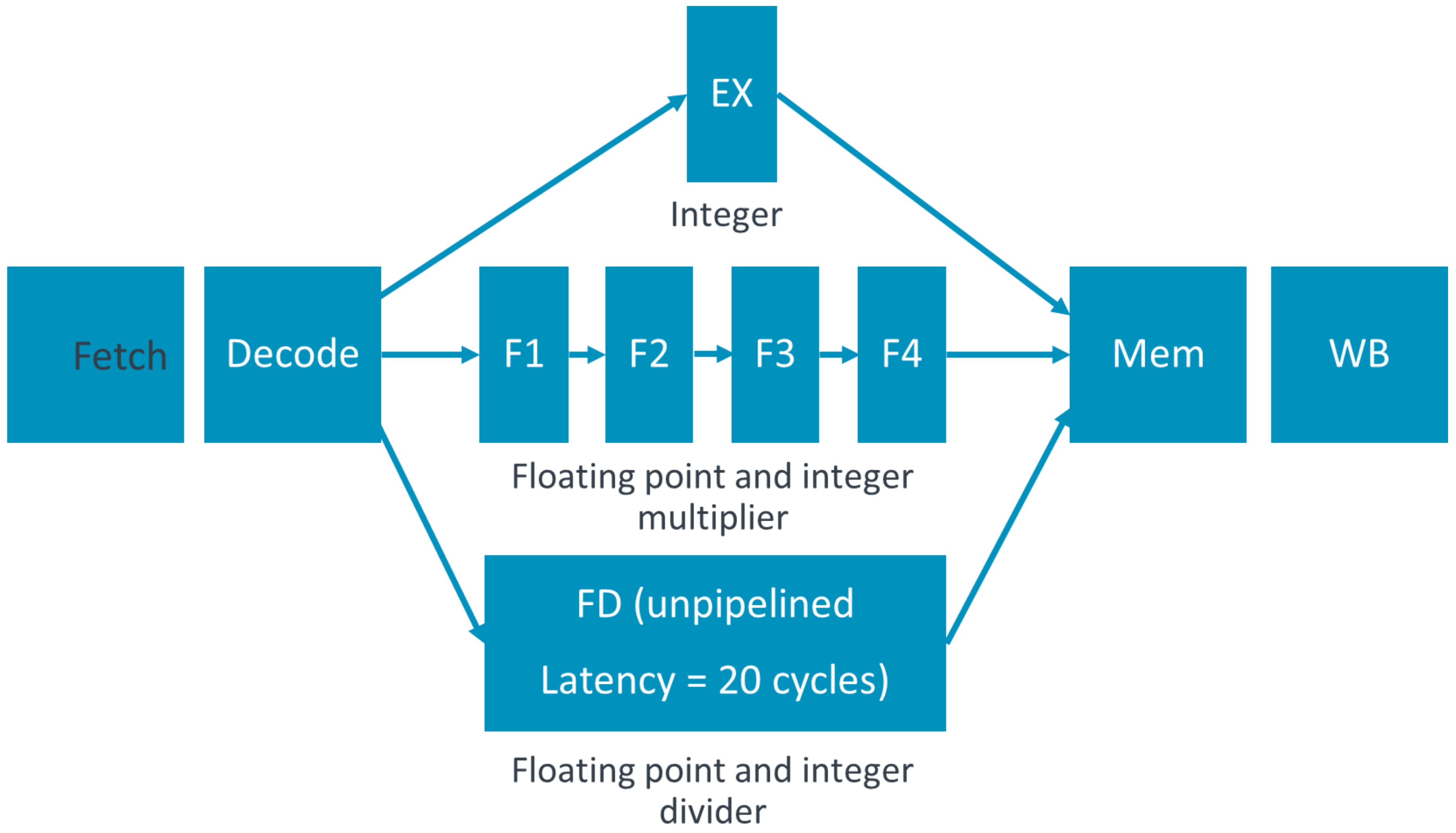
Course Review

Section II

Xinfei Guo
xinfei.guo@sjtu.edu.cn

July 27th, 2022

Diversified Pipelines


EX
Integer
Fetch
Decode
F1
F2
F3
F4
Mem
WB
Floating point and integer multiplier
FD (unpipelined
Latency = 20 cycles)
Floating point and integer divider

source: ARM

Store Queue + Load Queue

- Store Queue: handles forwarding, allows OoO stores
  - Entry per store (allocated @ dispatch, deallocated @ commit)
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read from SQ to write data cache (@ commit)
- Load Queue: detects ordering violations
  - Entry per load (allocated @ dispatch, deallocated @ commit)
  - Written by loads (@ execute)
  - Searched by stores (@ execute)
- Both together
  - Allows aggressive load scheduling
  - Stores don’t constrain load execution

The Memory (T5)

