CS110 Computer Architecture

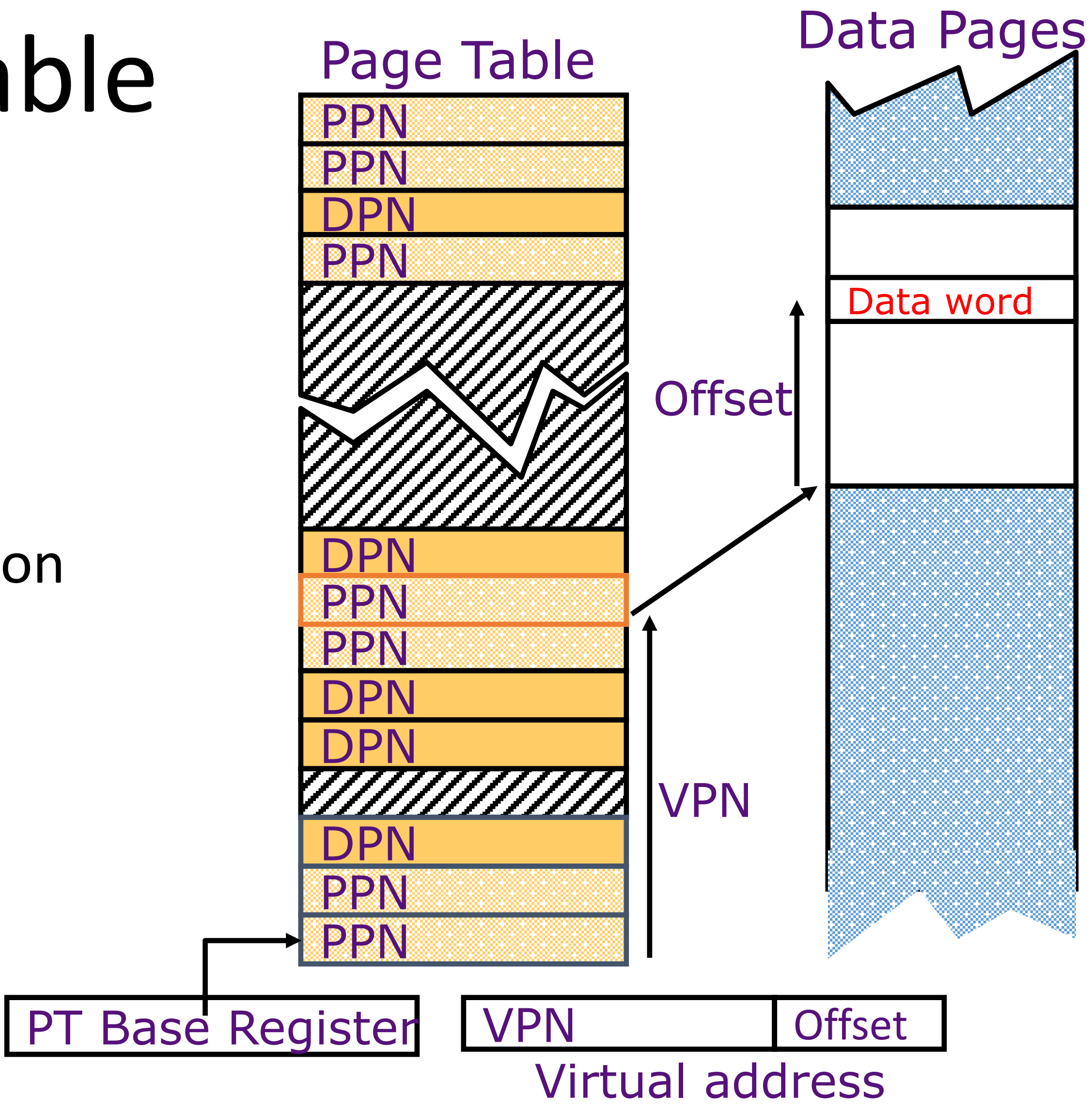
VM and Advanced Caches

Chundong Wang & Siting Liu

SIST, ShanghaiTech

Linear (simple) Page Table

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists
  - And either PPN or DPN:
  - ▨ PPN (physical page number) for a memory-resident page
  - 🟧 DPN (disk page number) for a page on the disk
  - Status bits for protection and usage (read, write, exec)
- OS sets the Page Table Base Register whenever active user process changes



Suppose an instruction references a memory page that isn't in DRAM?

- We get an exception of type "page fault"
- Page fault handler does the following:
  - If virtual page doesn't yet exist, assign an unused page in DRAM, or if page exists …
  - Initiate transfer of the page we're requesting from disk to DRAM, assigning to an unused page
  - If no unused page is left, a *page currently in DRAM is selected to be replaced* (based on usage)
  - The replaced page is written (back) to disk, page table entry that maps that VPN->PPN is marked as invalid/DPN
  - Page table entry of the page we're requesting is updated with a (now) valid PPN

Size of Linear Page Table

With 32-bit memory addresses, 4-KB pages:

=> $2^{32} / 2^{12} = 2^{20}$ virtual pages per user, assuming 4-Byte PTEs,

=> 2^{20} PTEs, i.e, 4 MB page table per process!

Larger pages?

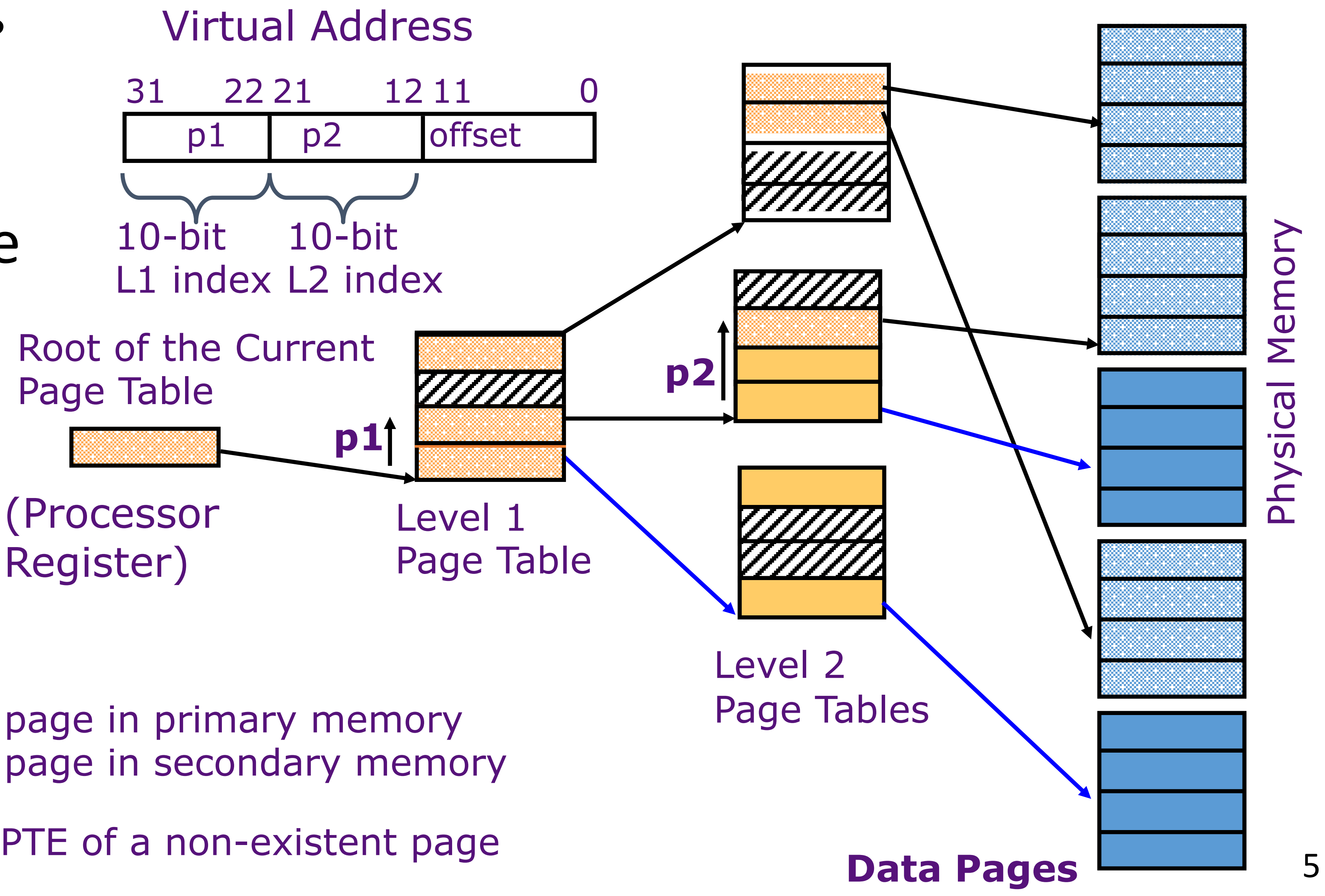
- Internal fragmentation (Not all memory in page gets used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-Byte PTEs (35 TB!)

What is the "saving grace" ? Most processes only use a set of high address (stack), and a set of low address (instructions, heap)

Hierarchical Page Table – exploits sparsity of virtual address space use

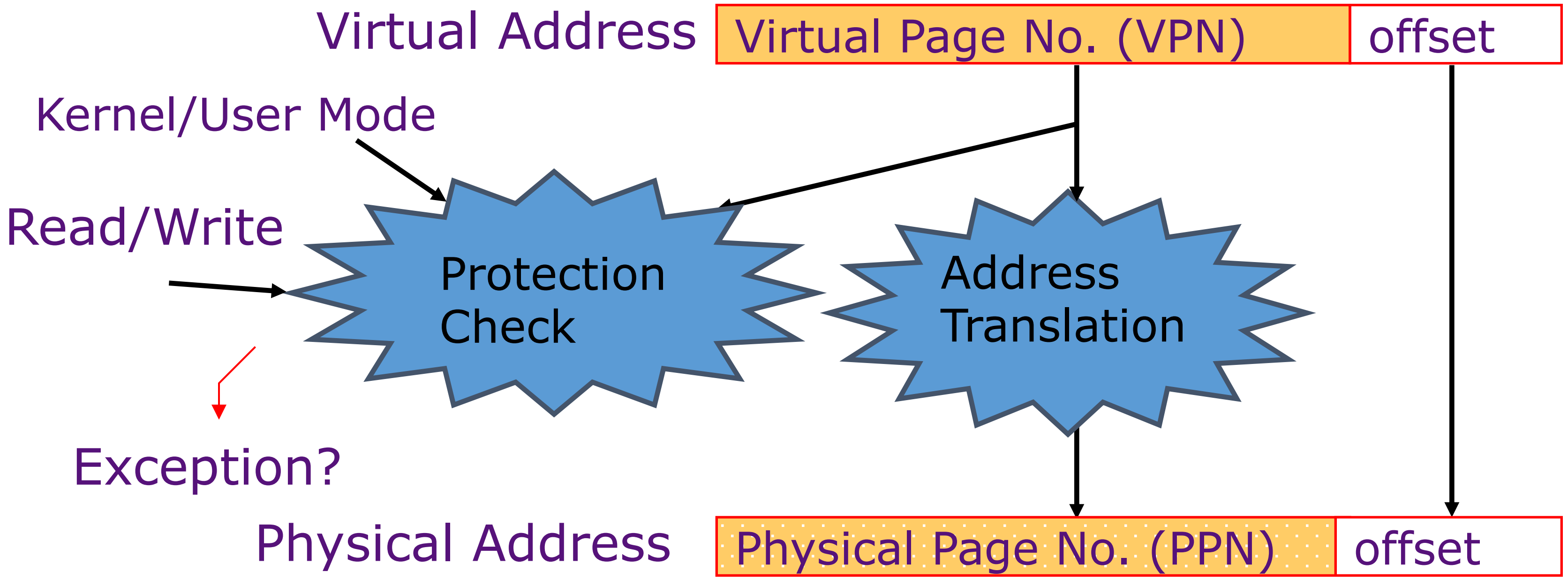


▨ page in primary memory

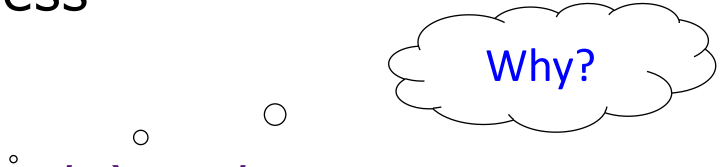
🟧 page in secondary memory

▨ PTE of a non-existent page

Address Translation & Protection



- Every instruction and data access needs address translation and protection checks



