Memories

EE685, Fall 2025

Hank Dietz

http://aggregate.org/EE685

UNIVERSITY OF KENTUCKY

Memory Terminology

- Volatile – power off, data fades away
- ROM – non-volatile Read Only Memory
- PROM, EPROM, OTP, EEROM, Flash, 3DXPoint – types of non-volatile programmable memory
- RAM – Random Access Memory (mostly volatile)
  - Core – non-volatile magnetic RAM technology
  - SRAM – Static RAM, fast but big cells
  - DRAM – Dynamic RAM, slow but small cells
  - EDO, SDRAM, DDR, RamBus – DRAM types
  - CXL – Compute eXpress Link
- Registers, Cache – fast working memories

More Memory Terminology

- Punched cards
- Punched paper tape
- Tape, Magtape
- Drum
- Disks:
  Floppy, Hard, Magneto-optical, Compact Disc, Digital Video (Versatile?) Disc, Blu-ray
- Solid State Disk, Optane

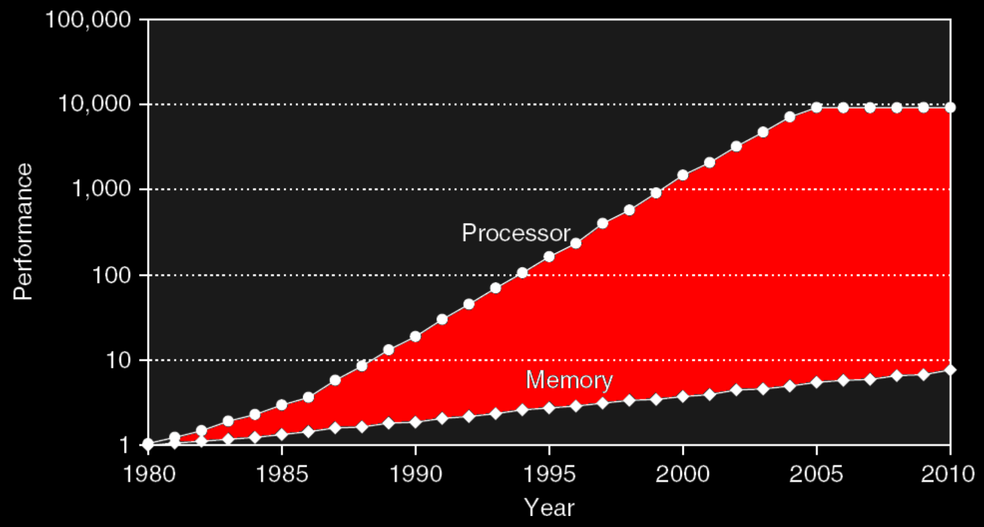
What we want, what we have

- What we want:
  - Unlimited memory space
  - Fast, constant, access time
    (UMA: Uniform Memory Access)
- What we have:
  - Memories are getting bigger
  - Growing complexity memory hierarchy
  - Temporal and spatial locality issues
    (NUMA: Non-Uniform Memory Access)

UK UNIVERSITY OF KENTUCKY

Memory speed hasn't kept up

- Pre-1970s memory faster than processor...

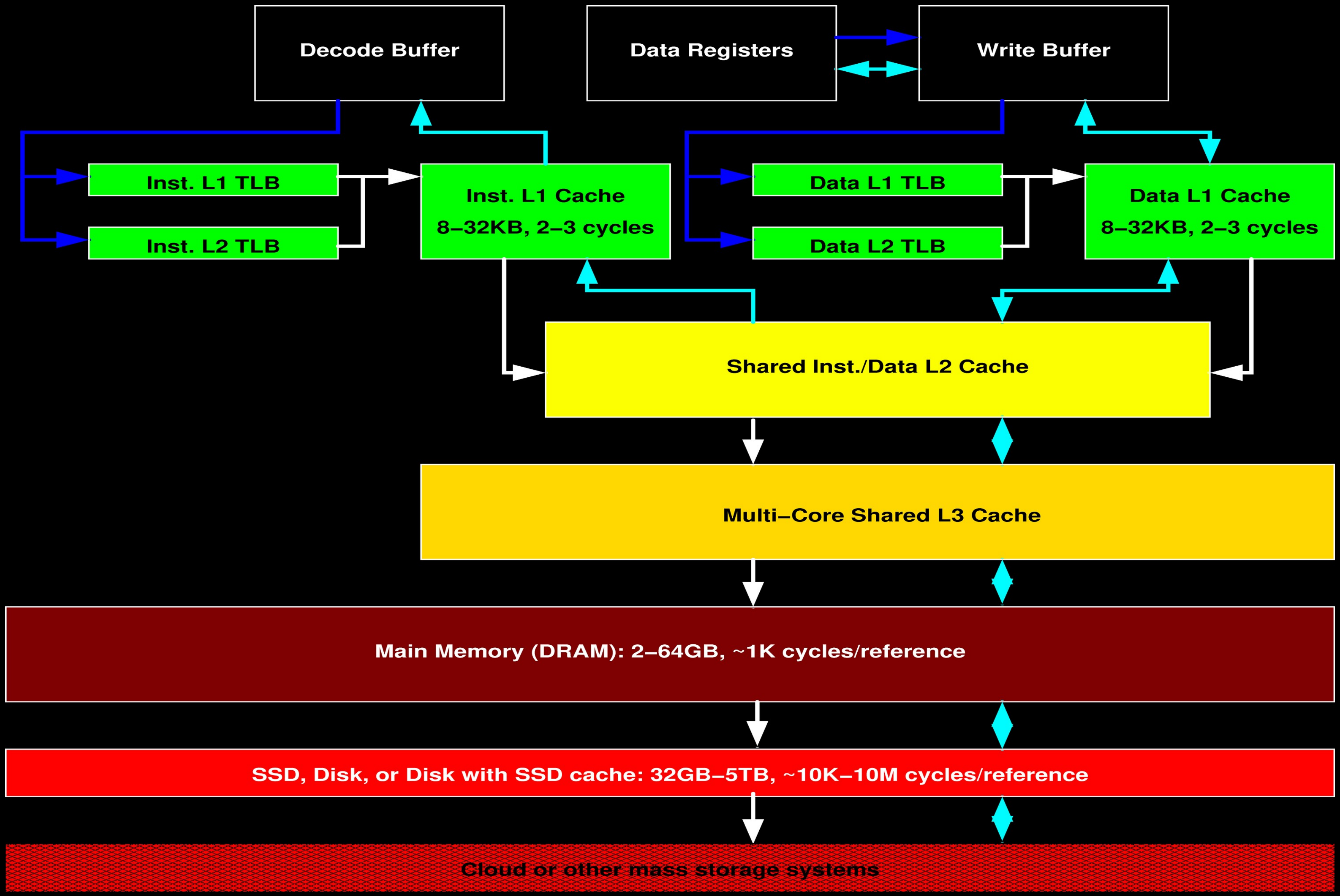


- Now thousands of times slower to access

Multi-core processor chips

- E.g., Intel Core i7 generates up to 2 refs/clock for each of 4 cores @ 3.2GHz
  - 25.6G 64-bit data refs/s
  - 12.8G 128-bit inst. refs/s