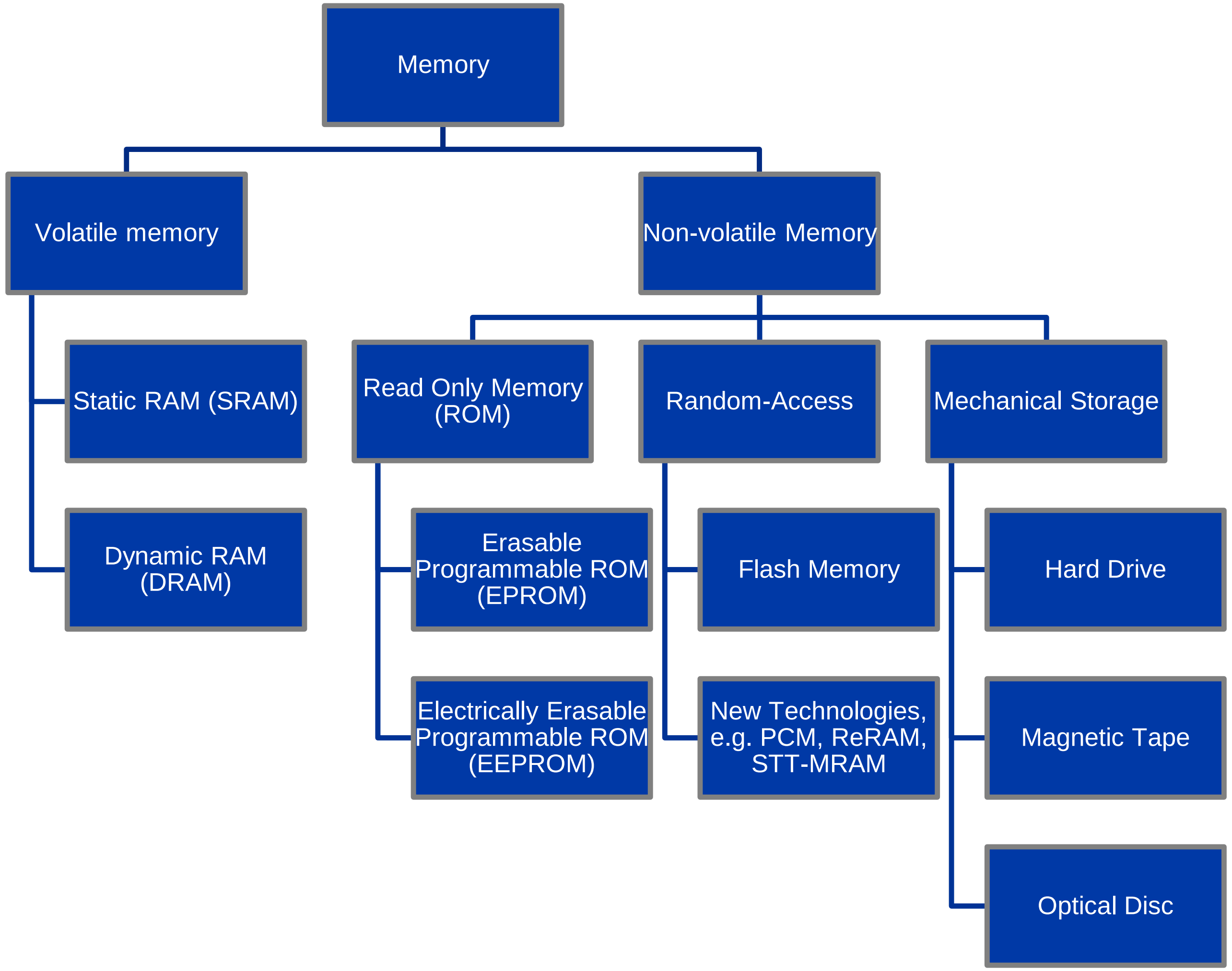
Types of Memory

Volatile memory

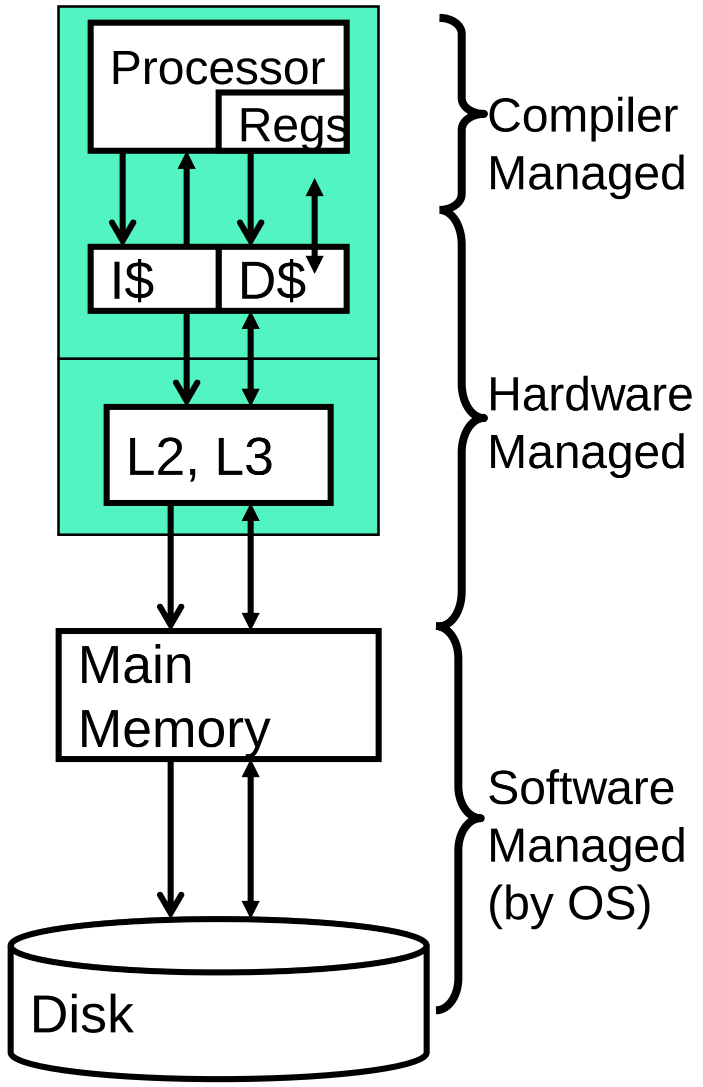
- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Non-volatile memory

