Coding for Cache Optimization

ECE 565

Performance Optimization & Parallelism

Duke University, Fall 2018

Motivation

- Memory Wall
  - CPU speed and memory speed have grown at disparate rates
    - CPU frequencies are much faster than memory frequencies
    - Memory access takes many CPU cycles
      - Hundreds, in fact!
      - The latency of a load from memory will be in the 60-80ns range
- Cache hierarchy
  - Caches are an integral part of current processor designs
    - To reduce the impact of long memory latencies
  - Cache hierarchies are often multiple levels today
    - L1, L2, L3, sometimes L4
    - Levels are larger and slower further down the hierarchy

Motivation (2)

- Cache hierarchy works on principle of locality
  - Temporal locality – recently referenced memory addresses are likely to be referenced again soon
  - Spatial locality – memory addresses near recently referenced memory addresses are likely to be referenced soon
- Locality allows a processor to retrieve a large portion of memory references from the cache hierarchy
- Thus the programs we write should exhibit good locality!
  - Good news – this is typically true of well-written programs

Locality Example

- Similar to the loop scenarios we've discussed –

```
int A[N];
int sum = 0;
for (i=0; i<N; i++) {
    sum = sum + A[i];
}
```

- Data locality for sum and elements of array A[]
- Code locality for program instructions
  - Loops create significant inherent code temporal locality
  - Sequential streams of instructions create spatial locality

Important Cache Performance Metrics

- Miss Ratio
  - Ratio of cache misses to total cache references
  - Typically less than 10% for L1 cache, < 1% for an L2 cache
- Hit Time
  - Time to deliver a line in the cache to the processor
  - 1-2 CPU cycles for L1, 15-20 cycles for L2, ~40 cycles for L3
  - Related concept is “load-to-use” time
    - # of CPU cycles from the execution of a load instruction until execution of an instruction that depends on the load value
- Miss Penalty
  - Time required access a line from the next hierarchy level
- Average access time = hit time + (miss rate * miss penalty)

Cache Friendly Code

- Strongly-related to the benefits we discussed for certain loop transformations
- Examples:
  - Cold cache, 4-byte words, 4-word cache blocks

```
for (i=0; i < N; i++) {
    for (j=0; j < N; j++) {
        sum += a[i][j];
    }
}
```

Miss rate = ¼ = 25%

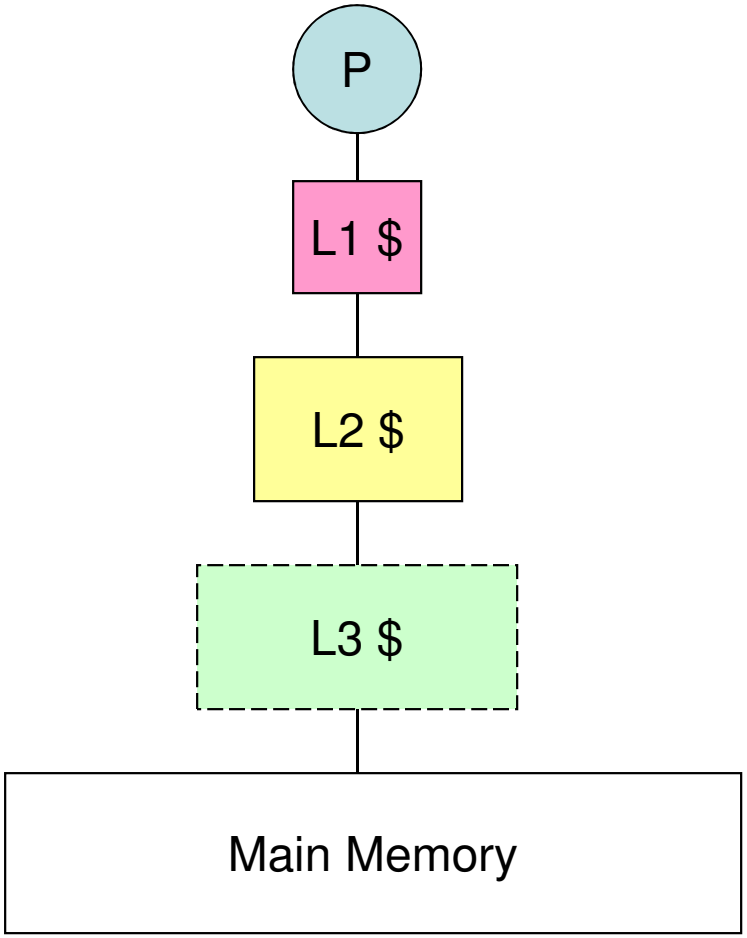
```
for (j=0; j < N; j++) {
    for (i=0; i < N; i++) {
        sum += a[i][j];
    }
}
```

Miss rate = 100%

Reverse-engineering a Cache

- Assume you have a machine
- You do not know its
  - Cache sizes (or number of levels of cache)
  - Cache block sizes
  - Cache associativity
  - Latencies
  - Data bandwidth
- We can find these out through test programs!
  - Write targeted code
  - Measure performance characteristics
  - Analyze the measurements