- *A good VM design needs to be fast (~ one cycle) and space efficient*

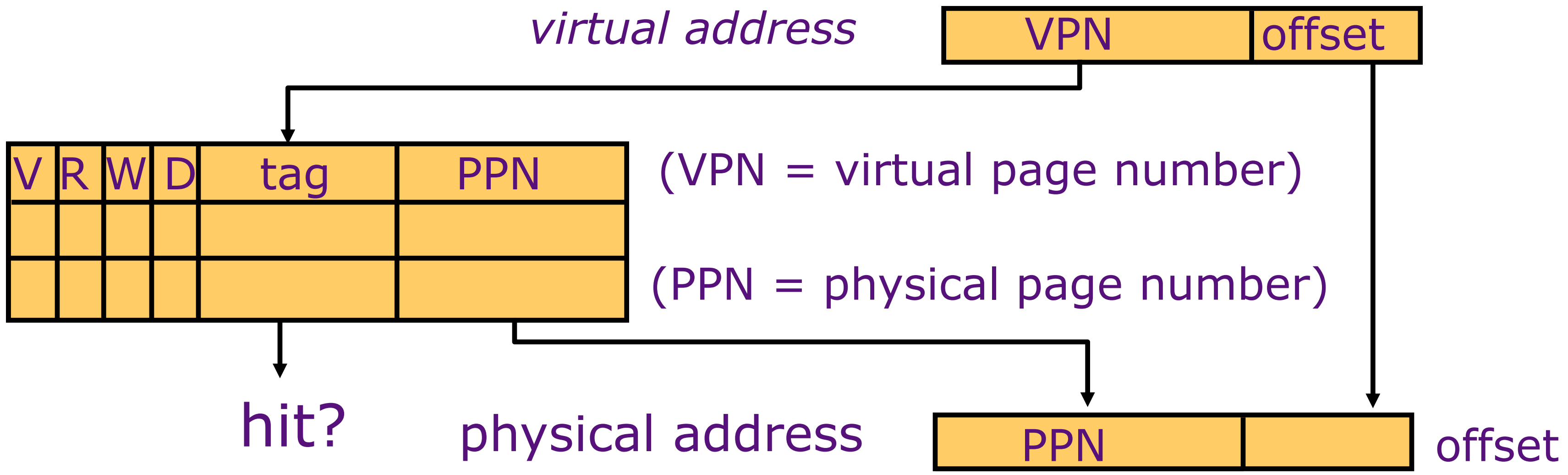
Translation Lookaside Buffers (TLB)

Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache some translations in TLB*

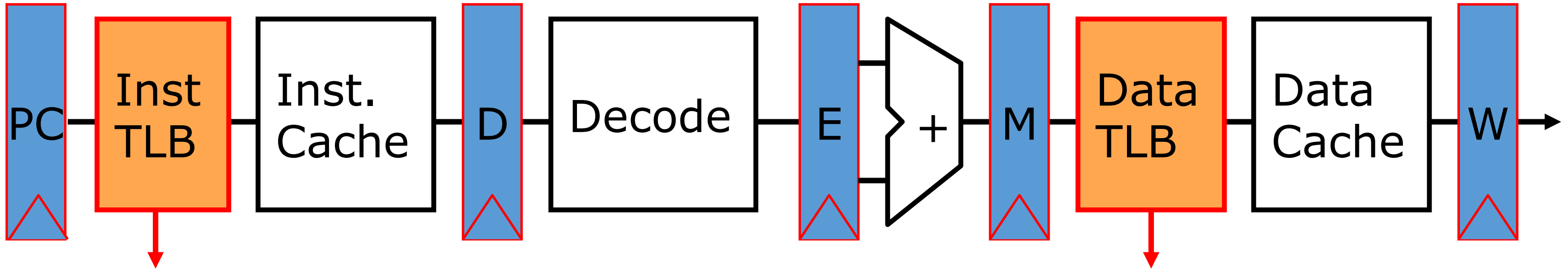
TLB hit	=> *Single-Cycle Translation*
TLB miss	=> *Page-Table Walk to refill*



TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- Upon context switch? New VM space! Flush TLB …
- “TLB Reach”: Size of largest virtual address space that can be simultaneously mapped by TLB

VM-related events in pipeline



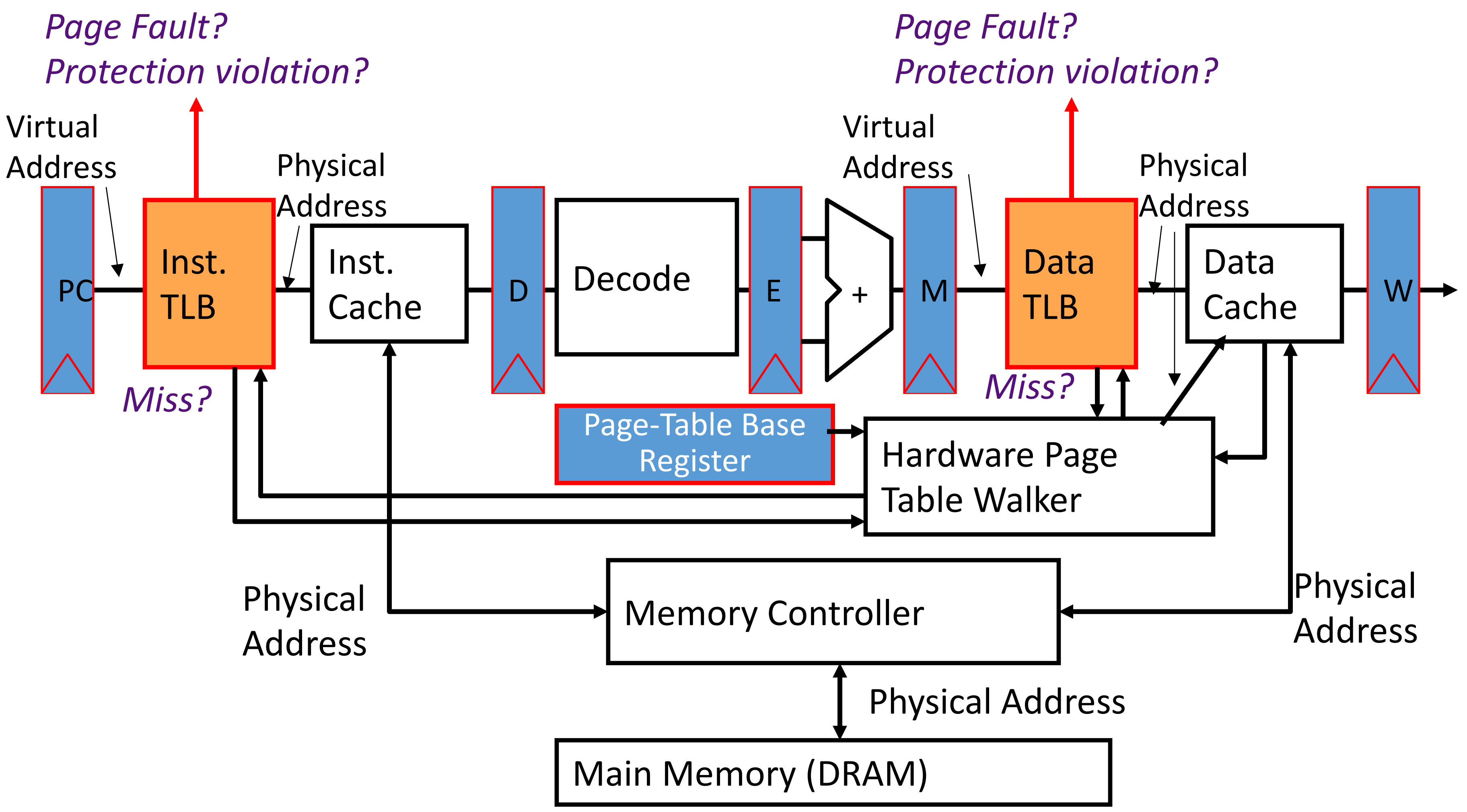
TLB miss? Page Fault? Protection violation?

TLB miss? Page Fault? Protection violation?

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware now
- Handling a page fault (e.g., page is on disk) needs a *precise* trap so software handler can easily resume after retrieving page
- Handling protection violation may abort process

Page-Based Virtual-Memory Machine

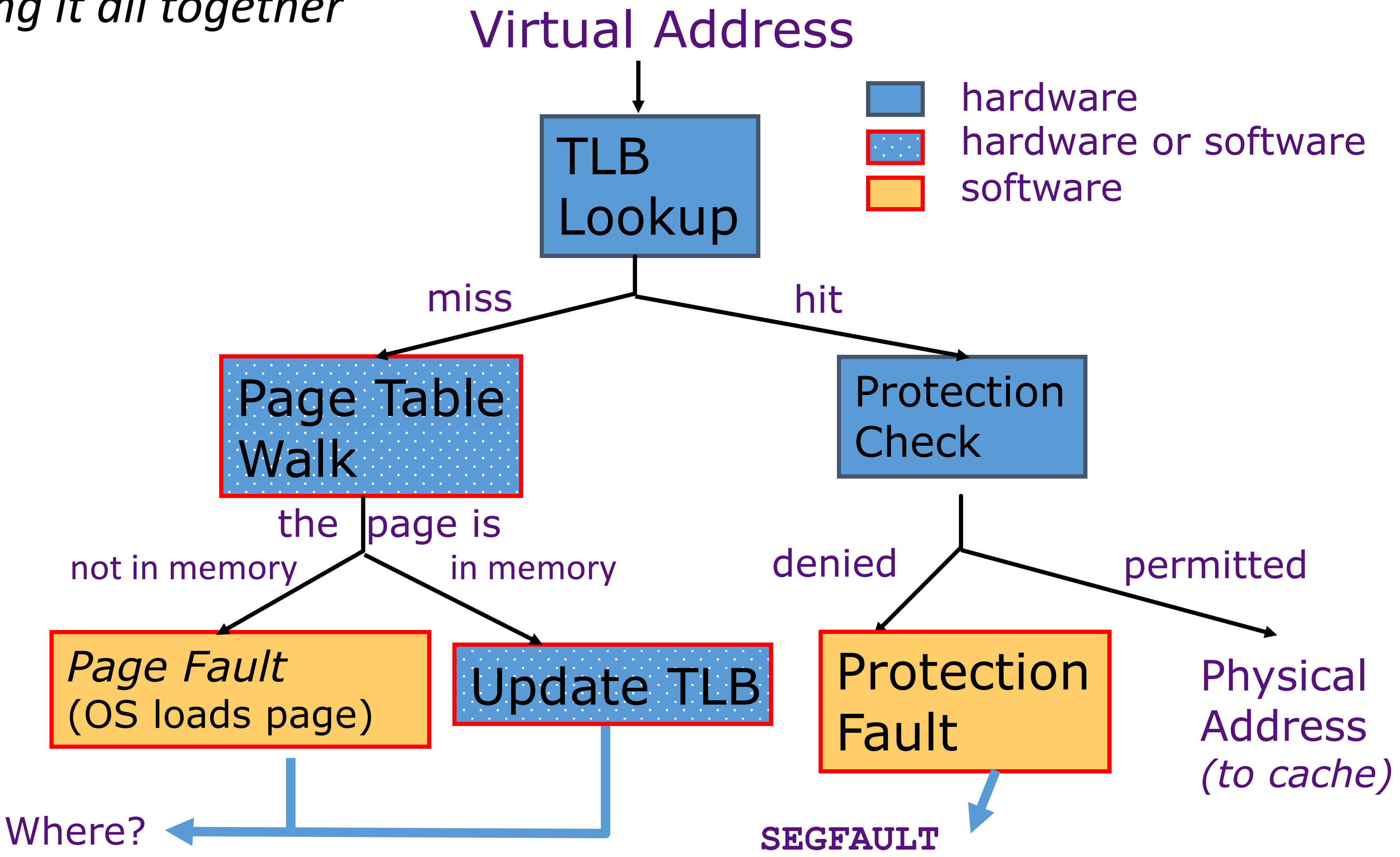
(Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