- Read only memory (ROM)
  - Erasable programmable ROM (EPROM)
  - Electrically erasable programmable ROM (EEPROM)
- Non-volatile random-access memory (NVRAM)
  - Flash memory
- Mechanical storage
  - Hard drive, magnetic tape



Concrete Memory Hierarchy

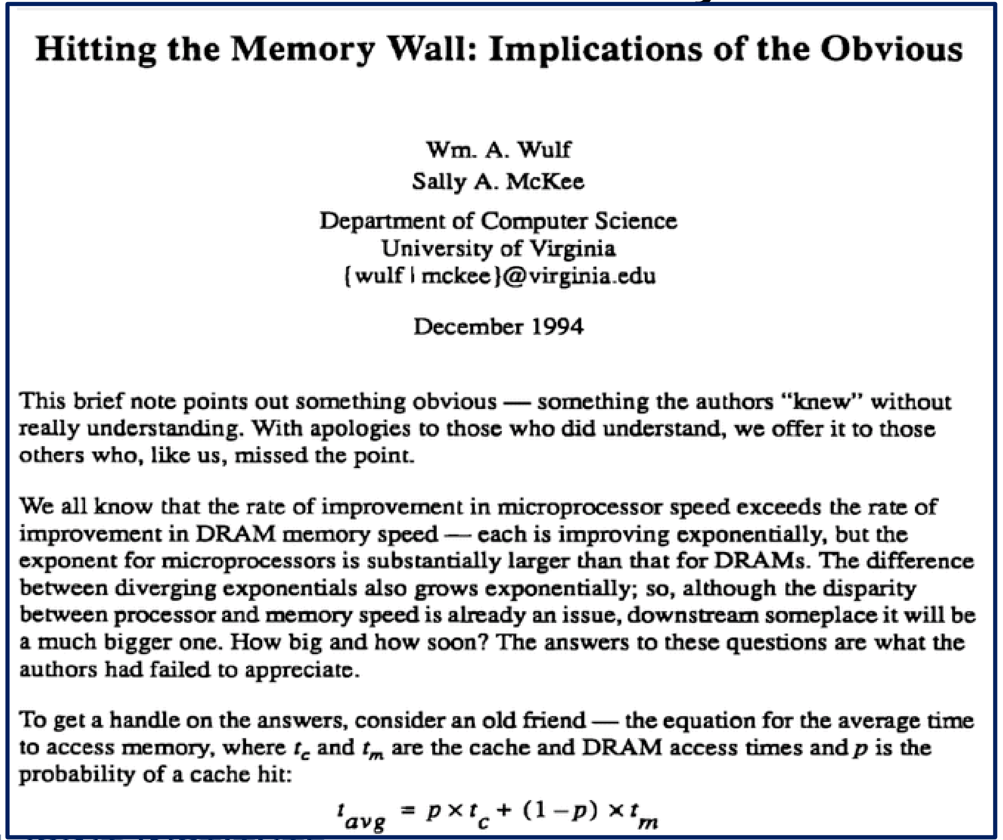


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

source: Prof. Joe Devietti @ U Penn

Memory Wall

- Originally theorized in 1994 by Wulf and McKee (U of Virginia)
- Processors are always waiting on memory, and CPU performance is therefore entirely limited by memory performance.



Hitting the Memory Wall: Implications of the Obvious

Wm. A. Wulf
Sally A. McKee
Department of Computer Science
University of Virginia
{wulf | mckee}@virginia.edu

December 1994

This brief note points out something obvious — something the authors "knew" without really understanding. With apologies to those who did understand, we offer it to those others who, like us, missed the point.

We all know that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed — each is improving exponentially, but the exponent for microprocessors is substantially larger than that for DRAMs. The difference between diverging exponentials also grows exponentially; so, although the disparity between processor and memory speed is already an issue, downstream someplace it will be a much bigger one. How big and how soon? The answers to these questions are what the authors had failed to appreciate.

To get a handle on the answers, consider an old friend — the equation for the average time to access memory, where t_c and t_m are the cache and DRAM access times and p is the probability of a cache hit:

$$t_{avg} = p \times t_c + (1 - p) \times t_m$$

"We all know that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed – each is improving exponentially, but the exponent for microprocessors is substantially larger than that for DRAMs.