In-Class Exercise

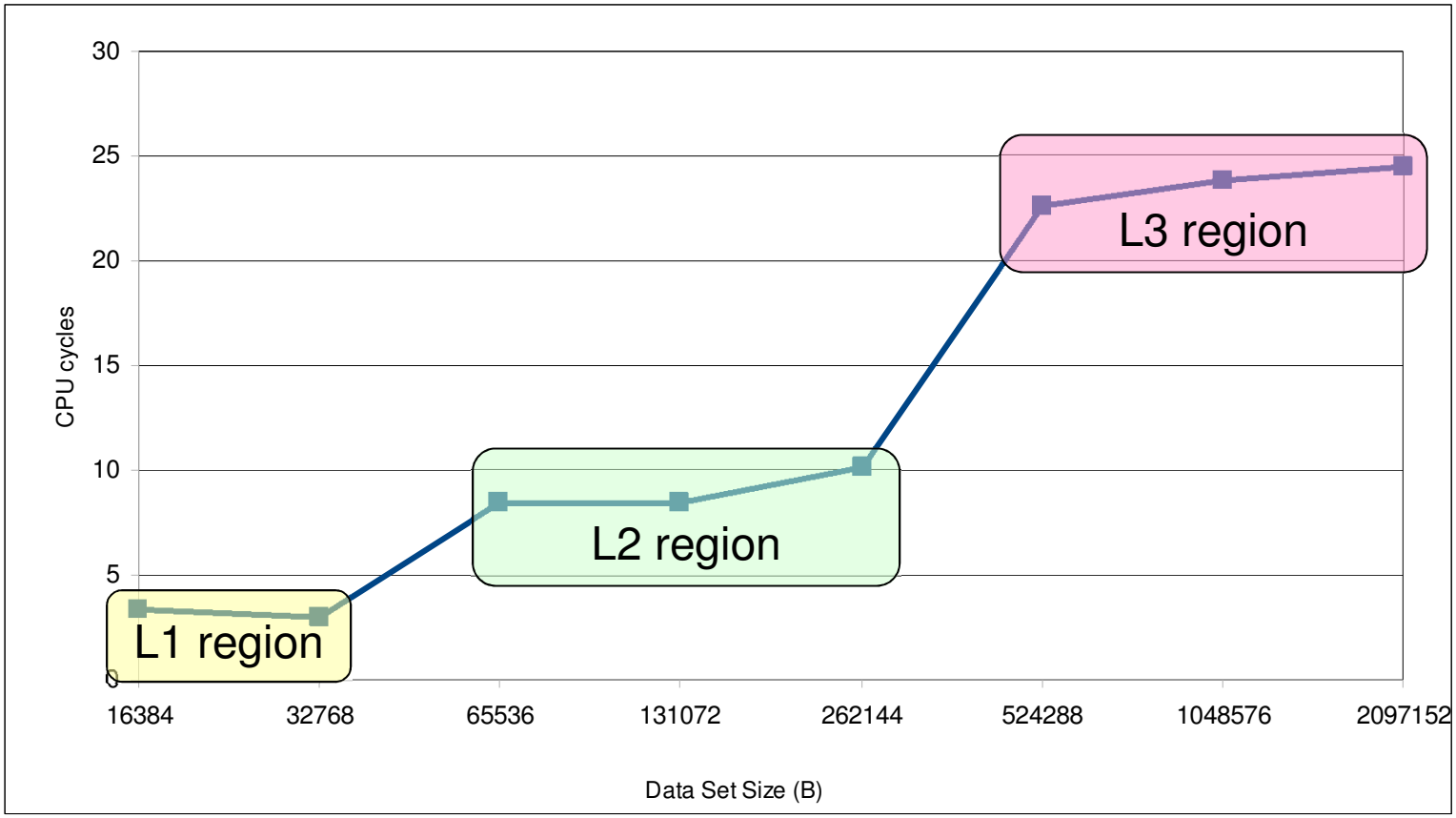


- Code a targeted test program to determine
  - # caches are in our machine
  - Size of each cache
  - Latency of each cache
- Assumptions
  - LRU replacement policy in use for each cache
  - Know cache block size
  - Each level of cache hierarchy has a different access latency

Test Code Summary

- (See links to code files on the class schedule page)
- Repeatedly access elements of a data set of some size
  - E.g. an array
  - Each memory access should depend on value from prior access
    - E.g. pointer chasing
    - This exposes memory latency of each access
- Record exectution time for this set of memory accesses
  - Calculate average latency from
    - measured time
    - known # of accesses
- When data set size grows larger than the size of a cache level –
  - We will see a step in the measured average access latency
  - Latency will stay constant while the data set size fits within a cache level
- Loop of repeated memory accesses can be unrolled
  - To reduce the interference from loop management instructions
- Access data set w/ fixed stride so each access touches a new cache block
- Can randomly cycle through data set elements to defeat prefetch
  - Prefetch could disrupt measurement and blur the transition between cache levels

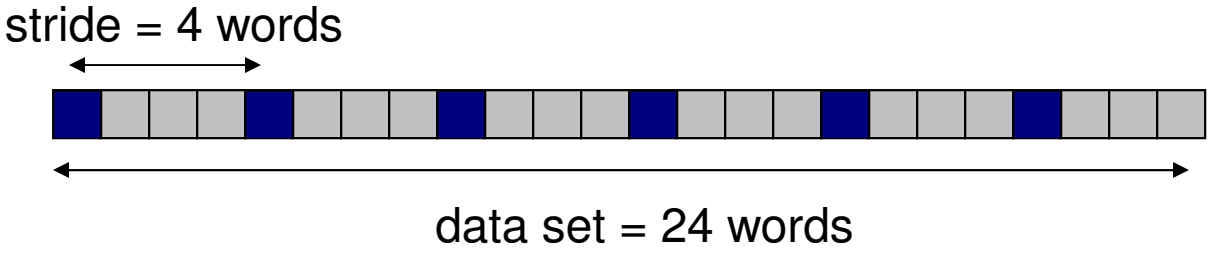
Example Results



* Measured on an Intel Core i7 CPU @ 2.4 GHz

* Program compiled with gcc -O3

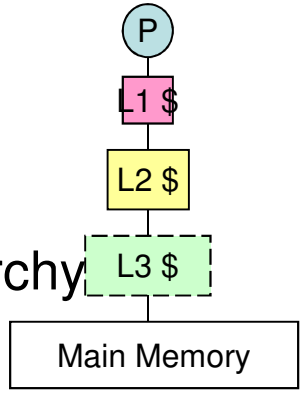
Cache Access Patterns



- Vary the data set accessed by our code
  - As data set grows larger than a cache level, performance drops
    - Performance can be measured as latency or bandwidth
- Vary the stride of data accessed by our code
  - Affects spatial locality provided by cache blocks
  - If stride is less than size of a cache line –
    - Initial access may cause cache miss, sequential accesses are fast
  - If stride is greater than size of a cache line –
    - Sequential accesses are slow