Address Translation:

putting it all together



Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

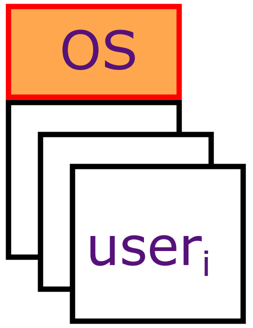
page table ≡ name space

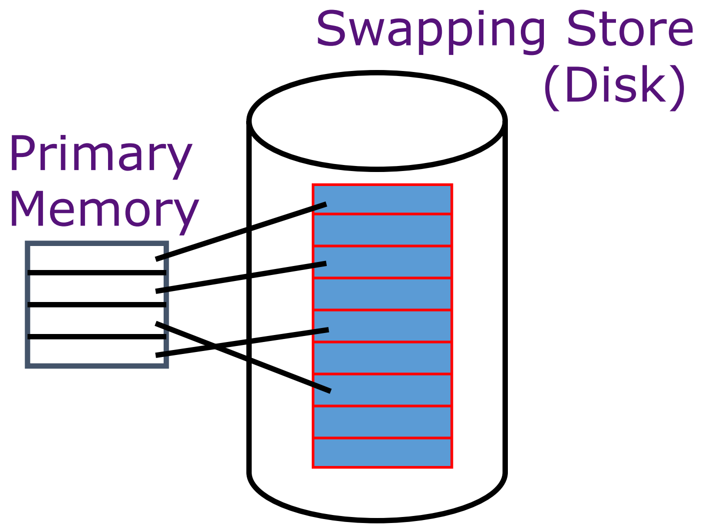
Demand Paging

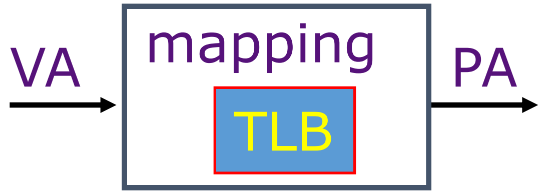
Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

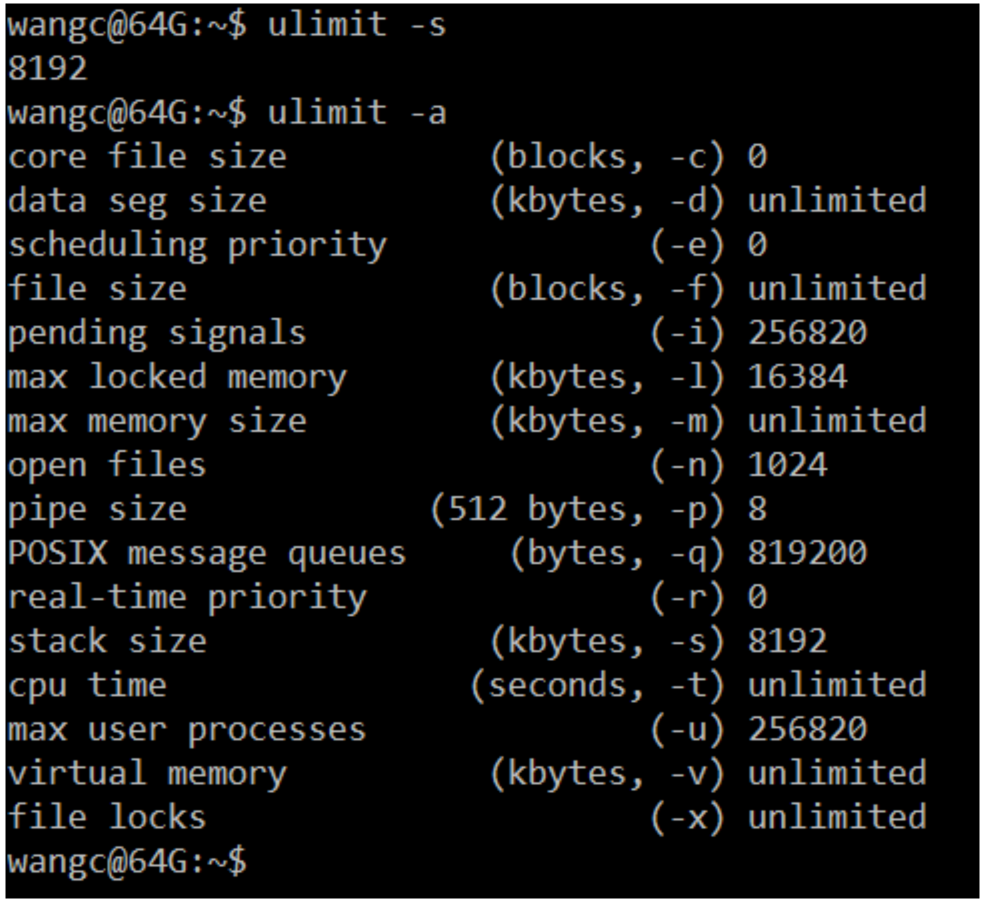
The price is address translation on each memory reference







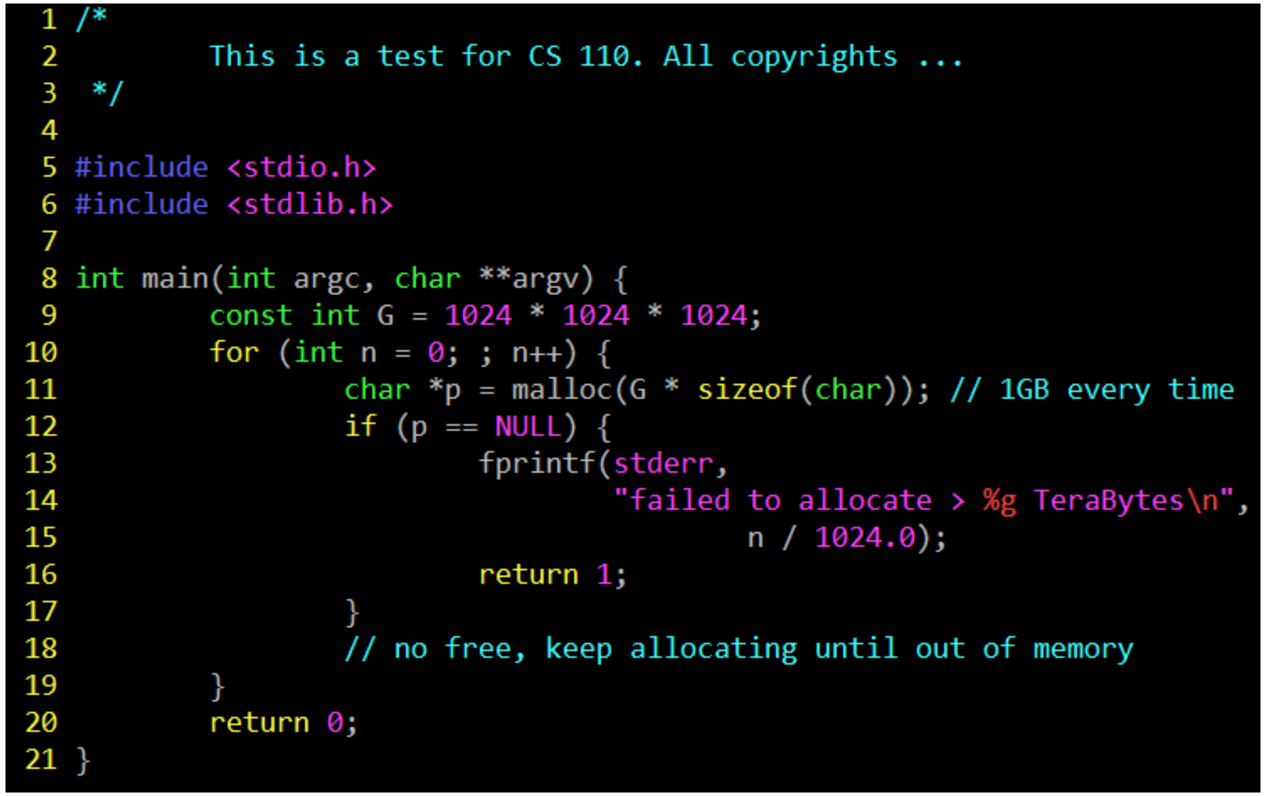
Unlimited?



```
wangc@64G:~$ ulimit -s
8192
wangc@64G:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 256820
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 256820
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
wangc@64G:~$
```

Remember: Out of Memory

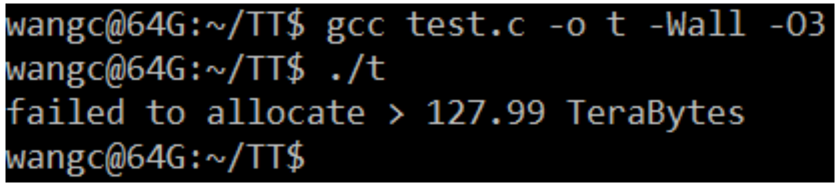
- Insufficient free memory: **`malloc()`** returns **`NULL`**



```c
/*
        This is a test for CS 110. All copyrights ...
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        const int G = 1024 * 1024 * 1024;
        for (int n = 0; ; n++) {
                char *p = malloc(G * sizeof(char)); // 1GB every time
                if (p == NULL) {
                        fprintf(stderr,
                                "failed to allocate > %g TeraBytes\n",
                                        n / 1024.0);
                        return 1;
                }
                // no free, keep allocating until out of memory
        }
        return 0;
}
```



```
wangc@64G:~/TT$ gcc test.c -o t -Wall -O3
wangc@64G:~/TT$ ./t
failed to allocate > 127.99 TeraBytes
wangc@64G:~/TT$
```

Limited VM Space with x86-64

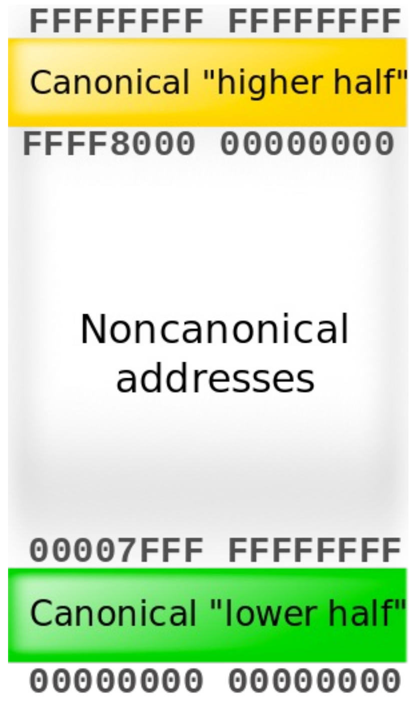
- 64-bit Linux allows up to **128TB** of virtual address space for individual processes, and can address approximately 64 TB of physical memory, subject to processor and system limitations.
- For Windows 64-bit versions, both 32- and 64-bit applications, if not linked with "*large address aware*", are limited to **2GB** of virtual address space; otherwise, **128TB** for Windows 8.1 and Windows Server 2012 R2 or later.