  Total 409.6GB/s… but DRAM is 25GB/s!
- Multi-port pipelined cache
- Multiple levels of caching
- Logic to support sharing
- Large fraction of area & power budget

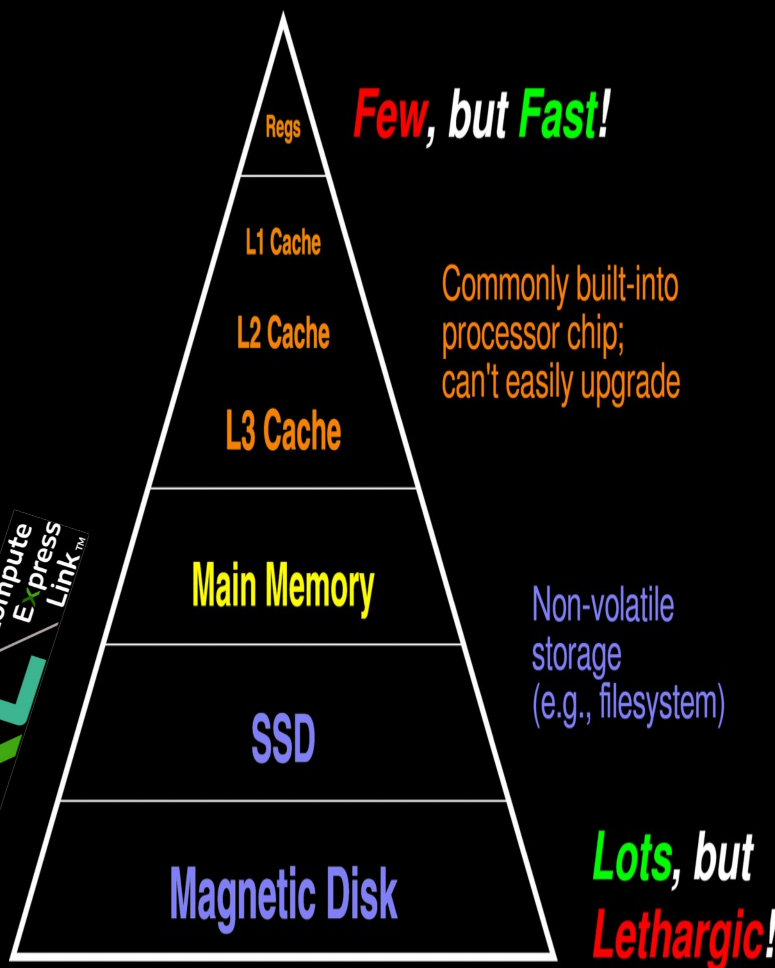


Decode Buffer
Data Registers
Write Buffer
Inst. L1 TLB
Inst. L2 TLB
Inst. L1 Cache
8–32KB, 2–3 cycles
Data L1 TLB
Data L2 TLB
Data L1 Cache
8–32KB, 2–3 cycles
Shared Inst./Data L2 Cache
Multi–Core Shared L3 Cache
Main Memory (DRAM): 2–64GB, ~1K cycles/reference
SSD, Disk, or Disk with SSD cache: 32GB–5TB, ~10K–10M cycles/reference
Cloud or other mass storage systems

The Memory Hierarchy

Regs: a few kB, 1 cycle
L1 Cache: 64kB, ~4 cycles
L2 Cache: 2MB, ~12 cycles
L3 Cache: 16MB, ~43 cycles
Main Memory: 32GB, $4/GB,
    ~248 cycles
SSD: 512GB, $0.10/GB,
    ~200k cycles,
Magnetic Disk: 14TB,
    $0.018/GB, ~20M cycles

CXL Compute Express Link™



How The Hierarchy Helps

- Main memory is too slow & too small; we want:
  - Capacity & cost of the big stuff (e.g., disk)
  - Access speed of the fast stuff (e.g., regs)
- If most things are in the top layers when we want to access them, this works…
  this is what we call good locality of reference
- Two basic types of **locality**:
  - **Temporal**: same thing accessed again soon
  - **Spatial**: nearby thing accessed soon

Managing The Hierarchy

- Everything "lives" in the bottom layer (e.g., disk)
- Drop copies in higher layers to access faster
  - SSD and disk are slow enough that OS software can manage copying
  - Caches need hardware management
  - Register copy management is explicitly done by the compiler via load/store instructions (GPUs & microcontrollers often also have local memories managed by the compiler)

Terms

- Cache line: one block of data in cache
- Dirty line: a block with value different from that block in lower memory levels
- Hit: data found here
- Hit rate or hit ratio: # hits / # references total
- Hit time: RAM access time + check hit/miss
- Miss: data not here, must forward read request
- Miss rate: # misses / # references total
- Miss penalty: time to replace line & deliver data

Basic cache design issues

- Placement (mapping)
- Identification:
  which line within the set do I want?
- Replacement policy:
  which line gets kicked-out to make space?
- Write strategy

Placement / Mapping

- Caches are basically hash tables indexed by hash value of the address requested
- Direct mapped:
  Each bucket holds one line
- Set associative:
  Each bucket holds set-size lines
- Fully associative:
  Only one bucket, which holds all lines

Mapping Addresses

- Each address is {*line number*} {*byte offset in line*}; 32B line → [4:0] are *offset*
- Cache set (bucket) index is hash({*line number*}), often contiguous bits from {*line number*}
- Line tag field holds bits of {*line number*} that are not implied by bucket index

E.g., a 4-way 16KB cache might be:
512 lines, each 32B long, with 128 buckets
Address[4:0] is offset, [11:5] is bucket index

Best replacement policy?

- Direct mapped → no choice
- Random
- Replace a clean (not dirty) line
- LRU (Least Recently Used): mark when line is accessed, replace not accessed recently
- LFU (Least Frequently Used)
- MRU and MFU: Most “”
- Belady’s MIN: replace line not used for the longest time in the future (how to know this?)
- Compiler-driven; e.g., using cache bypass

Best replacement policy?

- Sample comparison of LRU vs. Random
- Miss rate %:

	LRU	Ran	LRU	Ran	LRU	Ran
Size	2-way		4-way		8-way	
16KB	5.2	5.7	4.7	5.3	4.4	5.0
64KB	1.9	2.0	1.5	1.7	1.4	1.5
256KB	1.15	1.17	1.13	1.13	1.12	1.12

Write strategy

- Write through
  - Write always goes to main memory
  - Easy; needed for I/O devices in memory
- Write back
  - Write only when line replaced, saving traffic
  - Could do lazy writes when not busy
  - May need to read on miss to get rest of line
- Write allocate: write back, but don't wait for line to be read first; aka pre-arrival caching

Write Buffer

- Sort-of like a "level 0 data cache" (faster because no TLB in front of it)
- Buffer can re-group writes to form write to a larger fraction of a line (not just one byte or word)
- Need to be careful about task switches, etc.; may have to flush write buffer often

What causes a miss?