The difference between diverging exponentials also grows exponentially; so, although the disparity between processor and memory speed is already an issue, downstream someplace it will be a much bigger one."

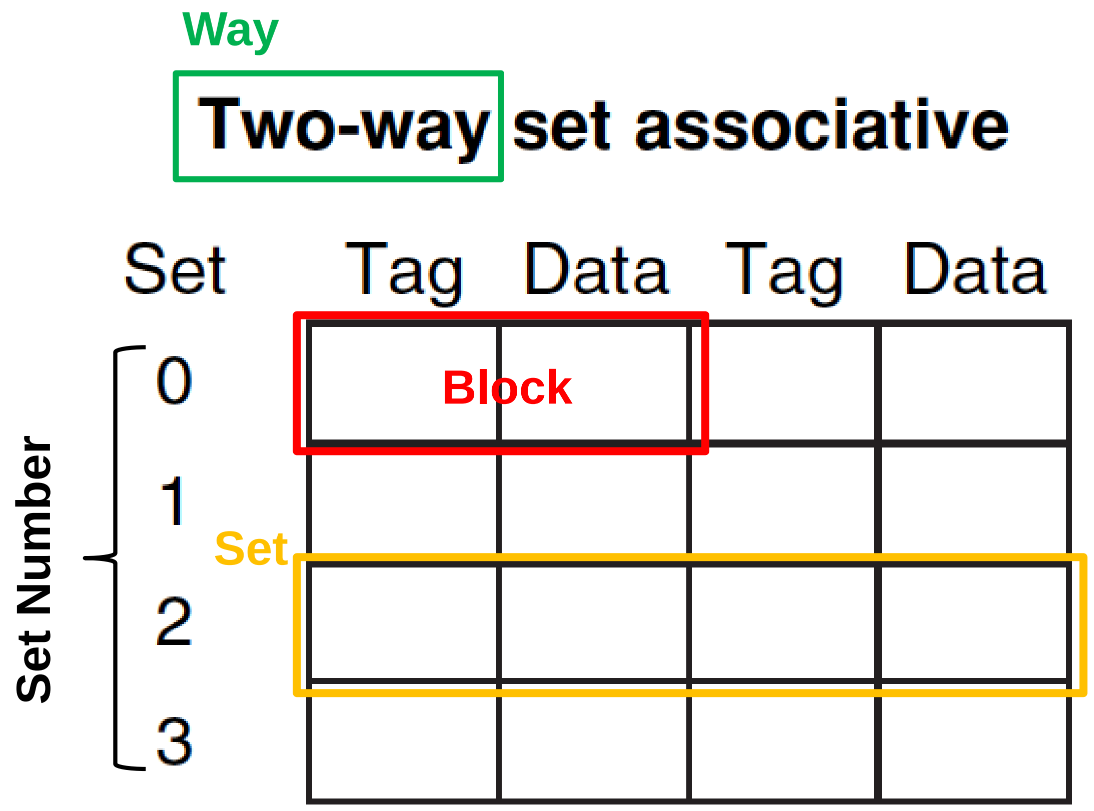
https://dl.acm.org/doi/10.1145/216585.216588

Cache motivation

- Fortunately, most programs don’t need access to all memory all of the time.
  - Accesses tend to exhibit locality of reference.
  - Temporal locality – if an address is accessed, it is likely to be accessed again soon.
  - Spatial locality – if an address is accessed, its neighbors are likely to be accessed soon.
  - Therefore, only a small number of addresses are likely to be accessed in the near future.
- Small memories are quick to access and can be placed near to the CPU.
  - If we can identify these locations likely to be accessed soon, then we can keep them in these memories.
- A cache stores copies of some memory locations for fast access when required.

Terminology

- Block
- Set
- Way
- # of Set



Each block can have multiple words.

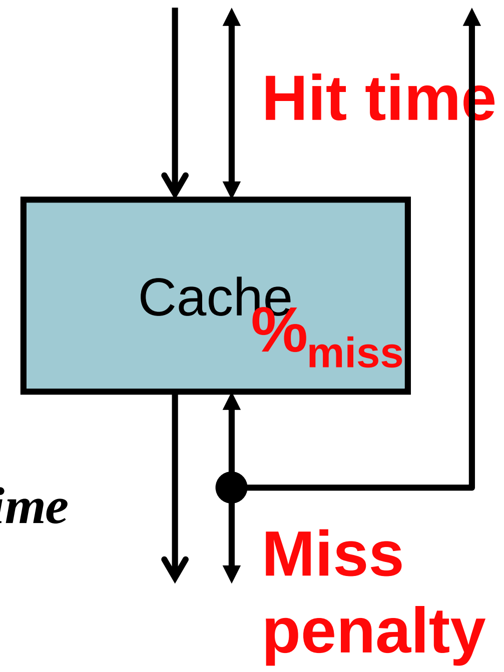
Cache performance

- $\text{CPI}_{\text{ALUOps}}$ does not include memory instructions
- AMAT = Average Memory Access Time

$$AMAT = HitTime + MissRate \times MissPenalty$$

$$= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}\right) + \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}\right)$$