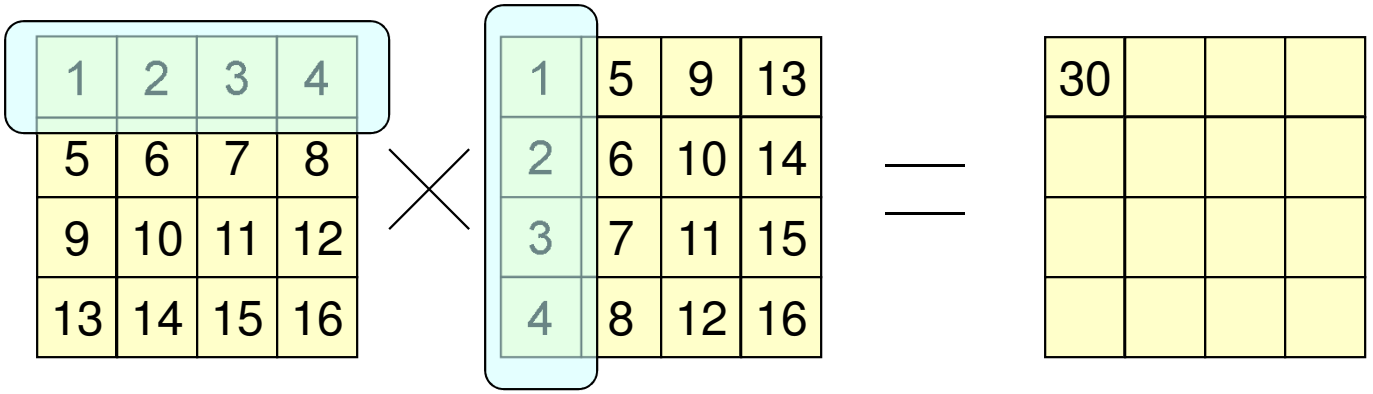
Memory Access Latency vs. Bandwidth

- Thus far, we've focused mostly on latency
  - Hit time for a cache level or memory
  - E.g. we put together example code using pointer chasing
    - Stresses the latency of each access
    - Only one memory access in flight at a given time
- Cache and memory bandwidth is also important
  - Bandwidth is a rate
    - Bytes per cycle
    - GB per second
  - Bandwidth gets smaller at lower levels of memory hierarchy
    - Just as latency grows larger
  - Some code is throughput sensitive, not latency sensitive
    - Code performance would improve with higher access bandwidth
      - Even if access latency increased



Matrix Multiplication

- Common operation in scientific applications
- Significant interaction with cache & memory subsystem

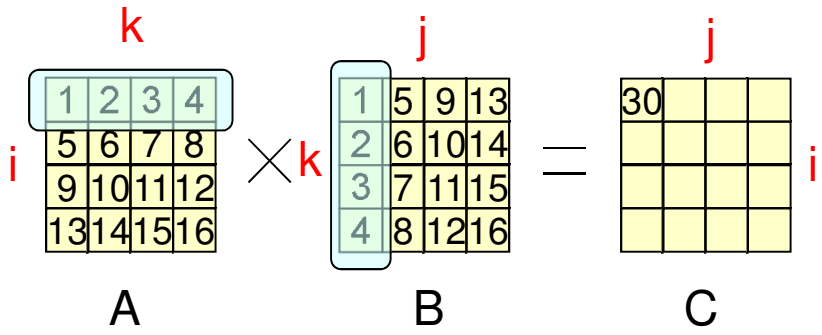


= 1*1 + 2*2 + 3*3 + 4*4

- Recall our memory layout discussion
  - E.g. C/C++ uses row-major order
  - 2D array is allocated as a linear array in memory

Matrix Multiplication Implementation

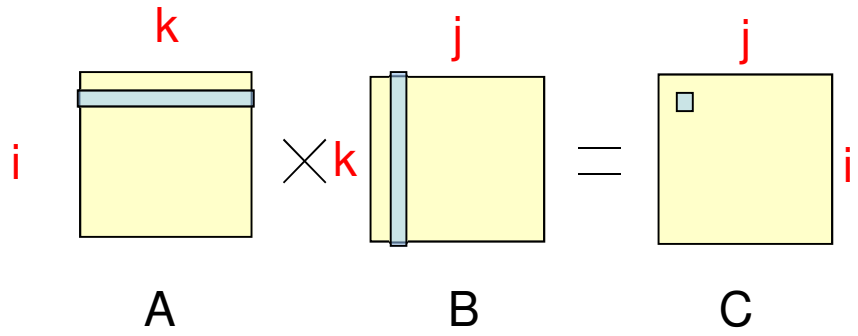
```
double A[N], B[N], C[N];
int i, j, k;
double sum;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        sum = 0;
        for (k=0; k<N; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```



- 3 Loops – i, j, k
  - 6 ways to arrange the loops and multiply the matrices
- $O(N^3)$ total operations
  - N reads for each element of A and B
  - N values to sum for each output element of C

Cache Analysis for Matrix Multiplication

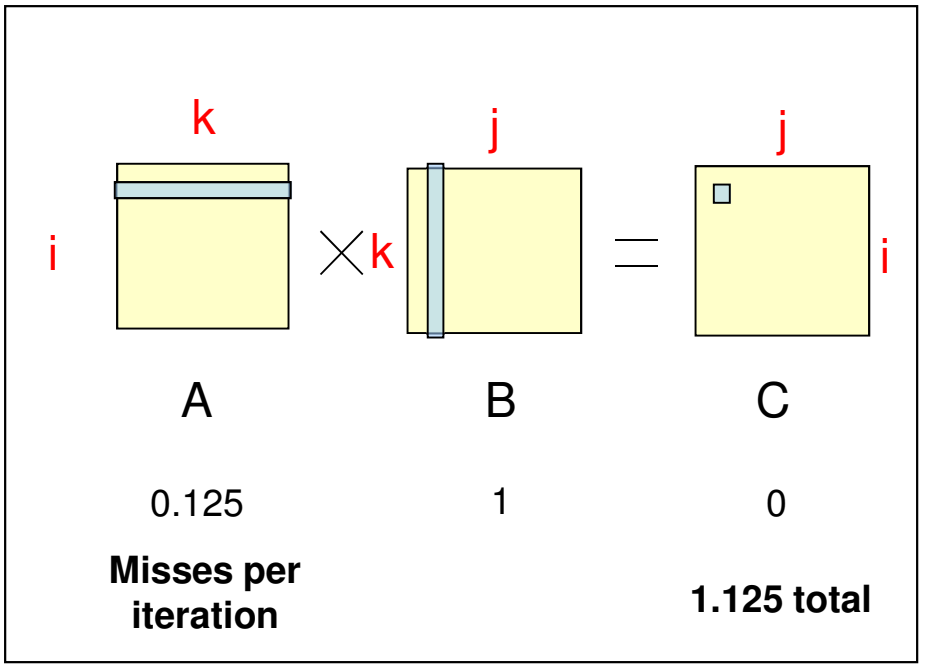
- Each matrix element is 64 bits (a double)
- Assumptions:
  - N is very large (cache cannot fit more than one row/column)
  - Cache block size = 64 bytes (8 matrix elements per block)
- Consider access pattern for i,j,k loop structure



  - A=good spatial locality; C=good temporal locality; B=poor locality

Matrix Multiplication

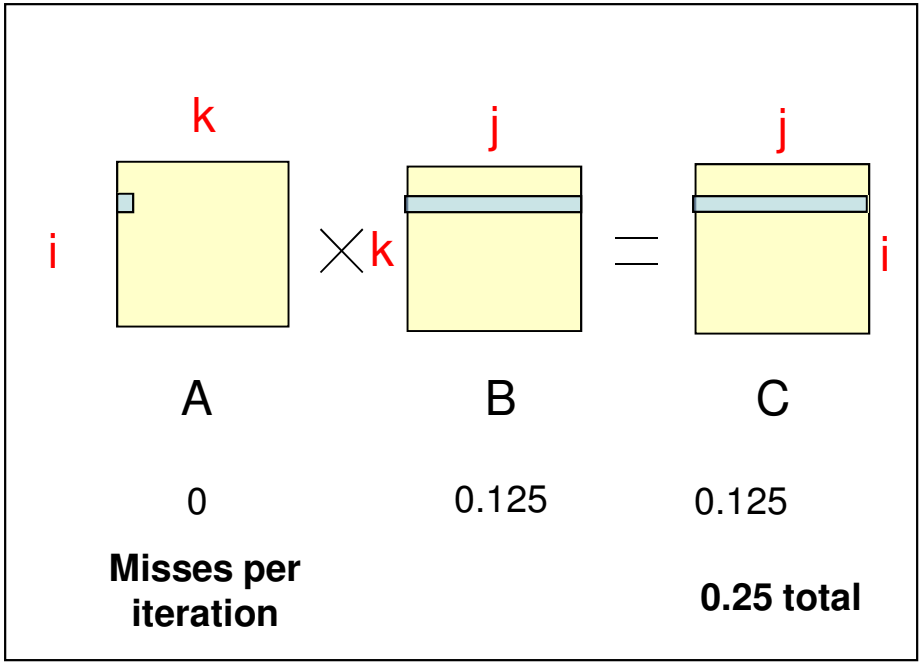
```
double A[N], B[N], C[N];
int i, j, k;
double sum;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        sum = 0;
        for (k=0; k<N; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```