- Compulsory
  - Never touched this block before
  - Shared fetch effect can avoid these when another process touches what I want first
- Capacity
  - Could have been from cache, but didn't fit
- Conflict
  - Could have fit, but cache mapping had a conflict with another line that caused this line to be replaced (e.g., direct mapped)

Cache optimizations

- Larger total cache size
  - Fewer capacity & conflict misses
  - Dumber replacement policy works ok
  - Increases hit time, die space, and power use
- Larger line size
  - Fewer compulsory misses (spatial locality)
  - More capacity & conflict misses
  - Increases miss penalty (block transfer time)

More cache optimizations

- Higher associativity
  - Reduces conflict misses
  - Increases hit time & power use
- More levels of cache
  - Smaller, faster, upper-level caches
  - More complex hardware structure

Still more cache optimizations

- Priority to read misses over writes
  - Reduces miss penalty
  - Modest increase in design complexity
- Avoiding address translation before indexing
  - Reduces hit time
  - Not what operating systems expect
  - Frequent cache flushes or need PID tags

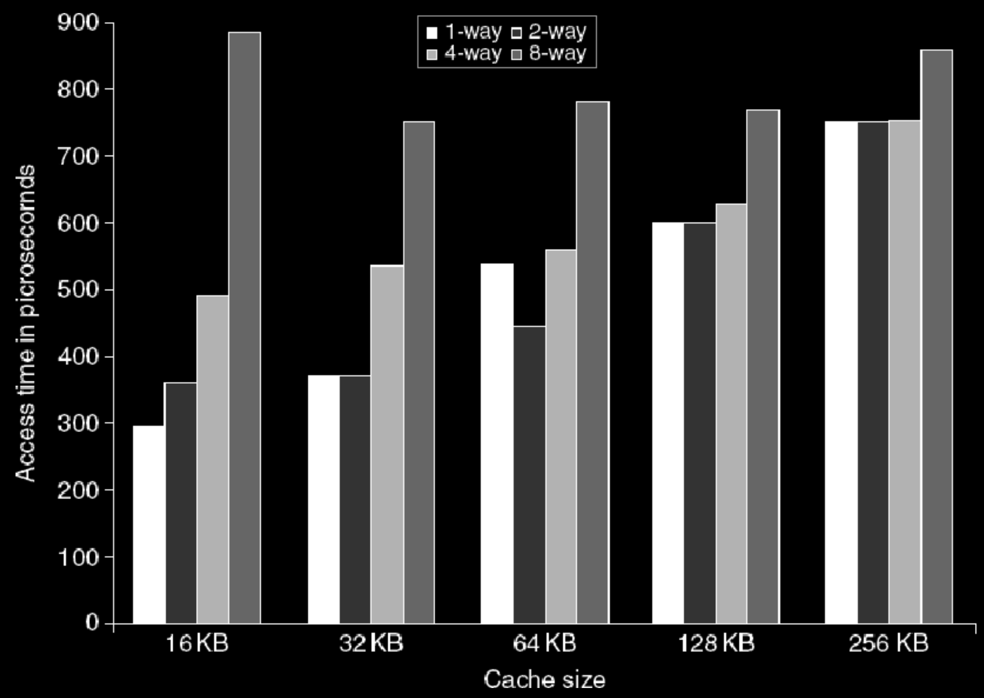
Advanced cache optimizations

Small & simple L1 caches

- Critical timing path is
  Access tags → compare tags → select line
- Direct-mapped can overlap tag compare with transmission of line data
- Lower associativity reduces power
  (fewer comparators, narrower data access)

Advanced cache optimizations

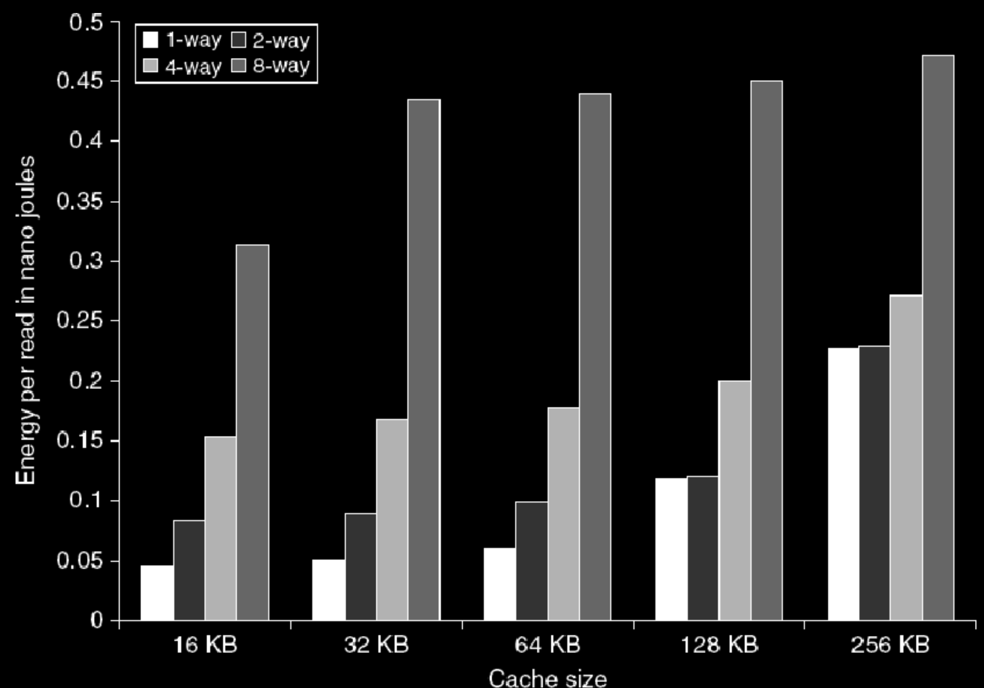
Small & simple L1 caches



Access time vs. L1 size and associativity

Advanced cache optimizations

Small & simple L1 caches



Read energy vs. L1 size and associativity

Advanced cache optimizations

Way prediction

- Used in MIPS R1000, ARM Cortex-A8
  - Helps where tag compares are serialized
  - Mispredict increases hit time
  - Accuracy >90% for 2-way, >80% 4-way
  - Inst. cache more predictable than data
- Way selection predicts line data and tag

Advanced cache optimizations

Pipelined cache

- Improves bandwidth
- Easier to do higher associativity
- Branch mispredict time increases
- Examples:
  Pentium was 1 cycle
  Pentium Pro – III were 2 cycles
  Pentium 4 – Core i7 are 4 cycles