Source: https://en.wikipedia.org/wiki/X86-64

48bit for address translation only

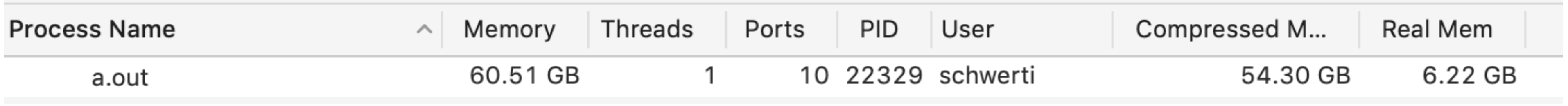
- Still provides plenty of space!
- Higher bits "sign extended": "canonical form"
- Convention: "Higher half" for the Operating System
- Intel has plans ("whitepaper") for 56 bit translation – no hardware yet

- https://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details



Using 128TB of Memory!?

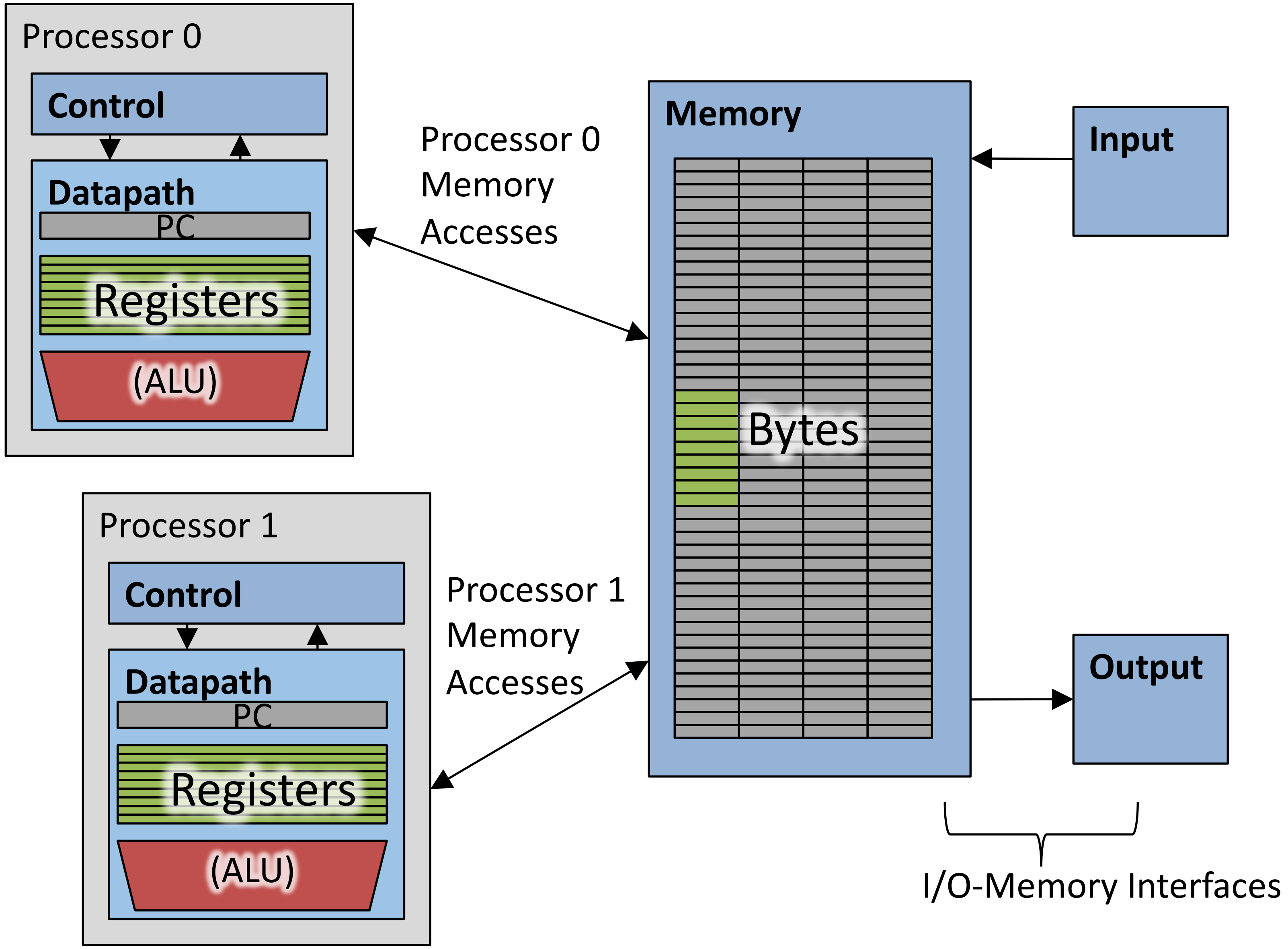
- A lazy allocation of virtual memory
  - Not used ➔ not allocated
  - Try reading and writing from those pointers:
    works!
  - Even writing Gigabaytes of memory:
    works!
- Memory Compression!
  - Take not-recently used pages, compress them => free the physical page
- https://www.lifewire.com/understanding-compressed-memory-os-x-2260327



Process Name	Memory	Threads	Ports	PID	User	Compressed M...	Real Mem
a.out	60.51 GB	1	10	22329	schwerti	54.30 GB	6.22 GB

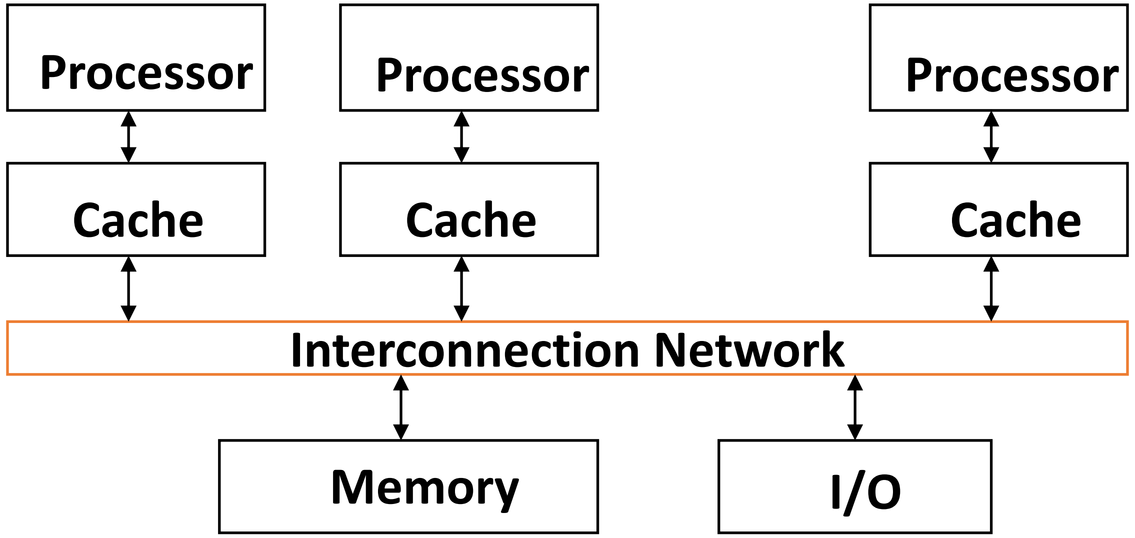
Cache Coherence

Simple Multi-core Processor



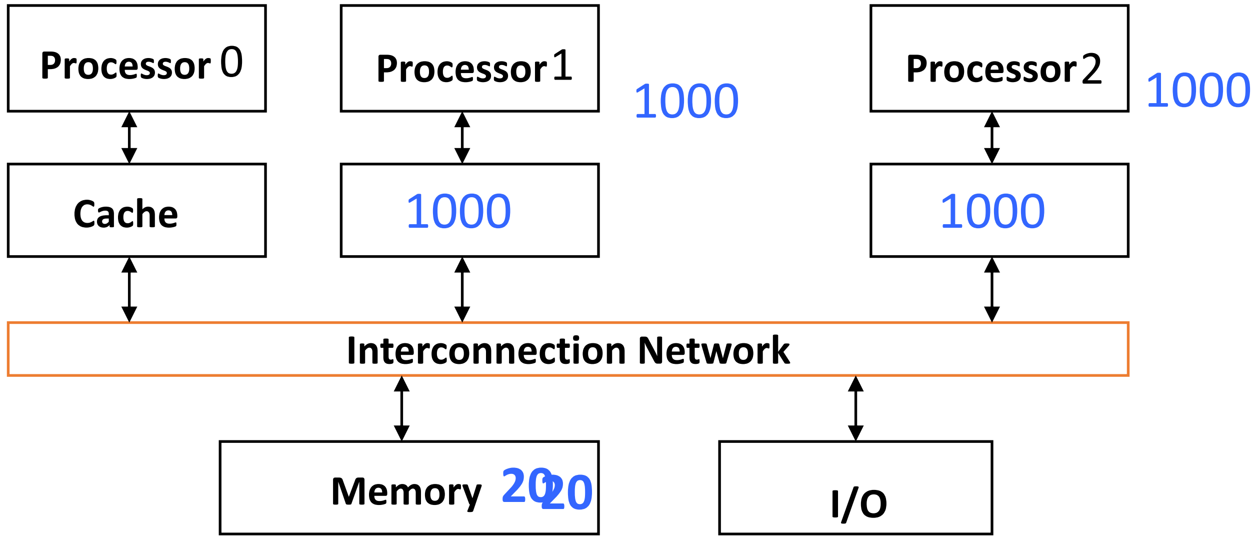
Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the **shared** common memory



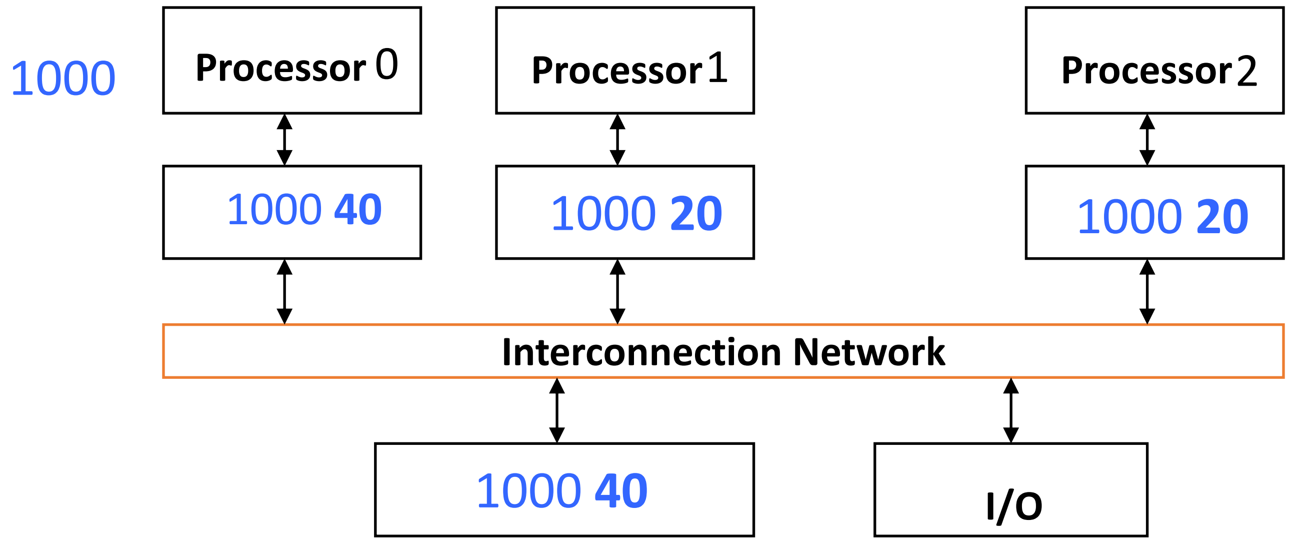
Shared Memory and Caches

- What if?
  - Processors 1 and 2 **read** Memory[1000] (value 20)