- i-j-k
  - Memory accesses for each inner loop iteration
    - 2 loads: element A[i][k] and element B[k][j]
      - A[i][k] access will be cache miss every 8/64 iterations
      - B[k][j] access will be cache miss every iteration
- j-i-k cache miss behavior same as i-j-k

Matrix Multiplication

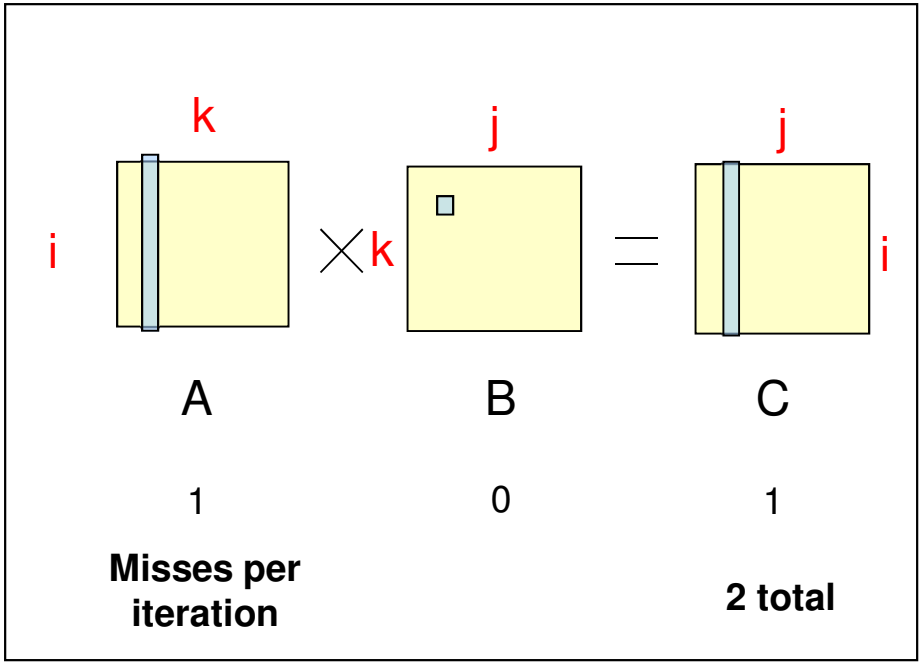
```
double A[N], B[N], C[N];
int i, j, k;
double tmp;
for (k=0; k<N; k++) {
    for (i=0; i<N; i++) {
        tmp = A[i][k];
        for (j=0; j<N; j++) {
            C[i][j] += tmp * B[k][j];
        }
    }
}
```



- k-i-j
  - Memory accesses for each inner loop iteration
    - 2 loads: element C[i][j] and element B[k][j]; 1 store: element C[i][j]
      - C[i][j] access will be cache miss every 8/64 iterations
      - B[k][j] access will be cache miss every 8/64 iterations
- i-k-j cache miss behavior same as k-i-j

Matrix Multiplication

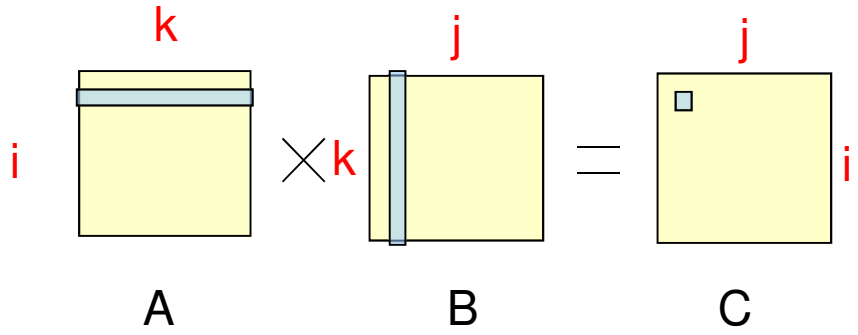
```
double A[N], B[N], C[N];
int i, j, k;
double tmp;
for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
        tmp = B[k][j];
        for (i=0; i<N; i++) {
            C[i][j] += tmp * A[i][k];
        }
    }
}
```