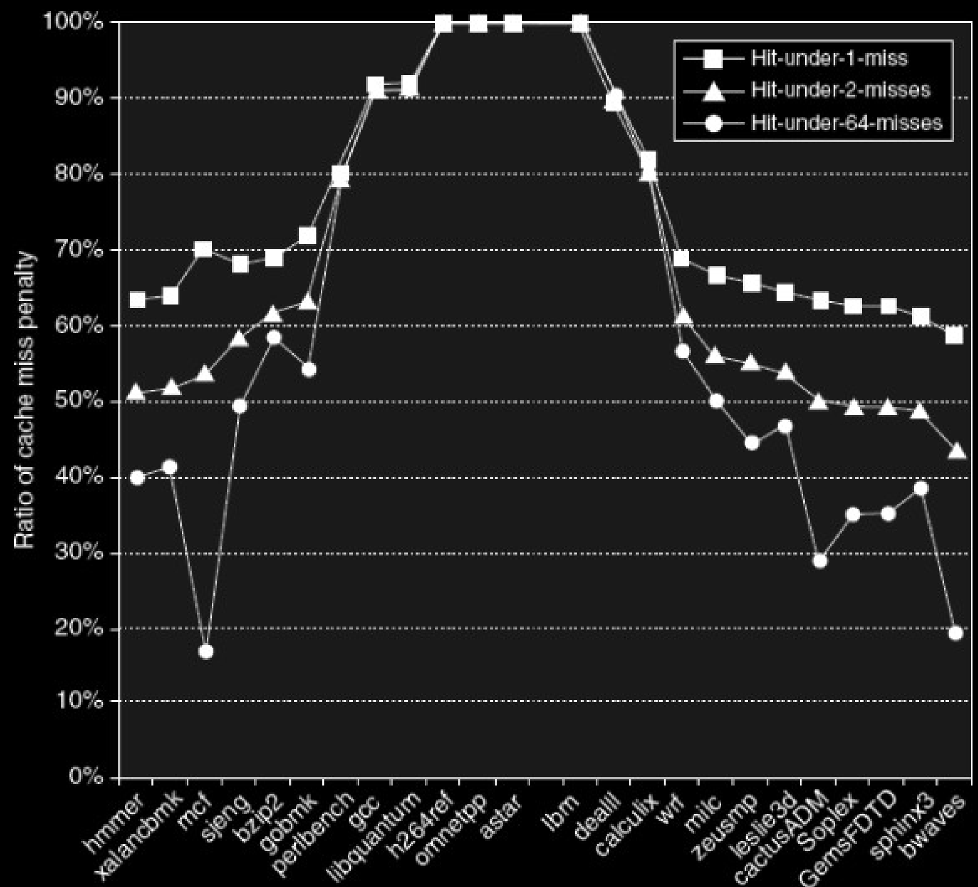
Advanced cache optimizations

Non-blocking cache

- Allows hits before previous misses complete
  - "Hit under miss"
  - "Hit under multiple miss"
- Required for L2 caches
- Processors can't hide L2 miss penalty

Advanced cache optimizations

Non-blocking cache



UNIVERSITY OF KENTUCKY

Advanced cache optimizations

Multi-banked cache

- Fragment cache into independent banks
  - ARM Cortex-A8 supports 1-4 banks of L2
  - Intel i7 supports 4-bank L1, 8-bank L2
- Banks commonly interleave on low bits of line address (sequential interleaving):

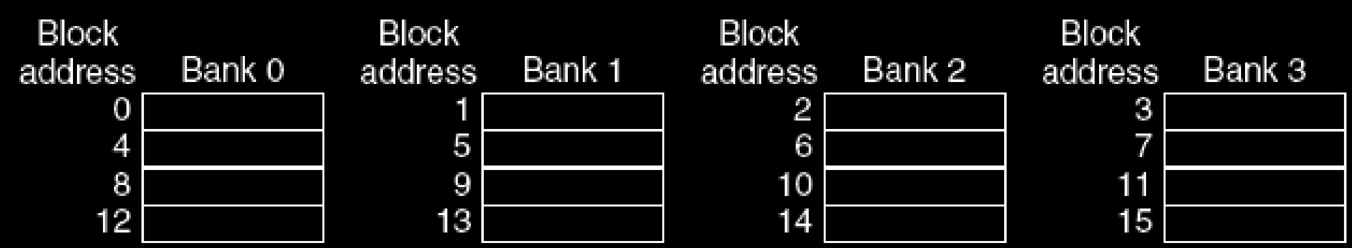


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Advanced cache optimizations

Critical word first

- Requested word in line is fetched first
- Requested word is returned immediately upon arrival at the cache – don't wait for full line
- Other words of line fetched in some order
- Can use rotated order
  - Start at word k, ith fetch is (k+i)%n
  - Sort-of like assuming sequential prefetch

Advanced cache optimizations

Early restart

- Requests words in normal order
- Requested word is returned immediately upon arrival at cache – don't wait for full line
- Potentially simpler than critical word first, probably not as effective…

Advanced cache optimizations

Merging write buffer

- This is the write buffer described earlier… essentially a FIFO of lines
- Does NOT always treat write buffer as a FIFO
  - Each line tracks which fields are “present”
  - Collects word writes into the same line entry
  - I/O addresses must still be FIFO
- Increases effective size of write buffer
- If entire line is present, cache doesn’t need to read the missing parts

Compiler optimizations

Linker optimization

- Changing link order can change caching by changing which addresses conflict in cache
  - If f() calls g(), different buckets for f() and g()
  - Profiling to detect conflict pattern
- Same idea can be used to pick addresses for data structures

Compiler optimizations

- Restructure code to change data access pattern
  - Group data (data layout)
  - Reorder accesses (loop transformations)
- Prevent cache pollution
  - Why cache what you get from a register?
  - Often double-map: cache / bypass
- Avoid saving data that isn't used again

Compiler optimizations

Merging/splitting arrays

- Array elements accessed together can be grouped together to enhance spatial locality
- Also separate those not accessed together

E.g., suppose `a[i]` and `c[i]` accessed together:

```
int a[N], b[N], c[N];
struct { int a, b, c; } abc[N];
struct { int a, c; } ac[N]; int b[N];
```

Compiler optimizations

Loop interchange

- Loop nest traversal order matches data layout
- Improves spatial locality

E.g., if `a[0][0]` is next to `a[0][1]`:

```
for (i=0; i<N; ++i)
  for (j=0; j<M; ++j) a[i][j] = 0;
for (j=0; j<M; ++j)
  for (i=0; i<N; ++i) a[i][j] = 0;
```

