virtual addresses, while the tags can be either virtual or physical

- In a virtually-indexed virtually-tagged (VIVT) cache, address translation is performed **after** cache access **only on a miss**
 - + Potentially lower access time and energy consumption
 - + Can have larger and more sophisticated TLBs
 - Permission bits need to be replicated in the cache
 - Requires associating virtual addresses with owner processes to deal with context switches
 - Introduces challenges in the form of synonyms (or aliases) and homonyms



Homonyms and Synonyms with Virtual Caches

Homonyms occur when the same virtual address points to different physical addresses

- Possible solutions
 - ▶ Use physically-addressed caches, flush the cache on each context switch, or add an address-space ID (ASID) to each tag

Different virtual pages point to the same physical page in synonyms

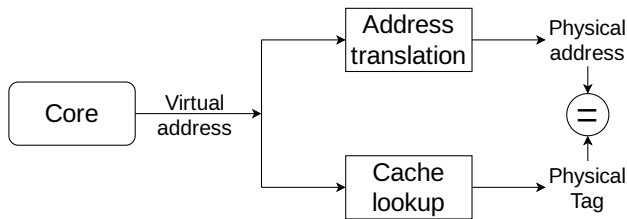
- Challenging because all synonyms must be kept coherent
- Possible solutions
 - ▶ Use physically-addressed caches, or use additional hardware support for tracking synonyms

X. Qiu and M. Dubois. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. IEEE Transactions on Computers, 2008.

H. Yoon and G. Sohi. Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping. HPCA 2016.

Virtually-Indexed Physically-Tagged Caches

- Indexing the cache is an expensive operation, so it is desirable to **overlap** indexing with TLB lookup
 - ▶ Compute index from the virtual address, look up the desired set
 - ▶ Compare tags after the PA is available
 - ▶ Leads to virtually-indexed physically-tagged (VIPT) cache



Virtually-Indexed Physically-Tagged Caches

● Homonyms

- ▶ Two different physical addresses will have different tags
- The TLB must be flushed on a context switch, or else a translation may end up with the wrong PA
 - Extend the tag bits with ASID

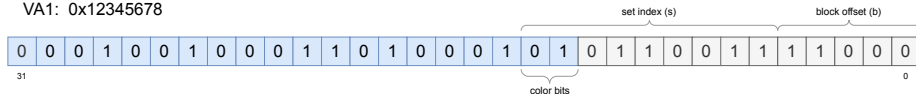
● Synonyms

- ▶ Deal with synonyms either by constraining index bits or through page coloring via OS support
- ▶ Constraining index bits ($s + b == 12$) implies the synonyms will map to the same set
 - The tags of the ways containing synonyms will be the same because the PA is the same
- ▶ Page coloring (e.g., ARM v6) tightly couples the hardware and the OS
- ▶ Using only 2 MB huge pages (low 21 bits remain same) breaks backward compatibility and has performance concerns

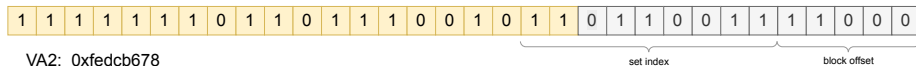
The Synonym Problem and Page Coloring

Given 4 KB pages, assume that virtual addresses 0x12345678 and 0xfedcb678 are mapped to the same physical address 0x9abcd678

VA1: 0x12345678



31



VA2: 0xfedcb678

set index

block offset

...	...
VA2	Data at PA
...	...
VA1	Data at PA
...	...

E-way set in a virtual cache

References

-  D. Patterson and J. Hennessy. Computer Organization and Design. Sections 5.1, 5.3–5.4, 5.7–5.8, 5th edition, Morgan Kaufmann.
-  J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Appendix B.1–B.4, Sections 2.1, 2.3 6th edition, Morgan Kaufmann.
-  R. Bryant and D. O'Hallaron. Computer Systems: A Programmer's Perspective. Sections 6.2–6.4, 3rd edition, Pearson Education.
-  J. L. Baer. Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors. Sections 6.1–6.3, Cambridge University Press.
-  M. Cekelov and M. Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. IEEE Micro, 1997.