- j-k-i
  - Memory accesses for each inner loop iteration
    - 2 loads: element C[i][j] and element A[i][k]; 1 store: element C[i][j]
      - C[i][j] access will be cache miss every 8/64 iterations
      - B[k][j] access will be cache miss every 8/64 iterations
- k-j-i cache miss behavior same as j-k-i

Matrix Multiplication Summary

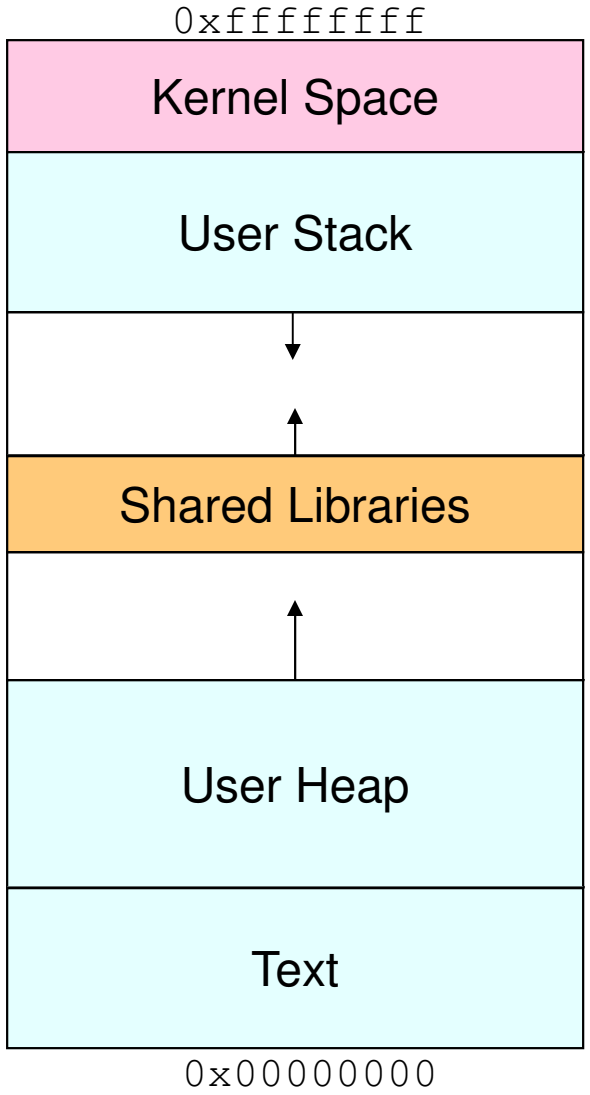
- k is innermost loop
  - A = good spatial locality
  - C = good temporal locality
  - Misses per iteration
    - 1 + (element sz/block sz)
- i is innermost loop
  - B = good temporal locality
  - Misses per iteration
    - 2
- j is innermost loop
  - B, C = good spatial locality
  - A = good temporal locality
  - Misses per iteration
    - 2 * (element sz/block sz)



Other Types of Caching

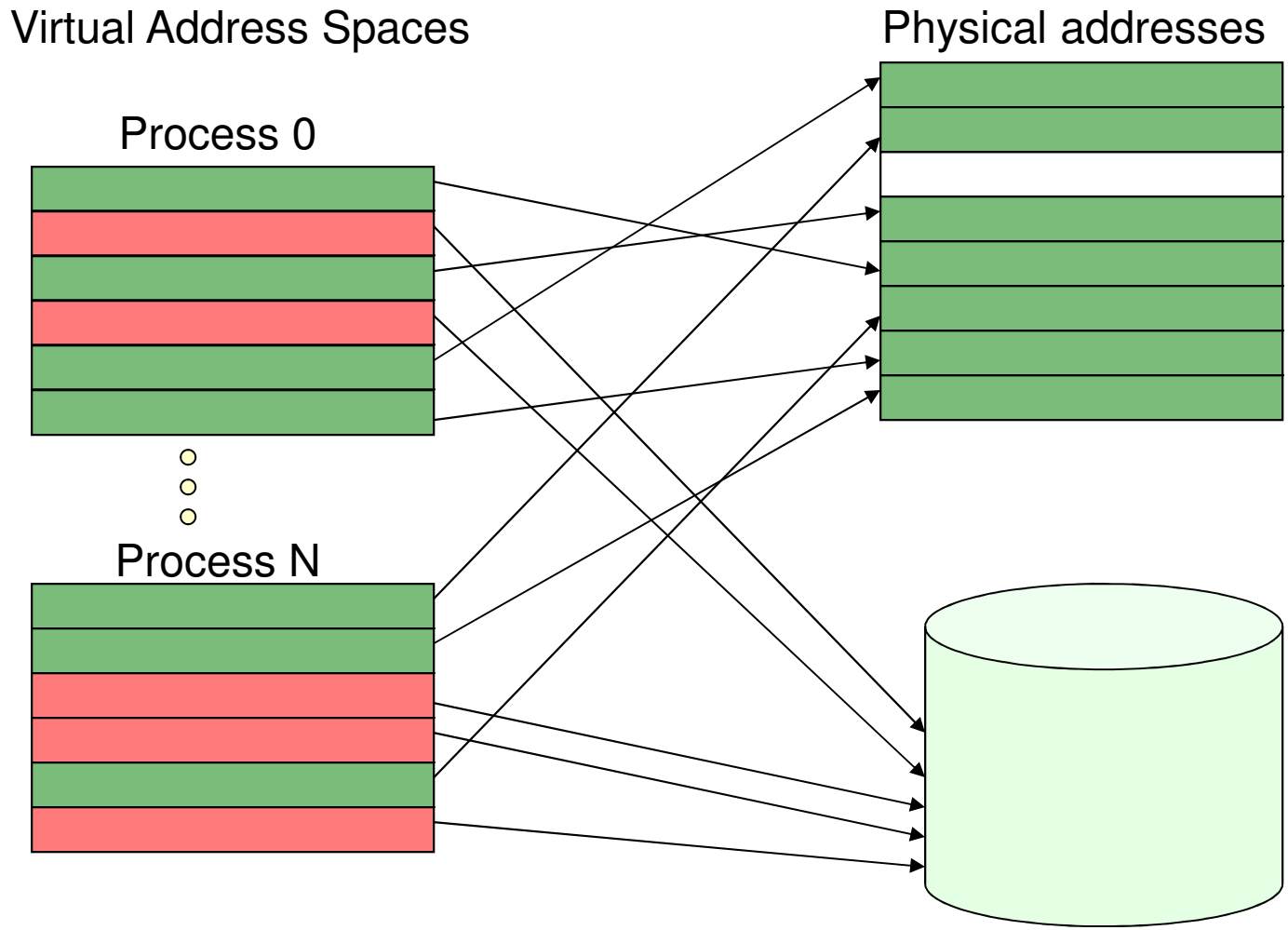
- Main memory is a cache for disk
  - Operates at a physical page granularity
- TLB is a cache for page table
  - Translation Lookaside Buffer
  - Operates at a page table entry granularity