Compiler optimizations

Loop fusion

- Fuse loops that work on similar data
- Improves spatial locality

```
for (i=0; i<N; ++i)
  for (j=0; j<M; ++j)
    a[i][j] = b[i][j] + c[i][j];
for (i=0; i<N; ++i)
  for (j=0; j<M; ++j)
    d[i][j] = a[i][j] * c[i][j];
```

```
for (i=0; i<N; ++i)
  for (j=0; j<M; ++j) {
    a[i][j] = b[i][j] + c[i][j];
    d[i][j] = a[i][j] * c[i][j]; }
```

Compiler optimizations

Loop blocking/stenciling

- Iterate in pattern that maximizes reuse

```
for (i=0; i<N; ++i)
  for (j=0; j<N; ++j) {
    r = 0;
    for (k=0; k<N; ++k)
      r += y[i][k] * z[k][j];
    x[i][j] = r; }
```

Compiler optimizations

Loop blocking/stenciling

- Iterate in pattern that maximizes reuse

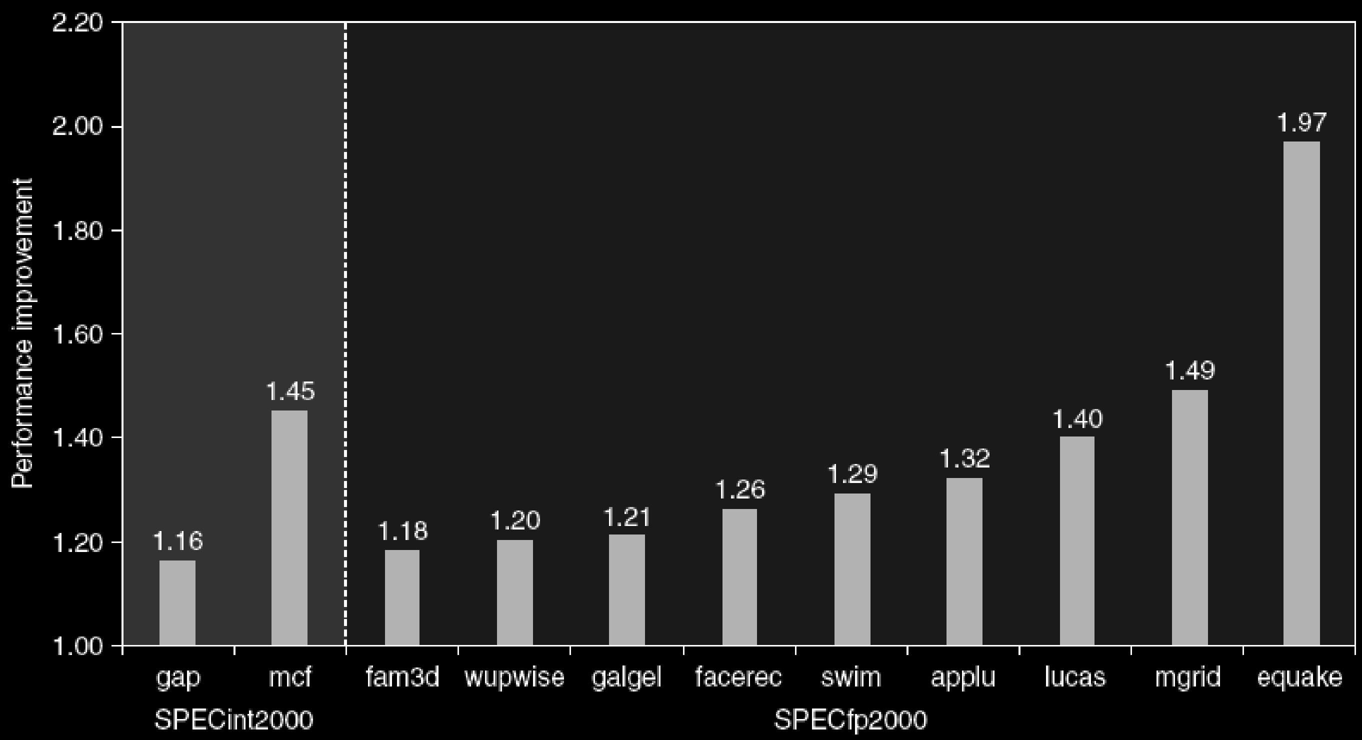
```
for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for (i=0; i<N; ++i)
      for (j=jj; j<min(jj+B,N); ++j) {
        r = 0;
        for (k=kk; k<min(kk+B,N); ++k)
          r += y[i][k] * z[k][j];
        x[i][j] += r; }
```

Prefetching

- Software (by compiler)
  - Hoist load to earlier position in program
  - *Suggest* hardware load into cache
- Hardware
  - Assume or recognize reference pattern and request expected next early
  - Line +/-1, strided, other patterns
- Works better for instructions than data
- Generally can abort a prefetch to cache, Prefetches can't fault (no exceptions)

Prefetching

- Fetch line and next line on a miss (Pentium 4)



Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Consistency Models

- The `volatile` keyword in C/C++ gives potential memory order constraints
- Strict: everybody sees result at next tick
- Sequential: everybody sees things as if they happened in a sequential order
- Weak Ordering: memory barriers/fences force ordering of before vs. after
- Transactional Memory: uses cache to buffer speculative memory writes, signal conflicts

Cache Coherence

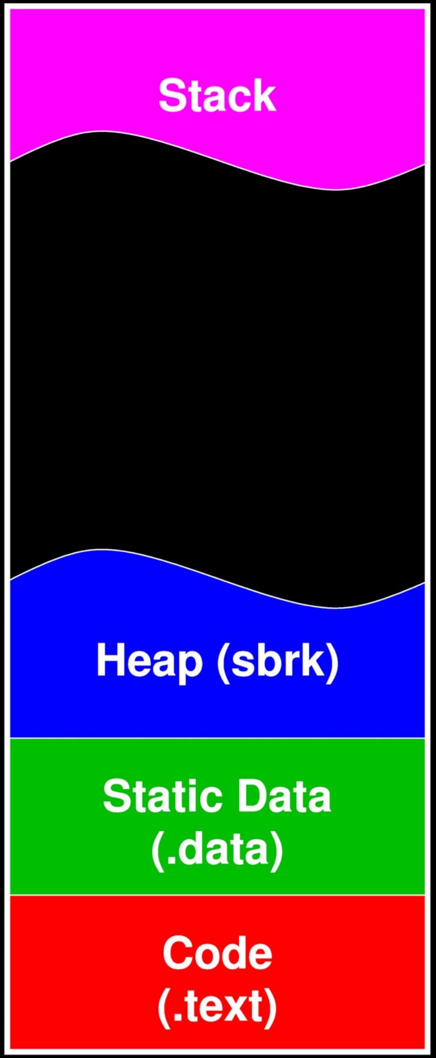
- How one maintains consistency
- What to do when something writes?
  - Invalidate: mark/discard old entries
  - Update: use the write data to update
- Who to notify?
  - Snooping: everybody watches
  - Ownership: only talk to owner
  - Directory: permissions, who to notify
- MESI Protocol: Modified (dirty), Exclusive, Shared (clean), Invalid – 4 line states

Verilog Implementation?