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT\right) \times CycleTime$$



- For example:
  - An L1 cache with 1 ns hit latency and 5% miss rate
  - Combined with an L2 cache with a 10 ns hit latency and 1% miss rate
  - And 100 ns main memory latency

AMAT = 1 + 0.05 * (10 + 0.01 * 100) = 6.5 ns

Review: "Three Cs" Model of Misses

- 1st C - Compulsory (cold)
  - occur when a program is first started
- 2nd C – Conflict
  - miss that occurs because two distinct memory addresses map to the same cache location
- 3rd C – Capacity
  - cache cannot contain all blocks needed for program execution

Effect of Cache Parameters on Performance

Bigger caches

- Pros:
  - Reduces capacity misses
- Cons:
  - Increases hit time
  - More expensive
  - Consumes more power

Larger cache blocks

- Pros:
  - Better spatial locality
  - Reduces number of tags
- Cons:
  - Increases miss penalty
  - Increases capacity misses
  - Increases conflict misses

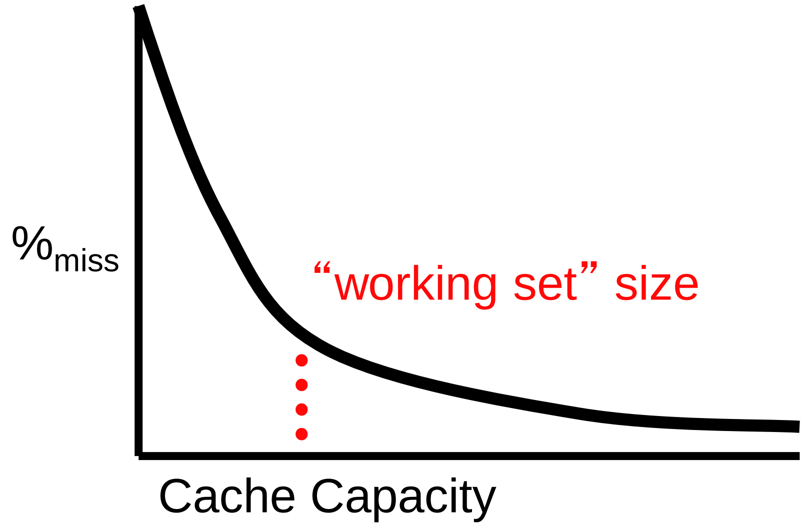
Higher associativity

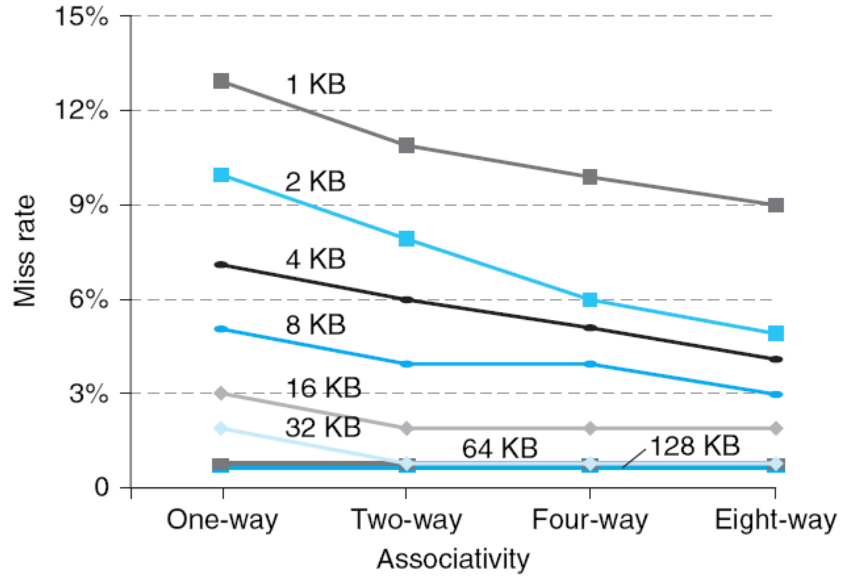
- Pros:
  - Reduces conflict misses
- Cons:
  - Increases hit time
  - Consumes more power

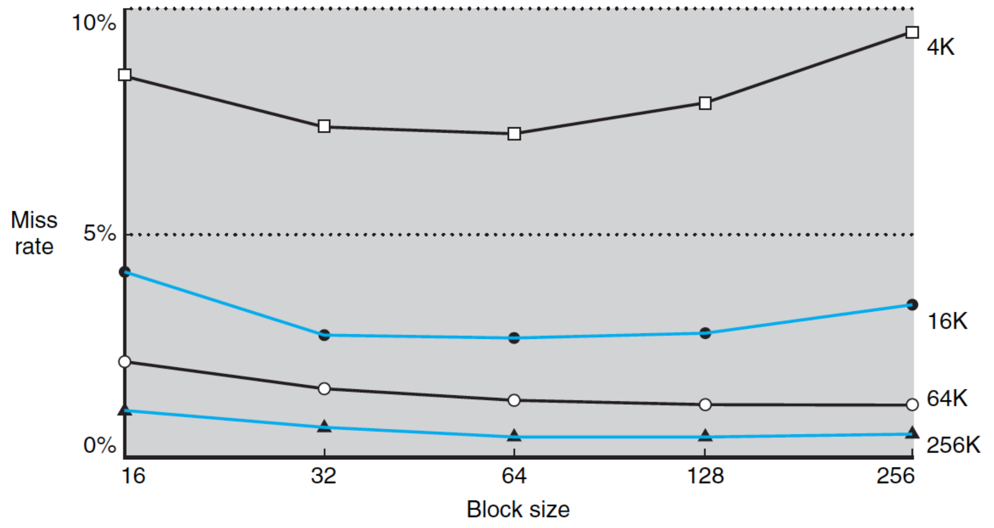
Miss Rate ABC: **A**ssociativity, **B**lock size, **C**apacity