Virtual Address Space

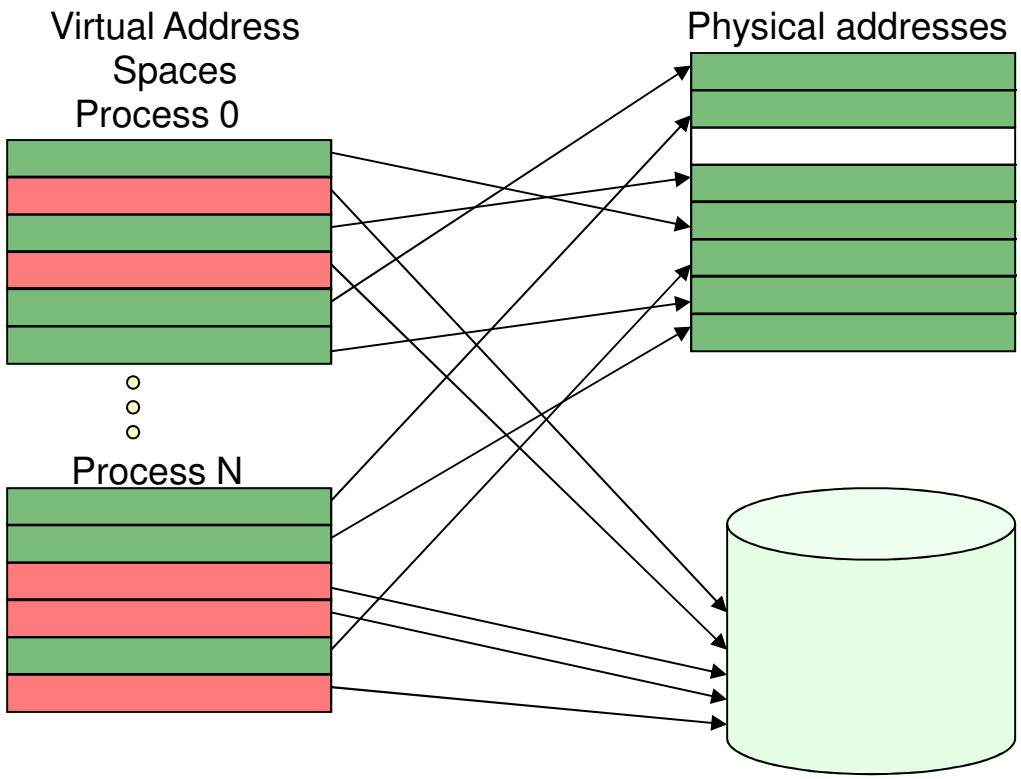


- Each process thinks it has access to the full address space provided by the machine
  - 4GB on 32-bit computers
  - ~16-256TB on 64-bit computers
  - Illusion provided by OS
- Every process has its own virtual address space
  - Contents visible only to that process
- Address space for even one process is larger than physical memory
- How does it fit?

Physical Memory as a Cache for Disk



Physical Memory as a Cache for Disk



Two Key Performance Aspects:

1) Which addresses from various processes should we keep in physical memory?

2) How do we do this mapping between a virtual address and its address in physical memory?

What To Keep in Physical Memory?

- Want to service memory accesses from DRAM, not disk
  - That is, if the access misses in all caches already
  - Disk is thousands of times slower than DRAM
    - ~60-80ns to several milliseconds
- Locality still rules the day
  - Just as we discussed for caches
- Want to capture spatial and temporal locality in memory
  - Temporal: retain recently accessed addresses in physical mem
  - Spatial: fetch nearby addresses into physical mem on disk access