- A cache is a memory with the usual address decode logic, but:
  - Address used is hash(memory_address)
  - Each Cache memory cell contains (tag, data, dirty, valid, …)/line
  - Tag match and replacement algorithm
  - Partial read/write of data field
- State machine sequences operations
- Can be pipelined (even out of order)

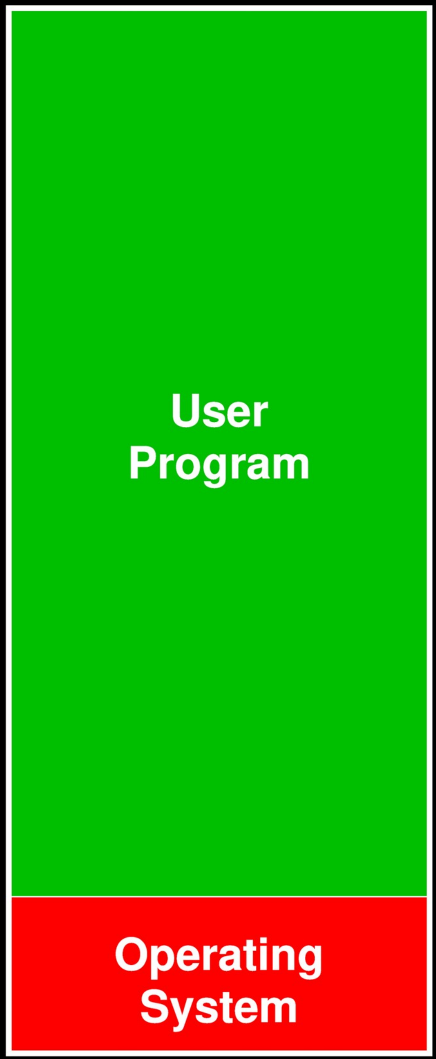
Memory Map of a Process

- Arranging stuff in memory:
  - Code starts at low address (0)
  - Static (fixed address) data
  - Heap typically grows up
  - Stack typically grows down
- Very bad if stack meets heap
  - Stack grows to cover SP
  - Heap grows by explicit calls to **`sbrk()`**, **`malloc()`**, **`new`**, etc.



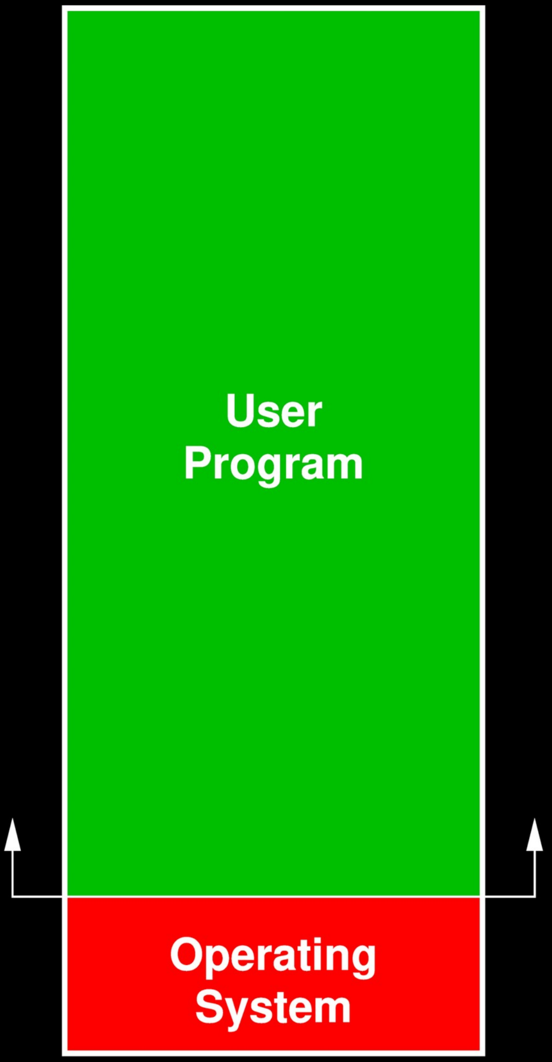
Memory Map of a Computer

- Originally, loaded one program at a time
  - OS was mostly a "loader"
  - User code could do anything
- Still a fairly common model for embedded computers and various microcontrollers



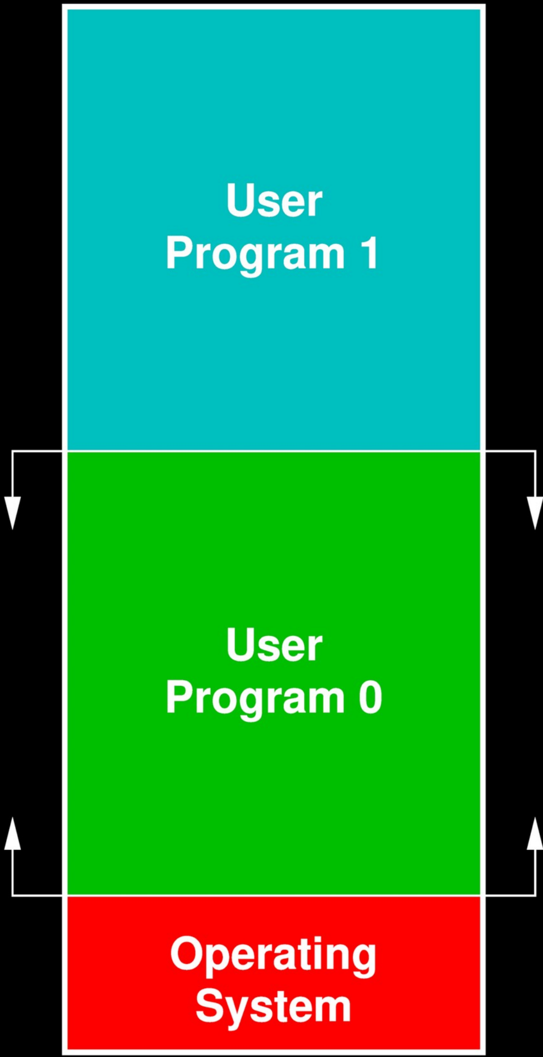
Protection

- A stray user program could corrupt the OS… add a fence register to protect it
- Processor respects fence unless in privileged mode
  - Become priv by system call or interrupt to trusted address
  - Surrender priv when return to user program