Important Tradeoffs

- Capacity and Performance
- Block Size and Performance
- Associativity and Performance







Compiler Optimizations to Reduce Miss Rate

- McFarling [1989] reduced misses by 75% in software on 8KB direct-mapped cache, 4 byte blocks
- Instructions
  - Reorder procedures in memory to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Loop interchange: Change nesting of loops to access data in memory order
  - Blocking: Improve temporal locality by accessing blocks of data repeatedly vs. going down whole columns or rows
  - Merging arrays: Improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop fusion: Combine 2 independent loops that have same looping and some variable overlap

Write Propagation

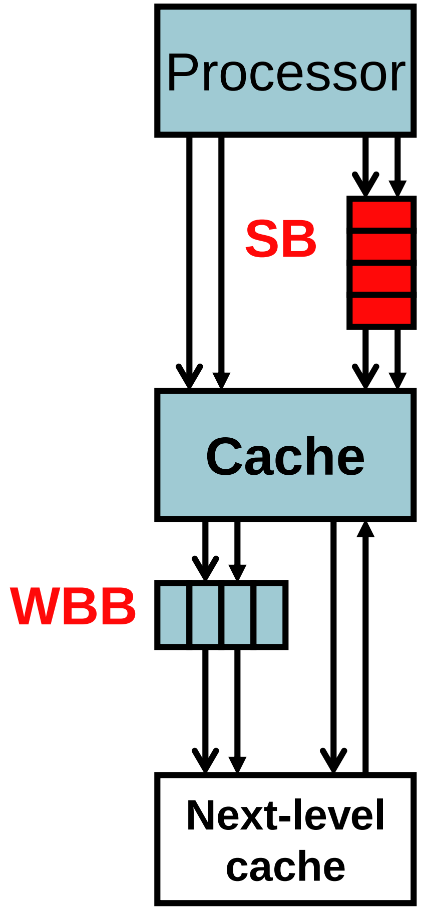
- When to propagate new value to lower-level caches/memory?
- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
  - memory (or other processors) always have latest data
  - Simpler management of cache
- **Option #2: Write-back**: when block is replaced
  - Now we have multiple versions of the same block in various caches and in memory!
  - Requires additional "**dirty**" bit per block (updated with new values)
  - much lower bandwidth, since data often overwritten multiple times
  - Better tolerance to long-latency memory?

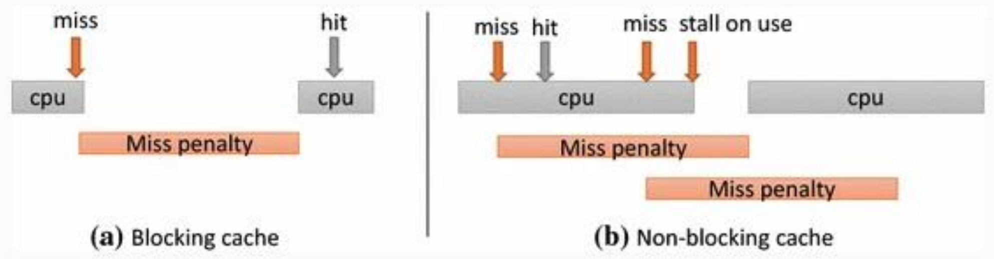
Write Allocate vs Non-Allocate

- What happens on write-miss?
- Write allocate: allocate new cache line in cache
  - Allocate cache block on miss by fetching corresponding memory block
  - Update cache block and then memory block
  - Commonly used (especially with write-back caches hoping subsequent writes will be captured in cache)
- Write non-allocate (or "write-around")
  - Simply send write data through to underlying memory/cache - don't allocate new cache line!
  - potential more read miss
  - Commonly used in write through (write must still go to lower level memory)