Shared Memory and Caches

- Now:
  - Processor 0 **writes** Memory[1000] with 40



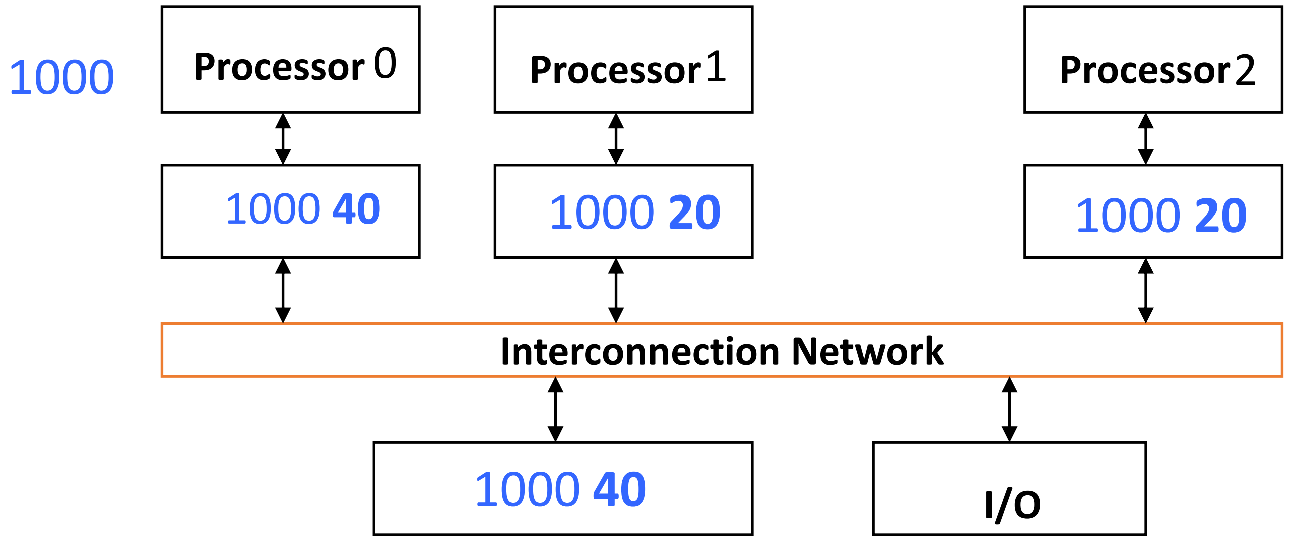
Problem?

Keeping Multiple Caches Coherent

- Architect's job: shared memory
  => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache

Shared Memory and Caches

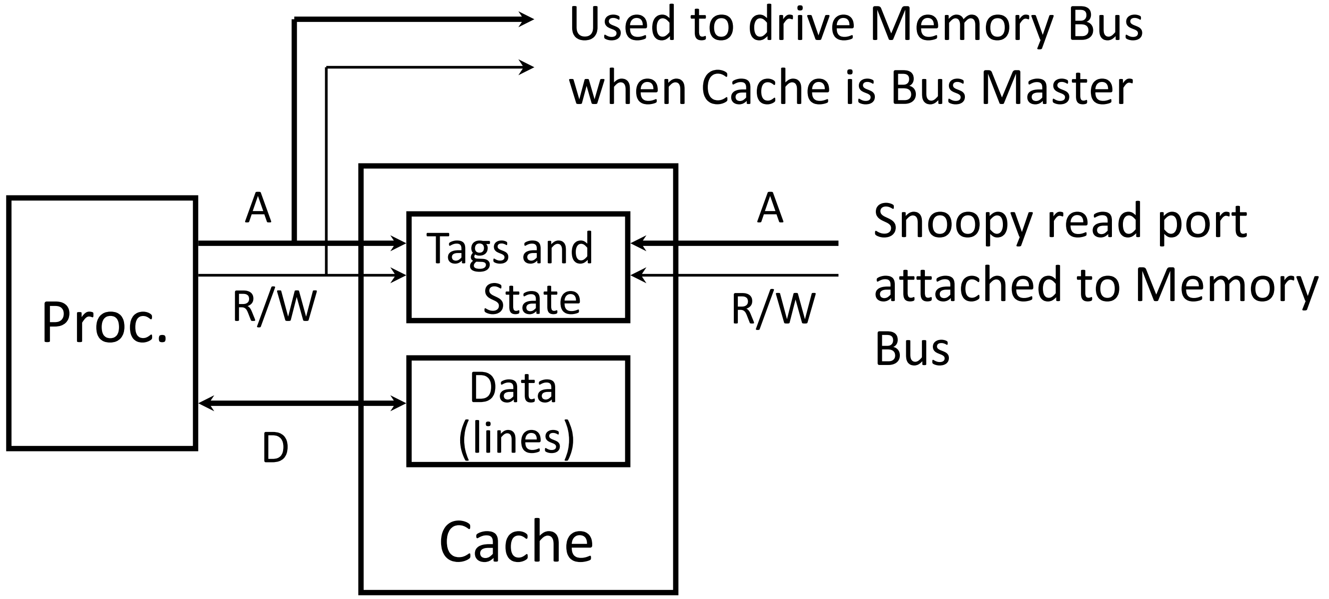
- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



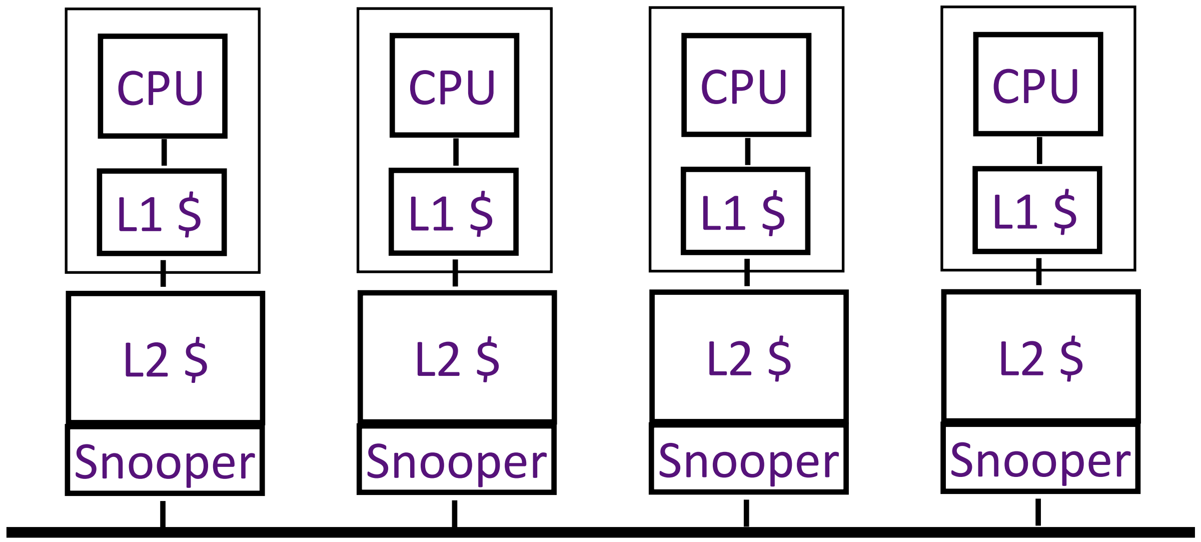
Processor 0
Write
Invalidates
Other Copies

Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
- Snoopy cache tags are dual-ported

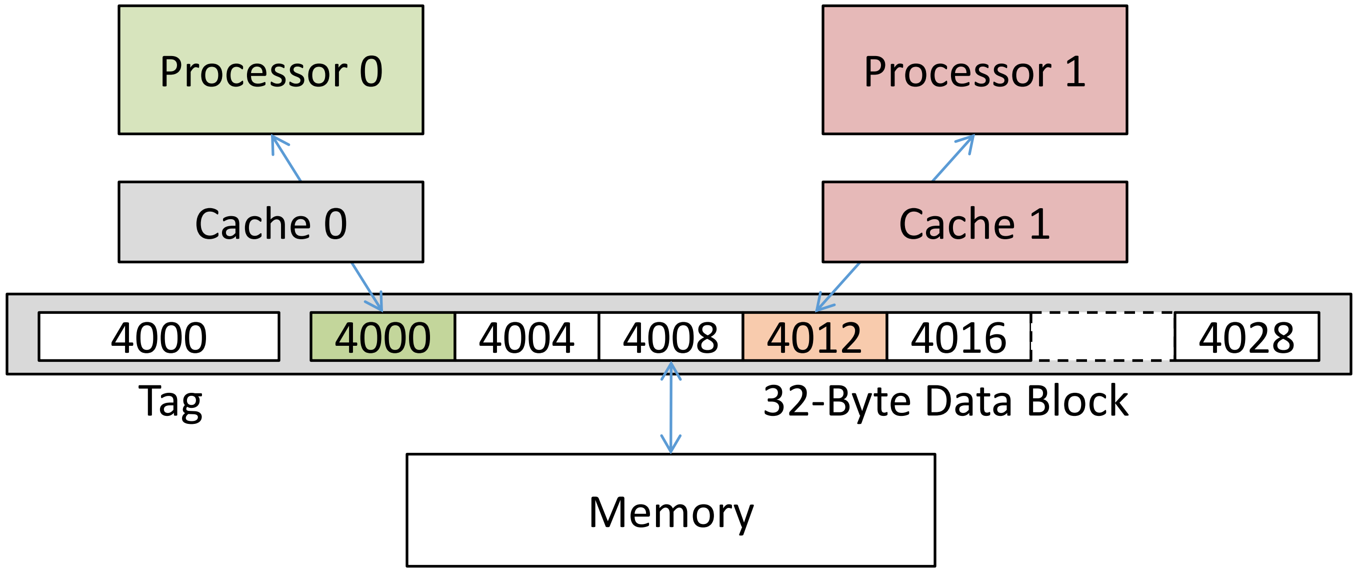


Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 => invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Cache Coherency Tracked by Block



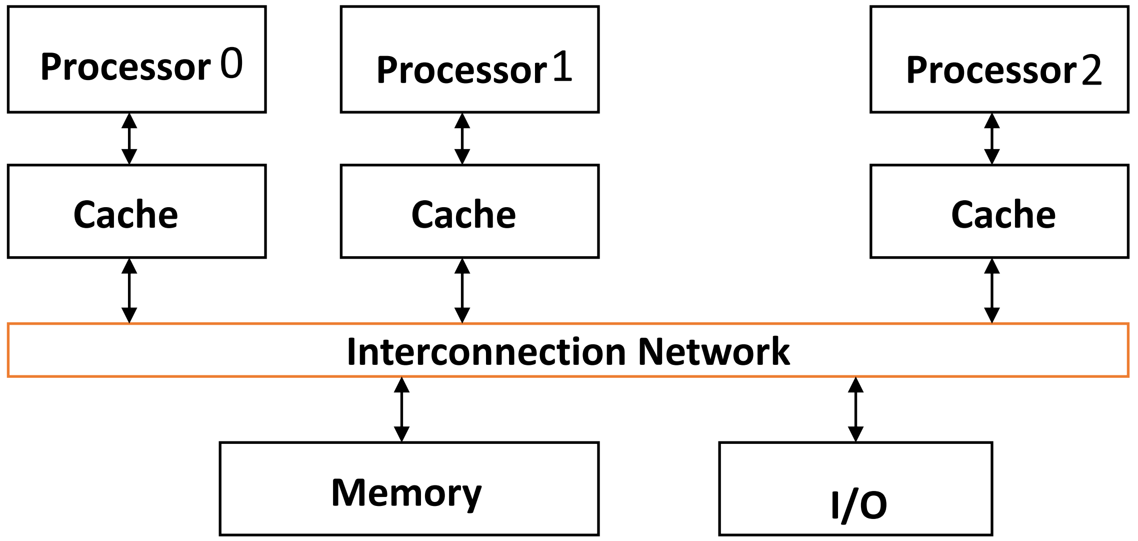
- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

Coherency Tracked by Cache Block

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?
  - Keep variables far apart (at least block size (64 byte))

Shared Memory and Caches

- Use valid bit to "unload" cache lines (in Processors 1 and 2)
- Dirty bit tells me: "I am the only one using this cache line"! => no need to announce on Network!