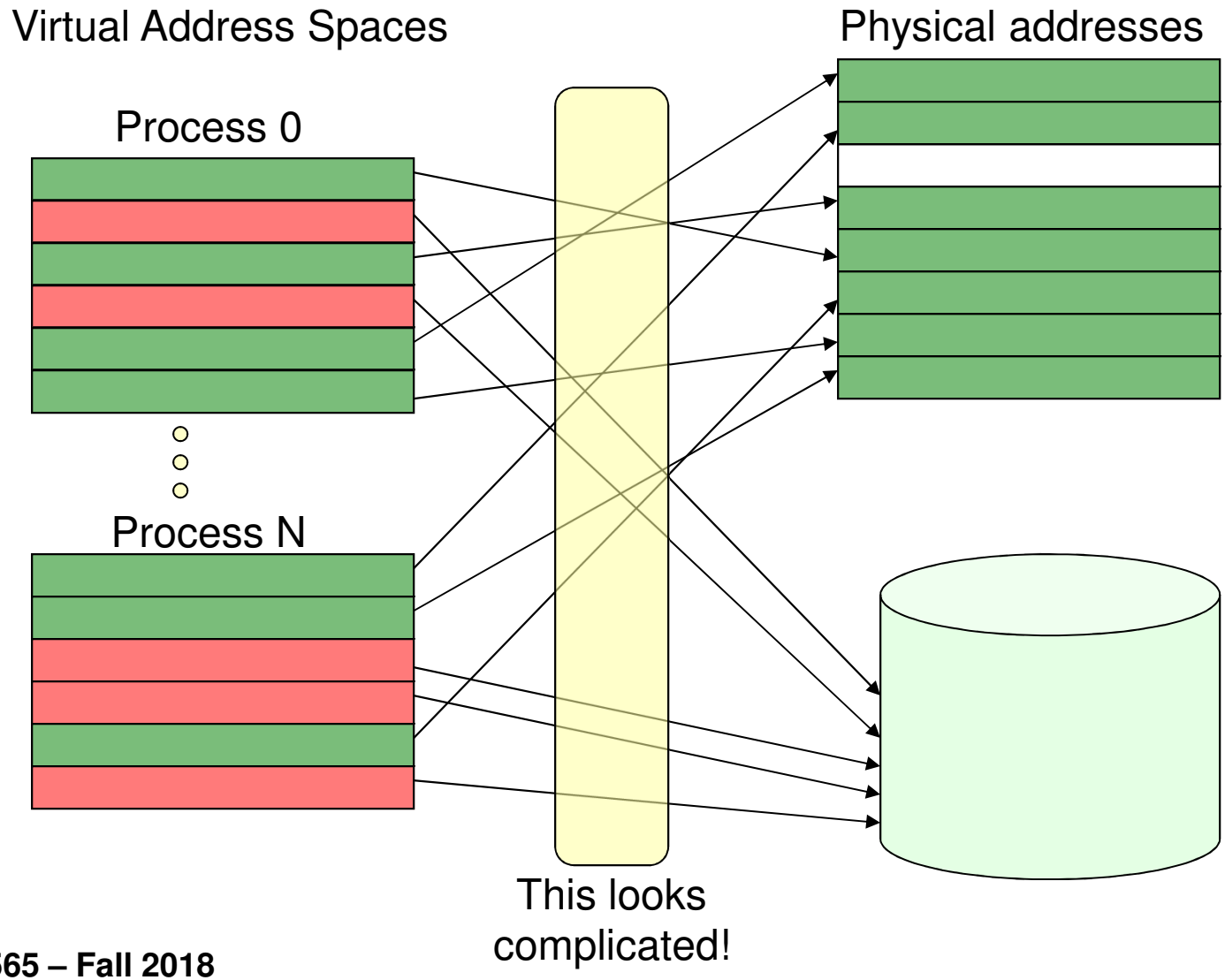
What To Keep in Physical Memory?

- OS manages the physical memory and disk accesses
  - Physical memory management is software-based
  - Operates on physical memory in units of **pages**
    - Pages have some size of 2^P (e.g. 4KB)
    - Much larger than cache block size due to huge latency of disk
  - OS treats physical memory as a fully associative cache
    - A virtual page can be placed in any physical page
  - Page replacement policies may be complex
    - SW can maintain and track much more state than HW
    - Since accessing a page from disk is so slow there is lots of time!
- Physical memory holds the large-scale working set of a program
  - Working set size of less than physical mem size
    - Good performance for a process after memory is warmed up
  - Working set sizes of all active processes greater than physical mem size
    - Can cause a severe performance problem
    - Called **thrashing**: pages are continuously swapped between memory and disk

Metrics

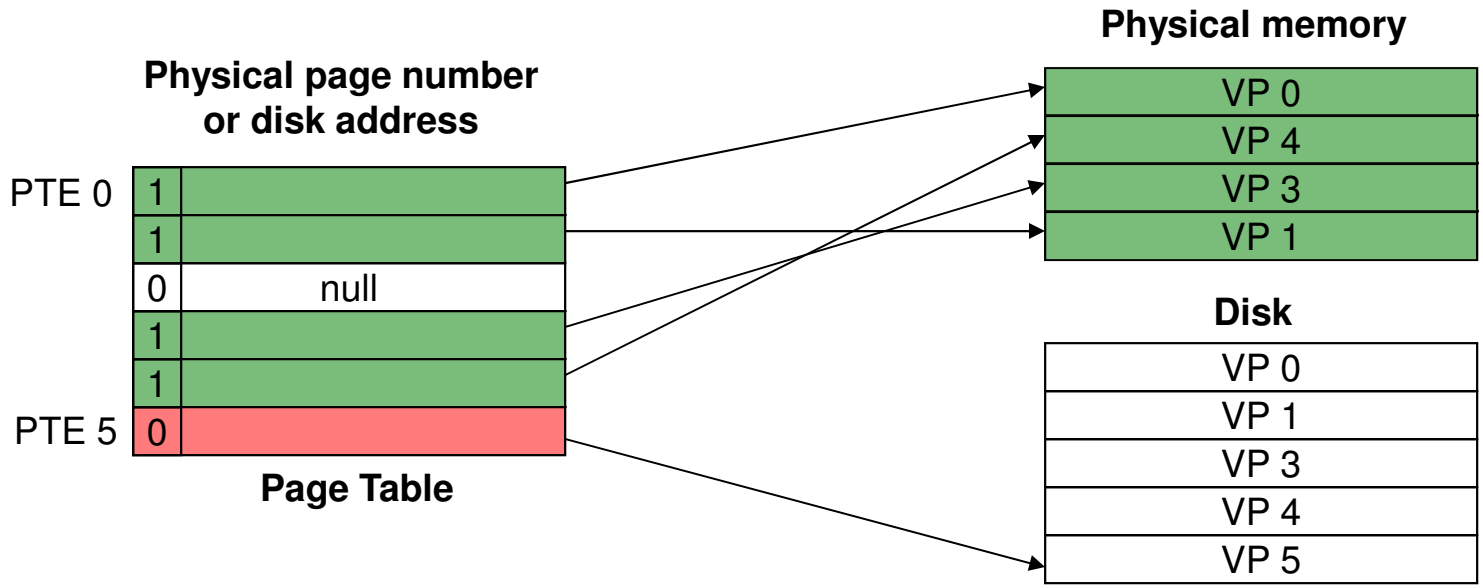
- Page hit
  - Memory reference to an address that is stored in physical mem
- Page miss (page fault)
  - Reference to an address that is not in physical memory
  - Misses are expensive
    - Access to disk
    - Software is involved in managing the process

Physical Memory as a Cache for Disk



What is Stored Where in Physical Memory?