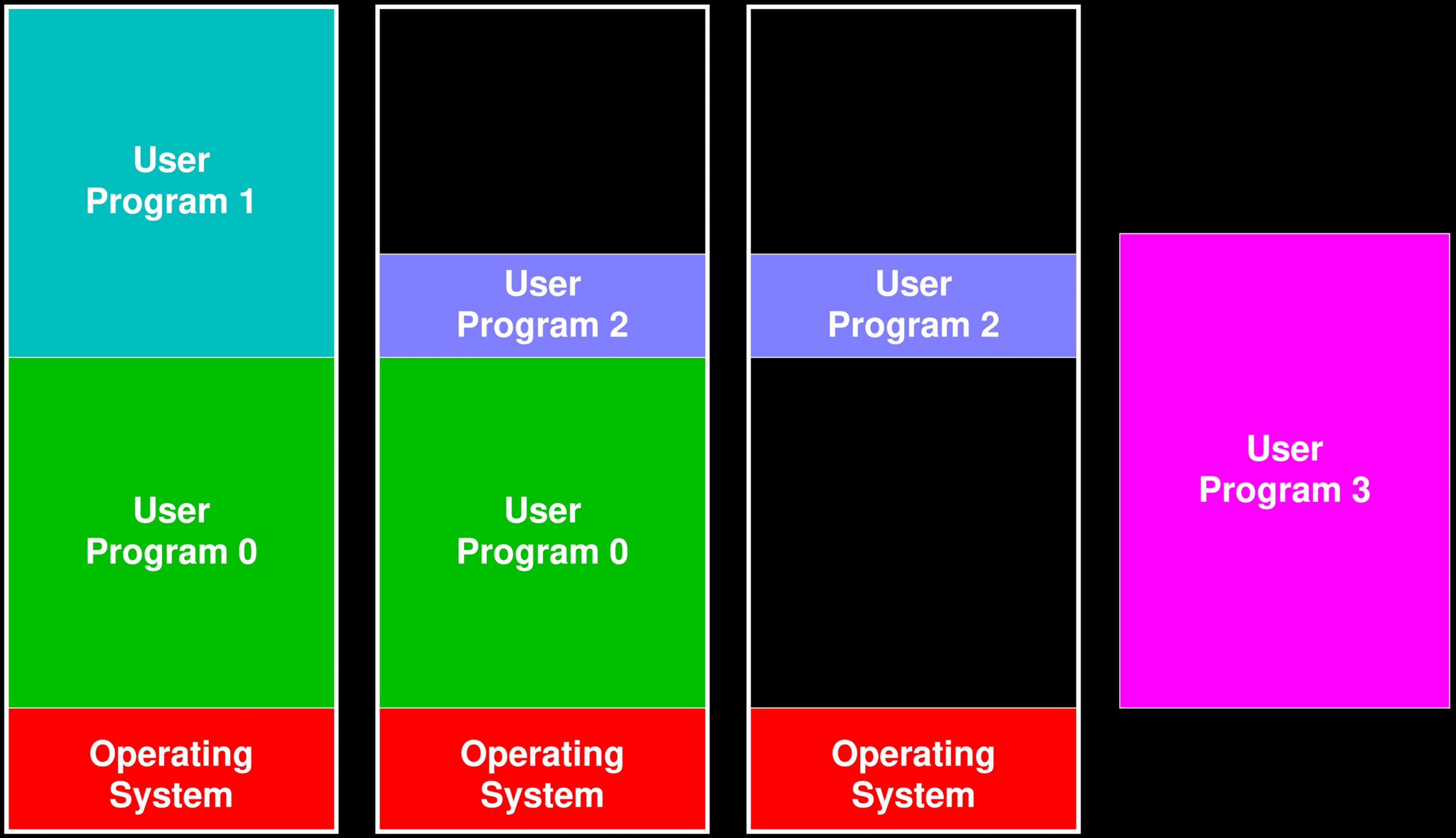


Batch Scheduling and Timesharing

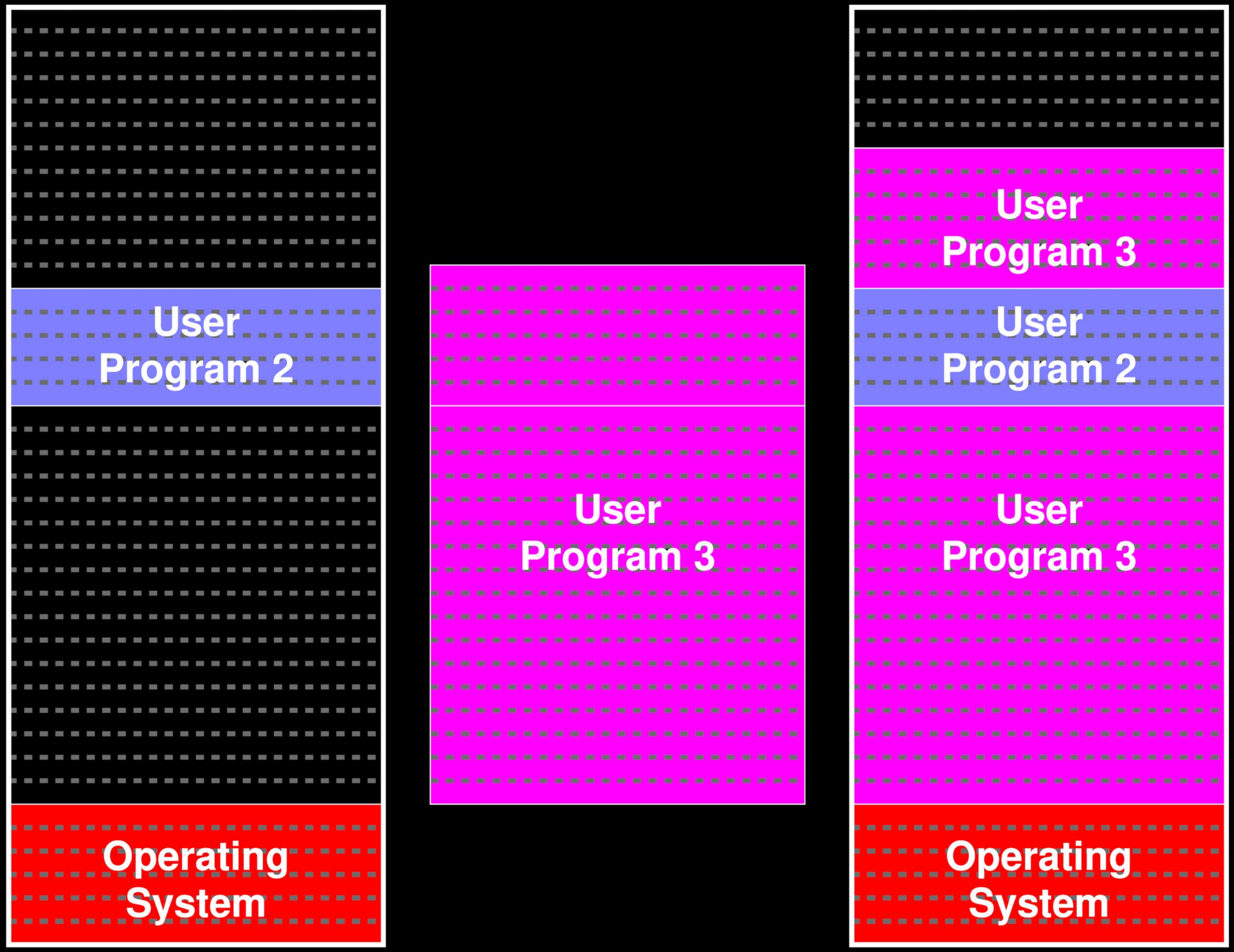
- Don't want expensive computer idle while waiting for printer, etc.
  - Load multiple jobs
  - Run 1 while 0 is waiting
- Timesharing: alternate running so all processes make progress
- Want two fence registers...