Review: Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block, impossible to avoid; small effect for long-running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity (not compulsory and…)
  - Cache cannot contain all blocks accessed by the program ***even with perfect replacement policy in fully associative cache***
  - Solution: increase cache size (may increase access time)
- Conflict (not compulsory or capacity and…):
  - Multiple memory locations map to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)
  - Solution 3: improve replacement policy, e.g.. LRU

Fourth “C” of Cache Misses: *Coherence* Misses

- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses

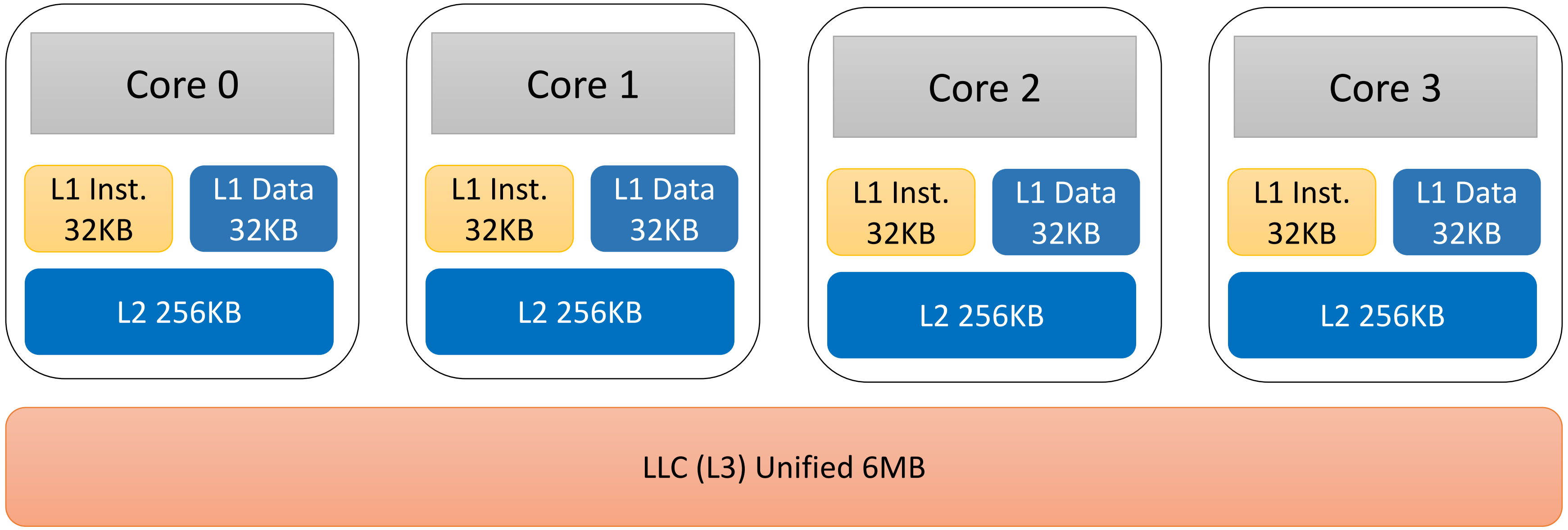
Coherence Protocols

- The cache coherence protocols ensure that there is a coherent view of data, with migration and replication.
  - A cache line has a state
- MSI
  - Modified, Shared, Invalid
- MESI
  - MSI + Exclusive
- MOESI
  - MESI + O
  - e.g., AMD processor family
- MESIF
  - MESI + F
  - e.g., Intel Xeon processors

Advanced Caches: MRU is LRU

Cache Inclusion

- Multilevel caches

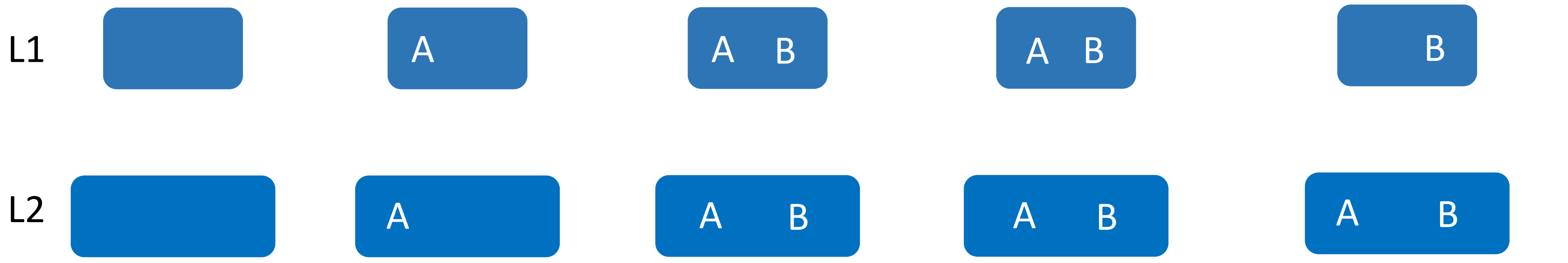


Intel Ivy Bridge Cache Architecture (Core i5-3470)

If all blocks in the higher level cache are also present in the lower level cache, then the lower level cache is said to be **inclusive** of the higher level cache.

Inclusive

$$L_n \subsetneqq L_{n+1} \; (n \geq 1)$$



Initial state

Read A miss; load A into L1 and L2

Read B miss; load B into L1 and L2

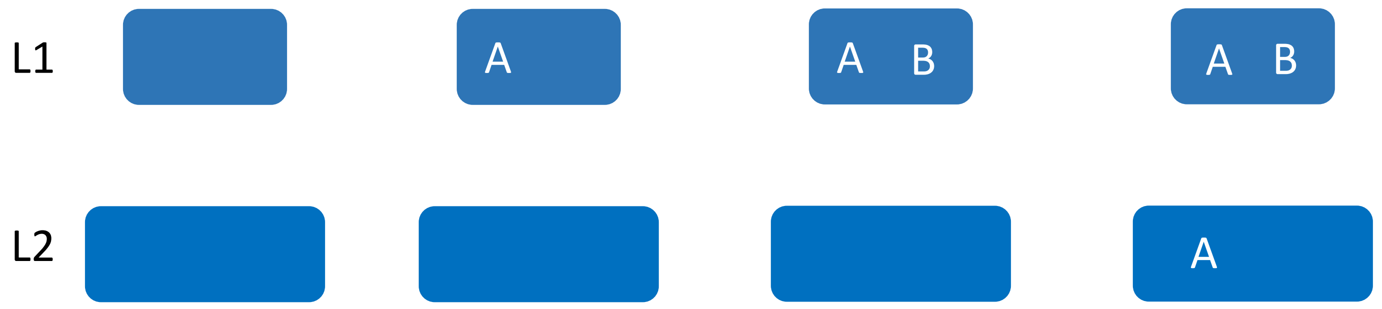
Evict A from L1 due to cache replacement

Evict B from L2 due to cache replacement

Back invalidation

Exclusive

$$L_n \cap L_{n+1} = \emptyset \ (n \geq 1)$$



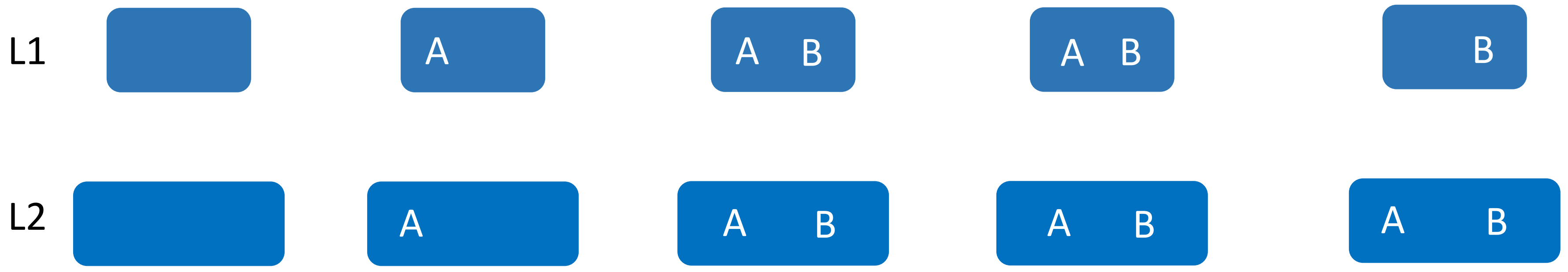
Initial state

Read A miss; load A into L1

Read B miss; load B into L1

Evict A from L1 due to cache replacement and place in L2

Non-inclusive



Initial state

Read A miss; load A into L1 and L2

Read B miss; load B into L1 and L2

Evict A from L1 due to cache replacement

Evict B from L2 due to cache replacement

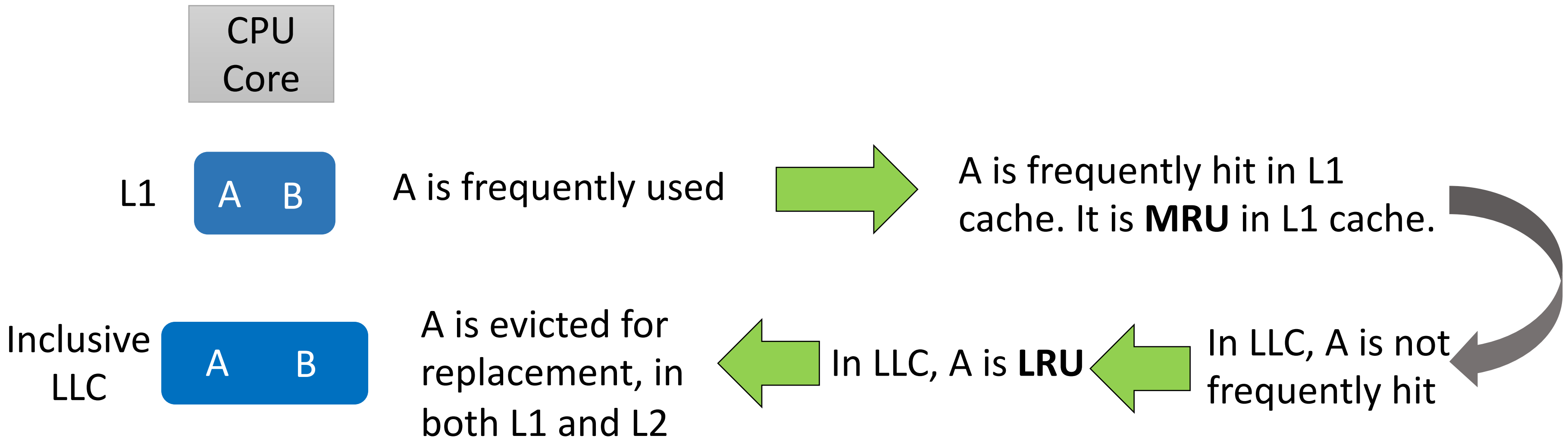
Real-world CPUs

- Intel Processors
  - Sandy bridge, inclusive
  - Haswell, inclusive
  - Skylake-S, inclusive
  - Skylake-X, non-inclusive
- ARM Processors
  - ARMv7, non-inclusive
  - ARMv8, non-inclusive
- AMD
  - K6, exclusive
  - Zen, inclusive
  - Shanghai, LLC non-inclusive

Inclusive, or not?

- Inclusive cache eases coherence
    - A cache block in a higher-level surely existing in lower-level(s)
    - A non-inclusive LLC, say L2 cache, which needs to evict a block, **must** ask L1 cache if it has the block, because such information is not present in LLC.
- Non-inclusive cache yields higher performance though, why?
    - No back invalidation
    - More data can be cached $\leftarrow$ larger capacity

‘*Sneaky*’ LRU for Inclusive Cache



A is frequently used

A is frequently hit in L1 cache. It is **MRU** in L1 cache.