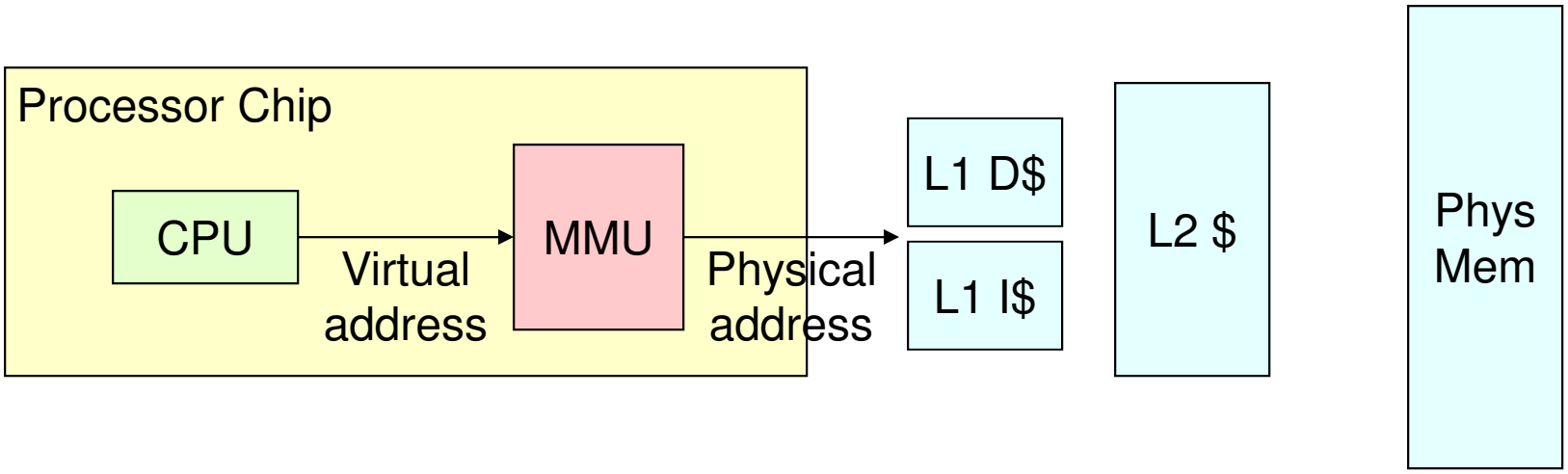
- Need to remember current location for *every* virtual page
  - Since physical mem is managed as fully associative cache
- Solution is called a **page table**
  - Maps virtual pages to physical pages
  - Per-process software data structure



Page Table Management

- Page tables are very large
  - One entry per page
  - For 64-bit address space:
    - Assume 4 GB total virtual memory (2^{32})
    - Assume 4KB pages
    - 2^{32} / 2^{12} = 2^{20} entries
    - PTE is 4B in x86 architecture = 2^{22} bytes
    - And that's just for one process!
- Keep portions of the page table in memory; rest on disk
  - The frequently and recently accessed portions, that is

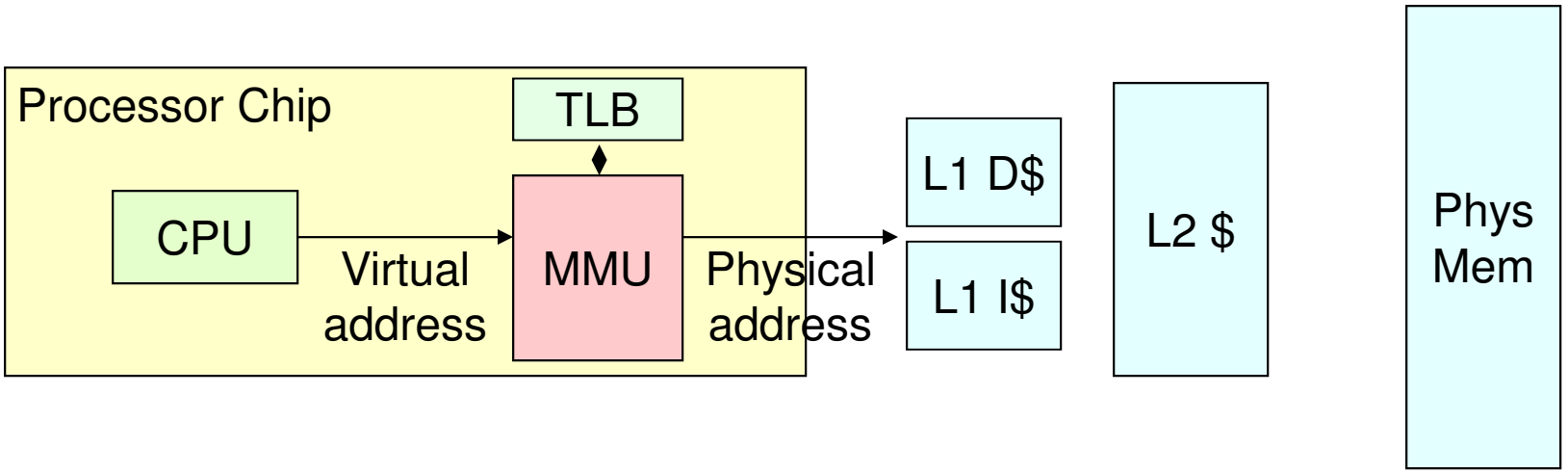
But That's Not Quite Enough



- Physical addresses are needed for cache lookups
  - Beginning at the L1 cache
- PTE is needed to turn a VA into PA
  - PTE is located in memory (at best)
  - Memory access required for every load or store?

Translation Lookaside Buffer (TLB)

- TLB is a very fast cache of PTEs
  - Located inside the MMU of a processor
  - Go directly from virtual page to physical page address
- Hierarchy of TLBs in current processor designs
  - Separate instruction & data L1 TLBs, L2 TLB



TLB Reach

- TLB Reach
  - Amount of memory accessible from the TLB
  - Should cover the working set size of a process
  - (# TLB entries) * (Page size)
- For example
  - 64 TLB entries in L1 DTLB * 64KB pages = 4MB reach

Increasing TLB Reach

- What if we need to cover larger working sets?
- We can't increase the number of TLB entries
  - Well, we could wait a few years for a newer processor
- We can increase the page size
  - Most modern architectures support a set range of page sizes
  - From 4KB to 16MB
- Example - hugepages
  - Consult your favorite OS manual to turn on hugepages
  - RHEL example –
  - E.g. libhugetlbfs

```
$ grep Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
$ grep HugePages_Total /proc/meminfo
HugePages_Total:       0
$ sysctl -w vm.nr_hugepages=128
$ grep HugePages_Total /proc/meminfo
HugePages_Total:     128
```