Memory Fragmentation



Memory Page Tables



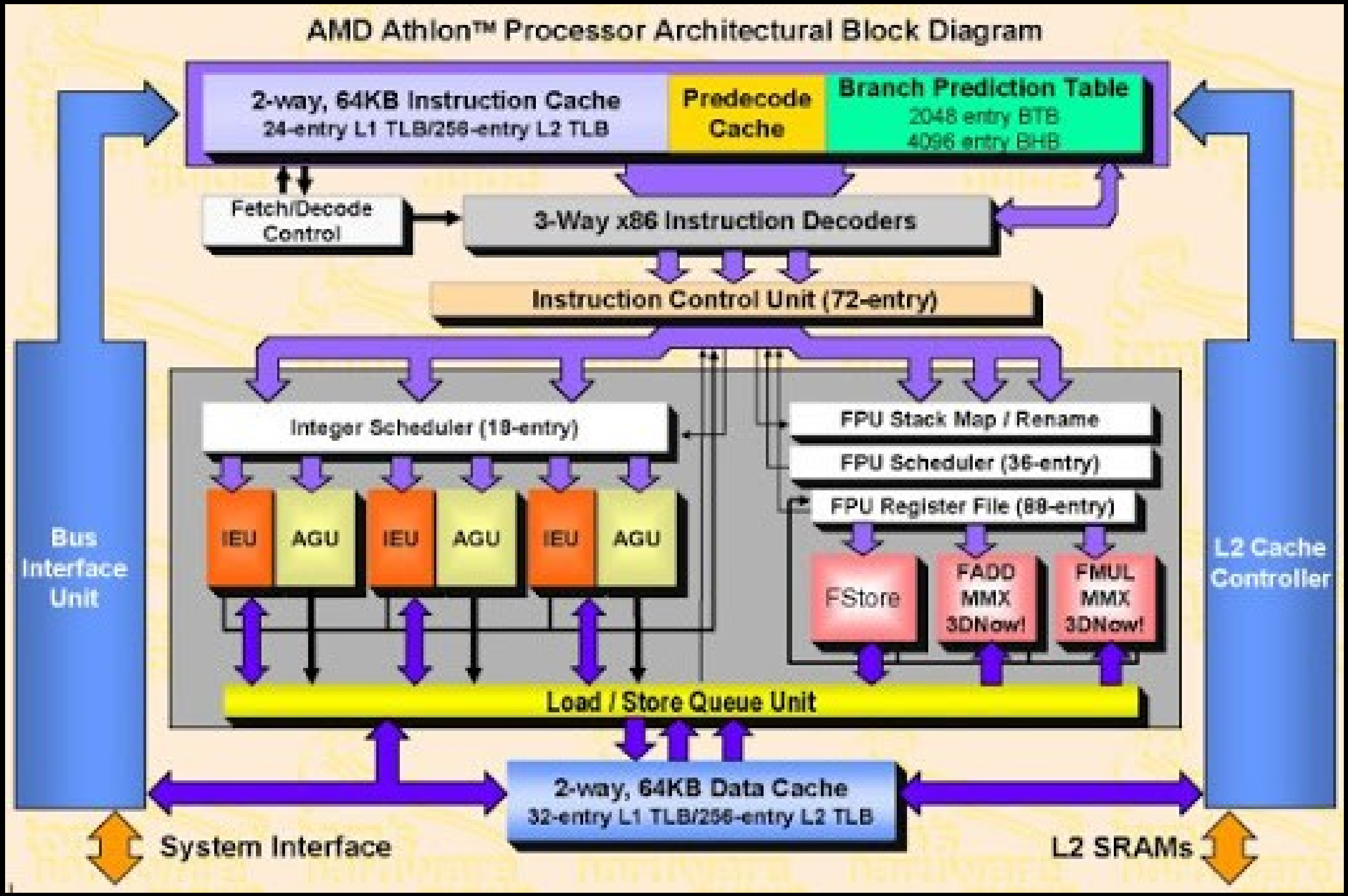
Logical vs. Physical Addresses

- Memory is divided into pages
  - Classically, each page is 4kB
  - Most systems also support 4MB pages
- Processor outputs logical (aka virtual) address
  - Top bits identify page number, bottom offset
  - Page table says where each page number is
  - Physical address substitutes page address in memory for logical page number

Page Table Issues

- 4kB pages are quite small...
  - IBM PC had 128KB memory, so 32 entries
  - With 4GB memory, need 1M page entries!
  - Each process needs a page table!
- Translation Lookaside Buffer (TLB)
  - Essentially a cache for page table entries
  - Translation typically before L1 cache… so the TLB needs to be fast, hence small
  - Can make L1/L2 TLBs, separate for I/D; don't wait for L1 miss to start search of L2

A Real Processor: AMD Athlon



4 TLBs: L1+L2 for each of code and data

Page Table Issues

- What happens for a TLB miss?
  - Instruction gets stopped, then restarted when the TLB has the appropriate entry…
    this requires hardware support
  - Must fetch page table entry (from memory)
- Thus, data in cache might not be accessible because TLB can't translate the address:

  e.g., L1 64kB cache has 1024 64B lines, but L1+L2 TLB might only have 256 entries!

Page Table Use

- Prevents memory fragmentation
- Allows per-page access protection (e.g., rwx)
- Don’t need to have everything in main memory!
  - Pages can not yet exist
  - Pages can be shared between processes
  - Pages can exist on disk
  - Pages can exist in a networked machine
- Pages can be slow to access from elsewhere

Page Table Benefits

- Pages can not yet exist
  - Stack, heap, and space between
- Pages can be shared between processes
  - DLLs: Dynamic Link Libraries
  - Inter-process communication
- Pages can exist on disk
  - Bigger than main memory
  - Fault in stuff as needed, mapped file I/O
- Pages can exist in a networked machine
  - DSM: Distributed Shared Memory

Verilog Implementation?

- A TLB is a memory with the usual address decode logic, but:
  - Address used is hash(memory_address)
  - Each TLB memory cell contains: (tag, physical_address, status)
  - Tag match and replacement algorithm
- Commonly pipelined such that L1 and L2 are sent a request at the same time and L2 aborts if L1 succeeds
- Hardware mechanism for handling misses