In LLC, A is not frequently hit

In LLC, A is **LRU**

A is evicted for replacement, in both L1 and L2

As a result, MRU block that should be retained might be evicted, which causes performance penalty.

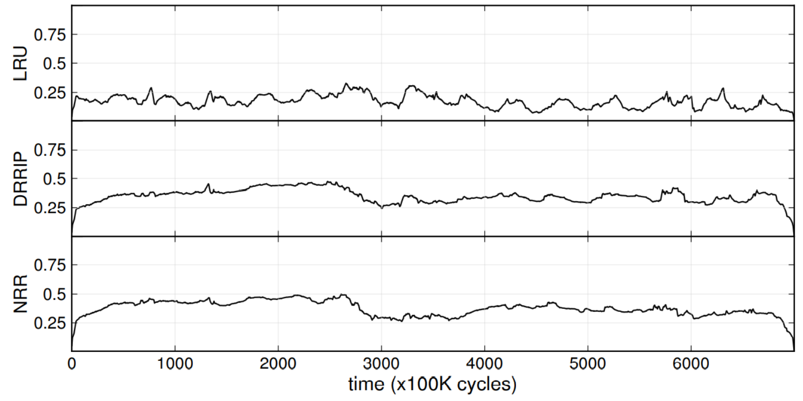
What if LLC is non-inclusive?

Should you be interested, you can click *https://doi.org/10.1109/MICRO.2010.52* to read the related research paper for details.

Advanced Caches: Reduce the size of LLC

Reduce LLC for high performance

- Problem
  - A considerable portion of the shared LLC is dead
- Why?
  - LLC accesses, caused by L1 and L2 misses
  - Locality not accurate due to filtering by L1 and L2
  - LLC uniformly handles any access request for line allocation/deallocation
- How to resolve?
  - Leverage the reuse locality to selectively allocate LLC lines

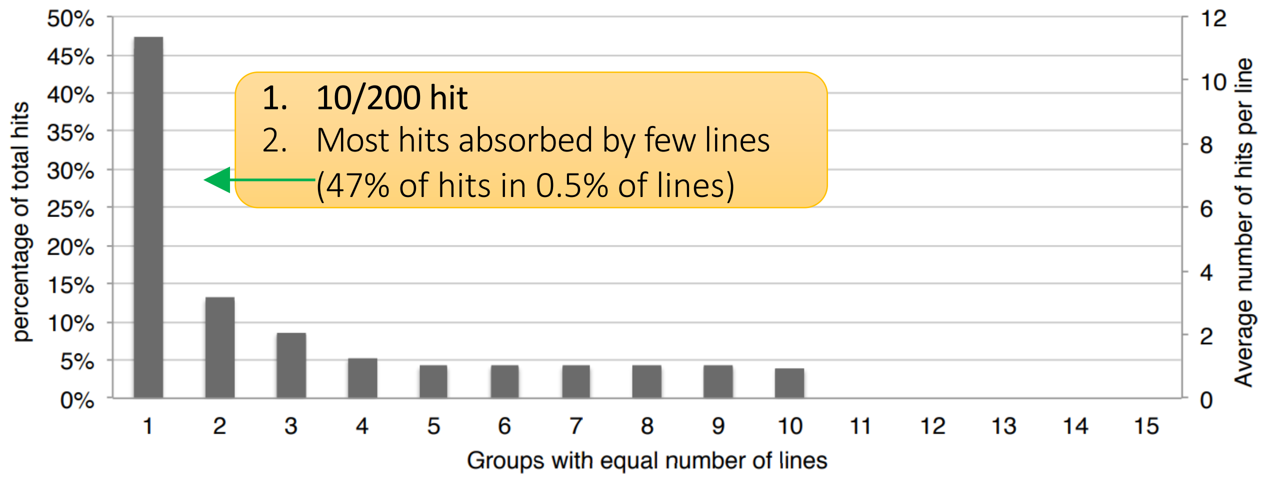


(a) Changes in the fraction of live lines over time

More than 83.8% LLC lines not productive

Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. 2013. The reuse cache: downsizing the shared last-level cache. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46). Association for Computing Machinery, New York, NY, USA, 310–321.

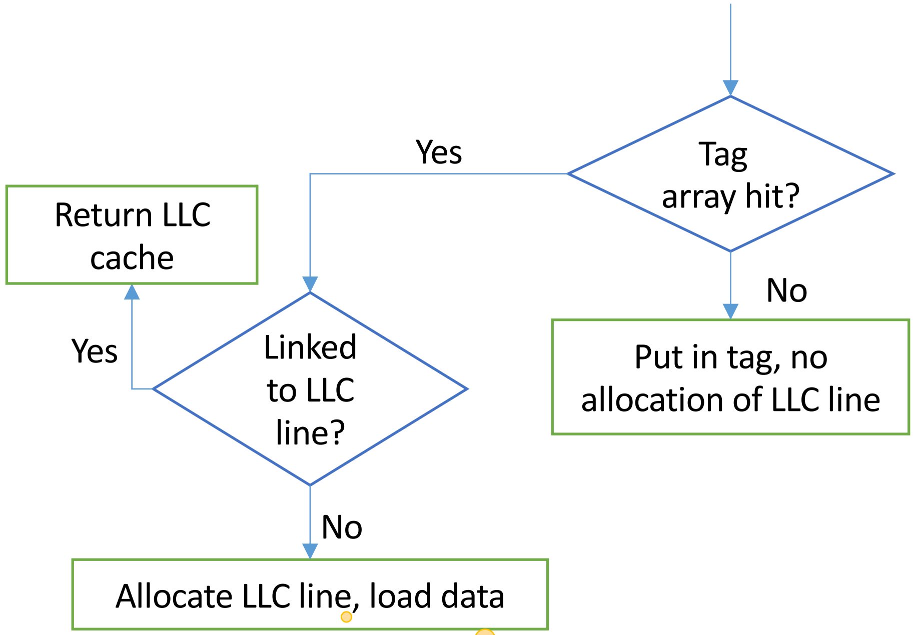
Selective allocation upon reuse locality

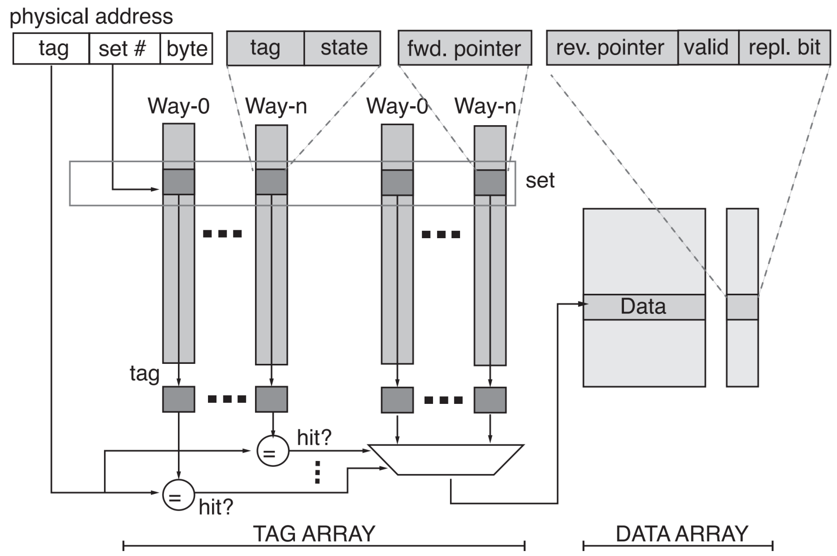


(b) Distribution of hits among all lines loaded (or reloaded) into the LRU SLLC during their stay. Each group represents 0.5% of the loaded lines

- Reuse locality
- Selective allocation
  - Tag and cache line decoupled
    - Conventionally, one tag for one cache line
    - Now, more tags than cache lines
      - Some place holders
  - Only keeping reused cache line

Allocation policy


Tag array hit?
Yes
No
Put in tag, no allocation of LLC line
Linked to LLC line?
Yes
Return LLC cache
No
Allocate LLC line, load data

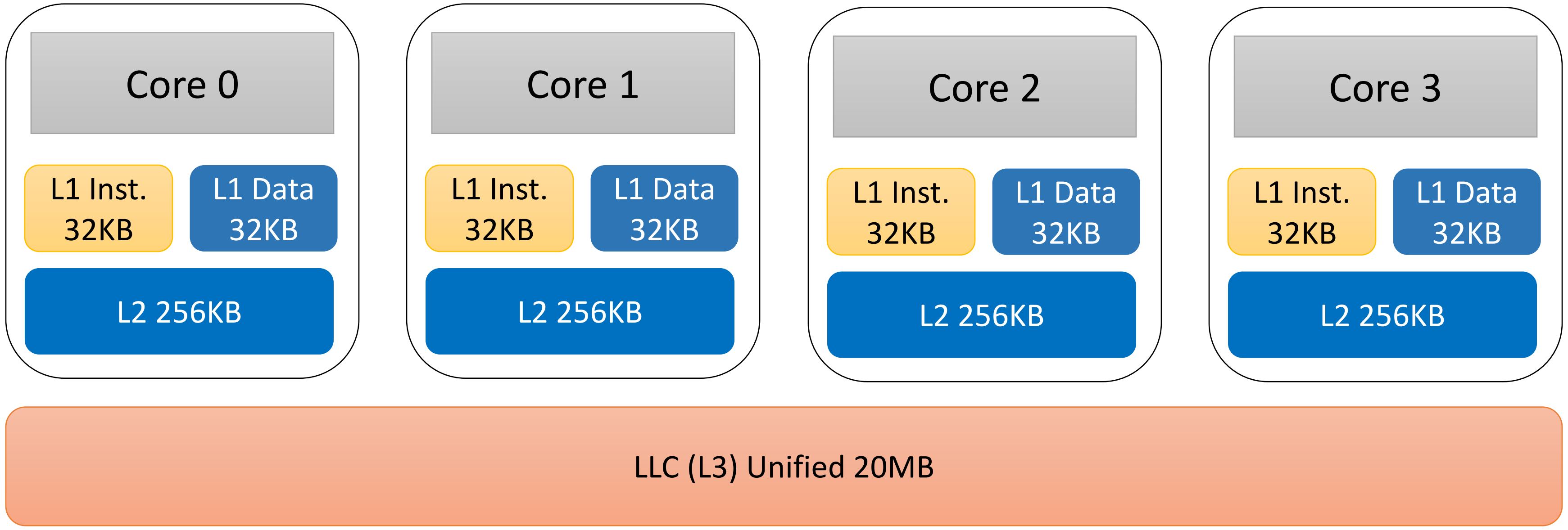


On replacement, the tag of evicted LLC line is kept. Once hit again, i.e., reused, data reloaded again.

Advanced Caches:
LLC is not monolithic

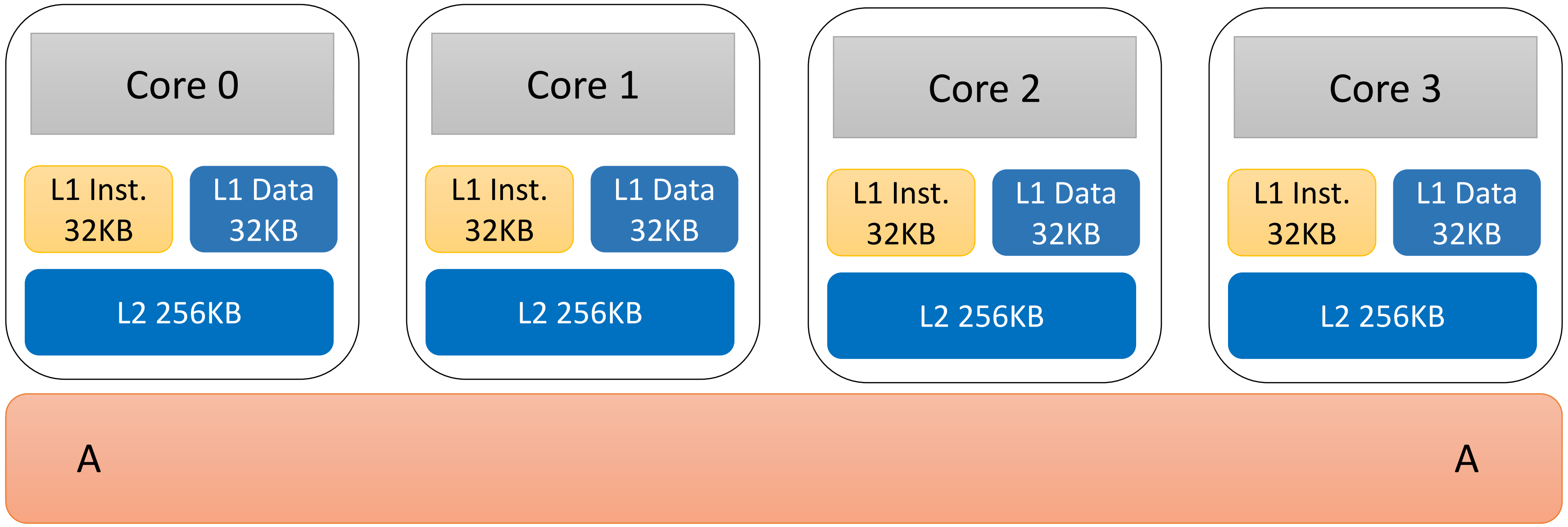
LLC is not monolithic

Intel® Xeon® Processor E5-2667 v3

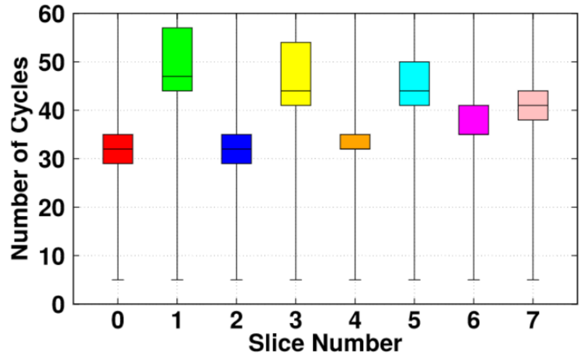


Previously, it’s considered that, to CPU cores, LLC is monolithic. No matter where a cache block in the LLC, a core would load it into private L2 and L1 cache with **the same** time cost.

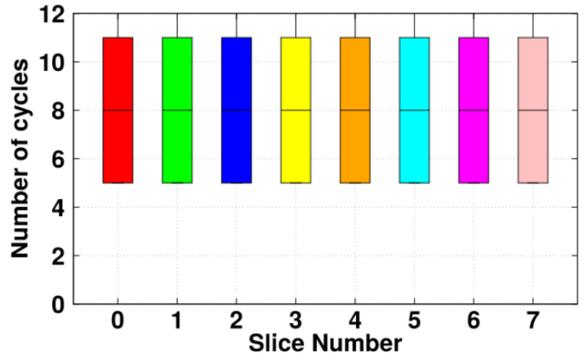
Intel® Xeon® Processor E5-2667 v3



LLC (L3) Unified 20MB / Eight Slices



(a) Read.



(b) Write.

LLC is fine-grained

From the paper https://doi.org/10.1145/3302424.3303977

Slice-aware memory management

- The idea seems simple
  - Put your data closer to your program (core)
- But it not *EASY* to do so
  - Cache management is undocumented, not to mention fine-grained slices
  - Researchers did a lot of efforts
    - Click https://doi.org/10.1145/3302424.3303977 for details
    - They managed to improve the average performance by 12.2% for GET operations of a key-value store.
    - 12.2% is a lot, if you consider the huge transactions every day for Google, Taobao, Tencent, JD, etc.

Advanced Caches: Replicating data in LLC

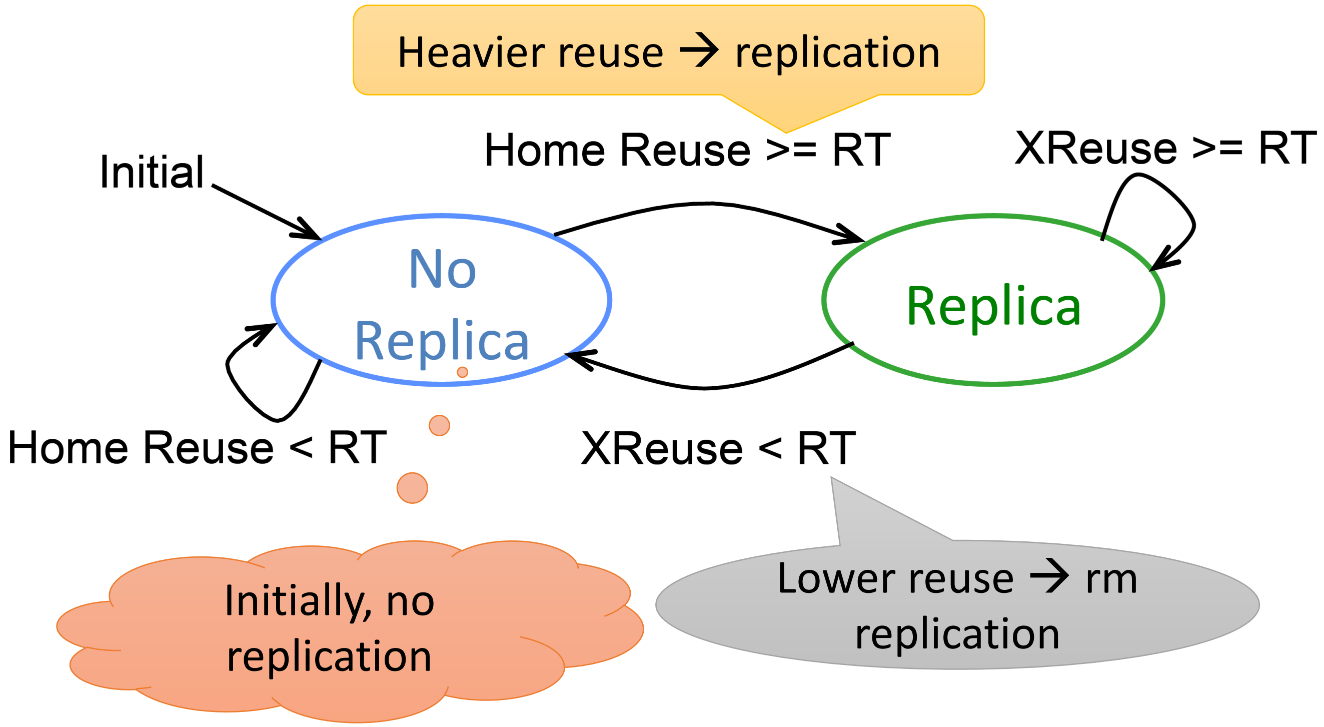
NUCA

- Non-uniform access cache
- The time cost to different locations in LLC is different for a particular core
  - In past slides, move data to a core using it
- Now replicate data at the mircoarch level
  - Towards a core using it

G. Kurian, S. Devadas and O. Khan, "Locality-aware data replication in the Last-Level Cache," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 1-12, doi: 10.1109/HPCA.2014.6835921.

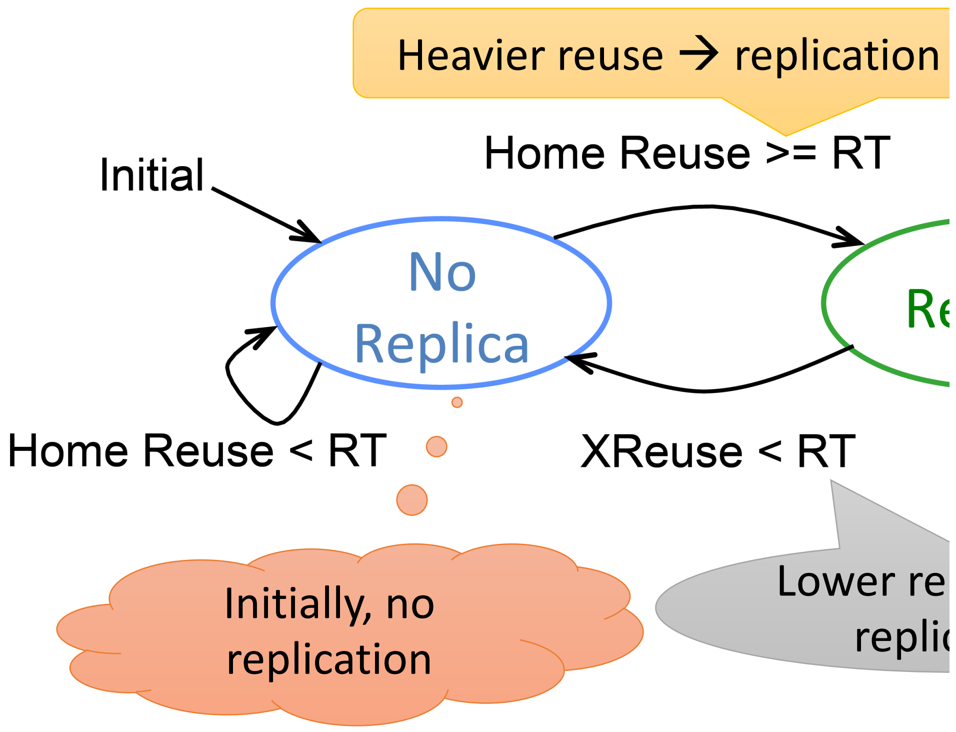
How to replicate cache lines?

- Reuse


Heavier reuse → replication
Initial
Home Reuse >= RT
XReuse >= RT
No Replica
Replica
Home Reuse < RT
XReuse < RT
Initially, no replication
Lower reuse → rm replication

How to replicate cache lines?

- Reuse



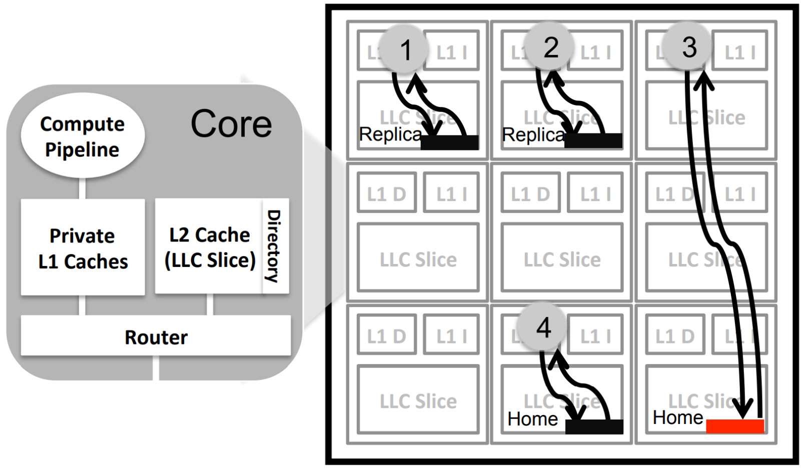


Figure 2. ① – ④ are mockup requests showing the *locality-aware LLC replication* protocol. The *black* data block has high reuse and a local LLC replica is allowed that services requests from ① and ②. The low-reuse *red* data block is not allowed to be replicated at the LLC, and the request from ③ that misses in the L1, must access the LLC slice at its home core. The home core for each data block can also service local private cache misses (e.g., ④).

G. Kurian, S. Devadas and O. Khan, "Locality-aware data replication in the Last-Level Cache," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 1-12, doi: 10.1109/HPCA.2014.6835921.

Conclusion

- There are many interesting facts of CPU cache
- To make the best of cache can boost your program’s performance!