Optimization Techniques

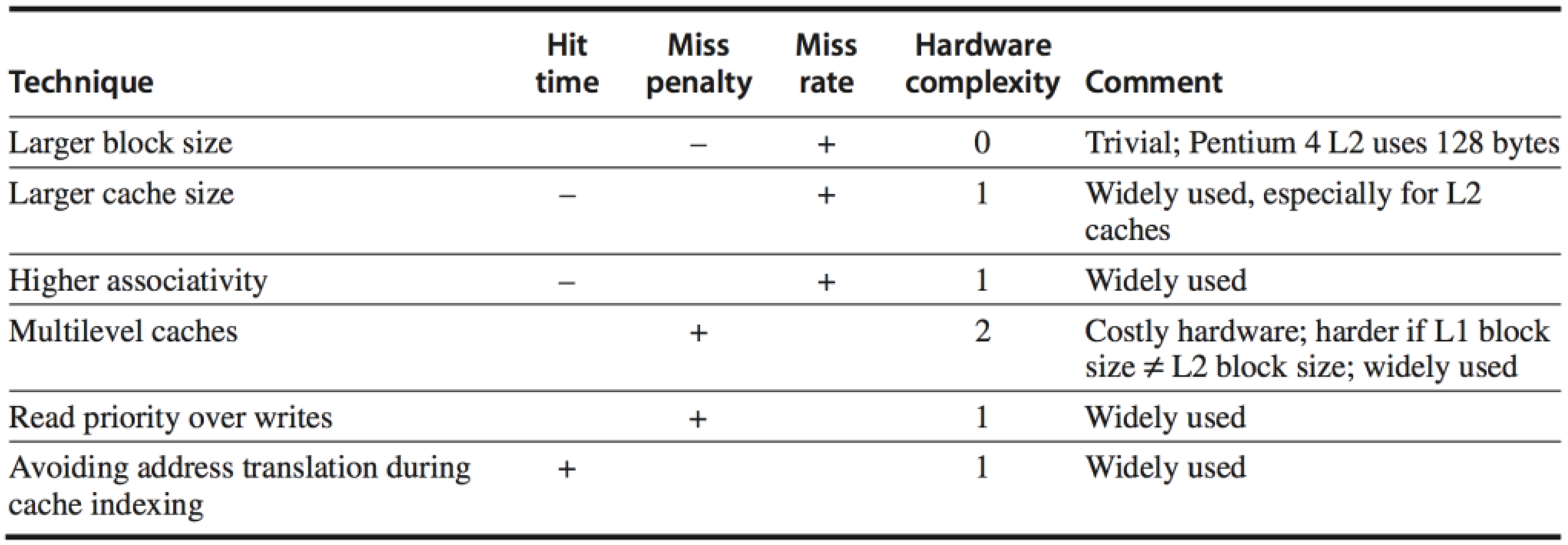
- Write Buffer for Write-Through
- Store buffer vs. writeback-buffer
- Non-blocking Cache
- Way Prediction





Blocking versus non-blocking cache

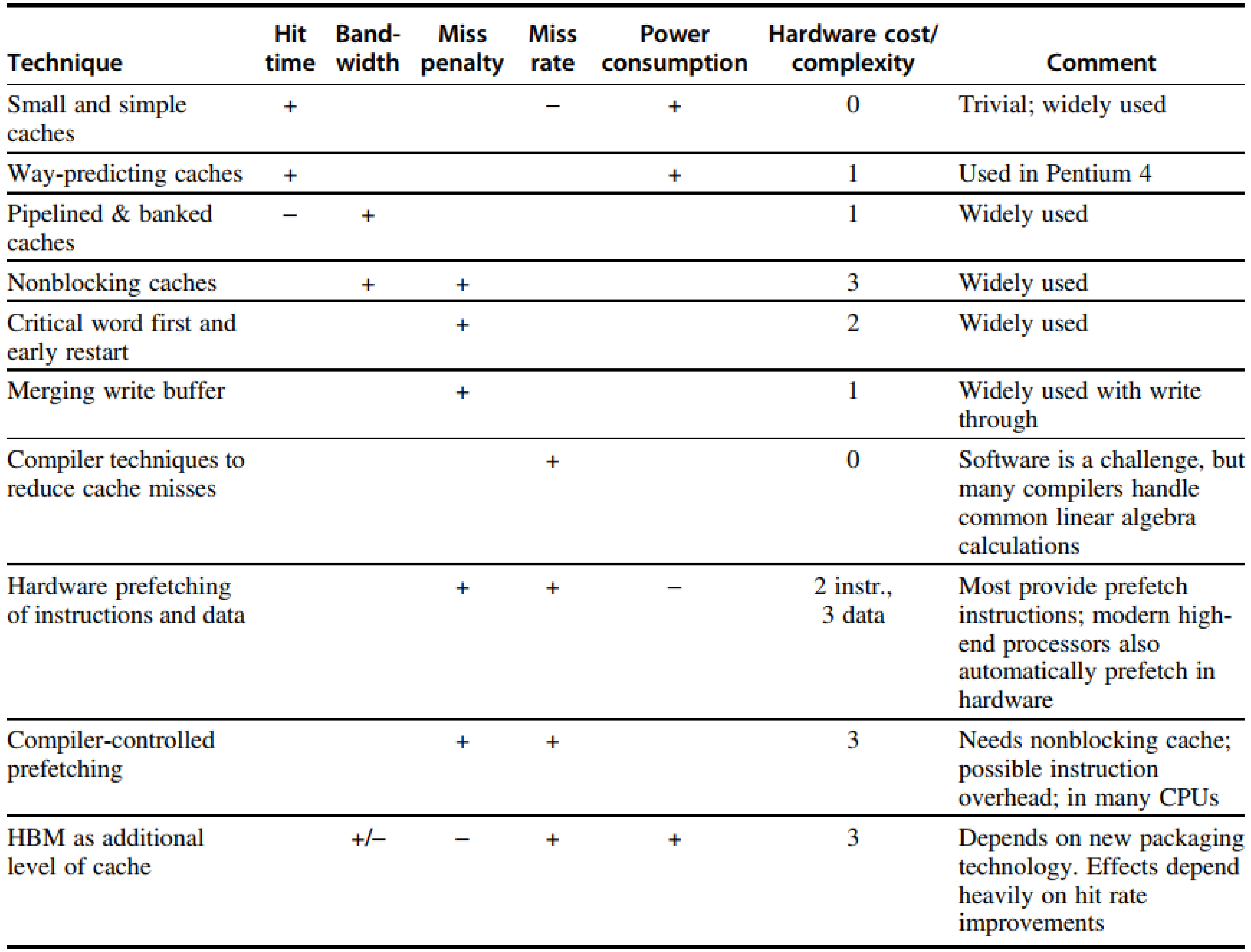
Summary (Basic Cache Optimization)



Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + 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.

Summary (Advanced Cache Optimization)

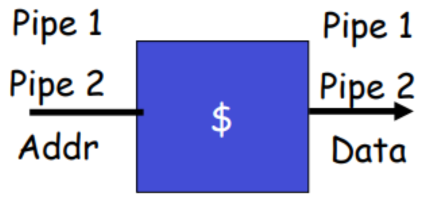


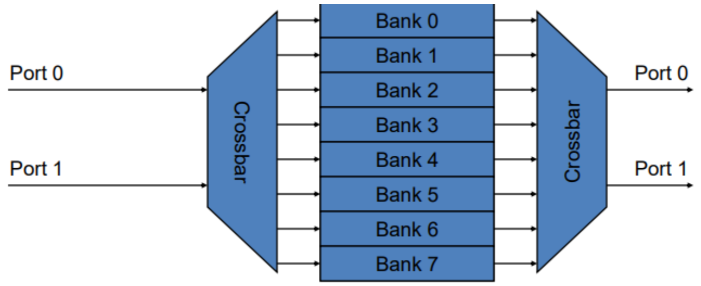
Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical 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
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

JOINT INSTITUT
交大密西根学院

Superscalar Memories

- Increasing issue width => wider caches
- Parallel cache access is harder than parallel FUs
  - fundamental difference: caches have state, FUs don't
  - one port affects future for other ports
- Several approaches used
  - true multi-porting
  - muliple cache copies
  - virtual multi-porting
  - Non-Uniform Cache Architecture (NUCA)

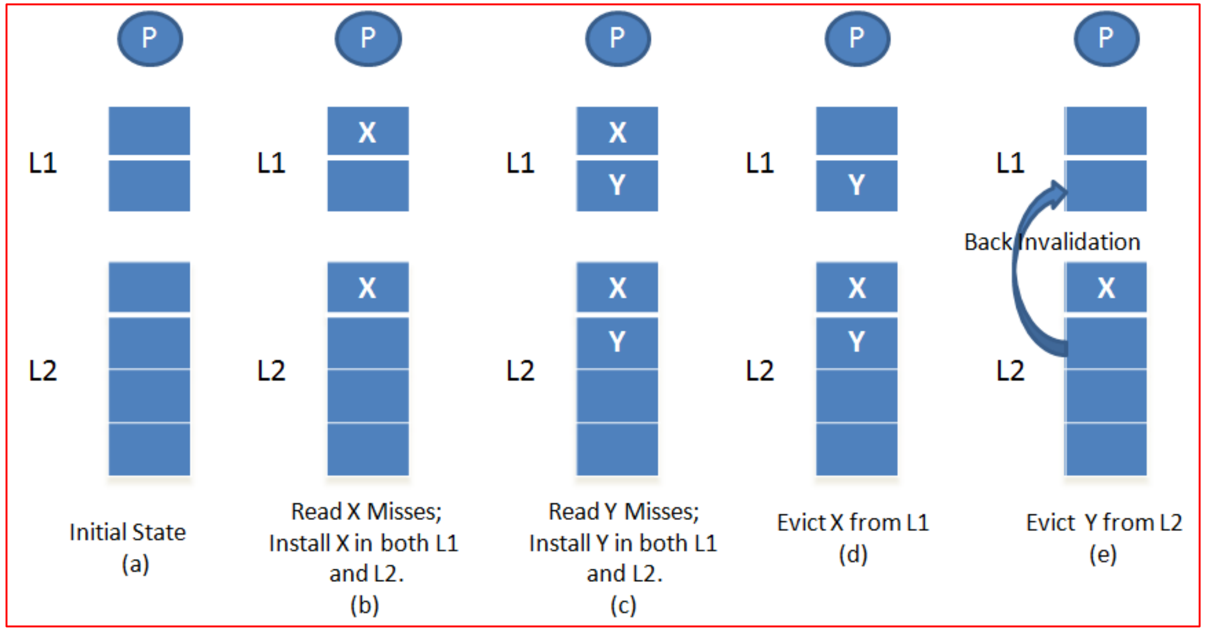




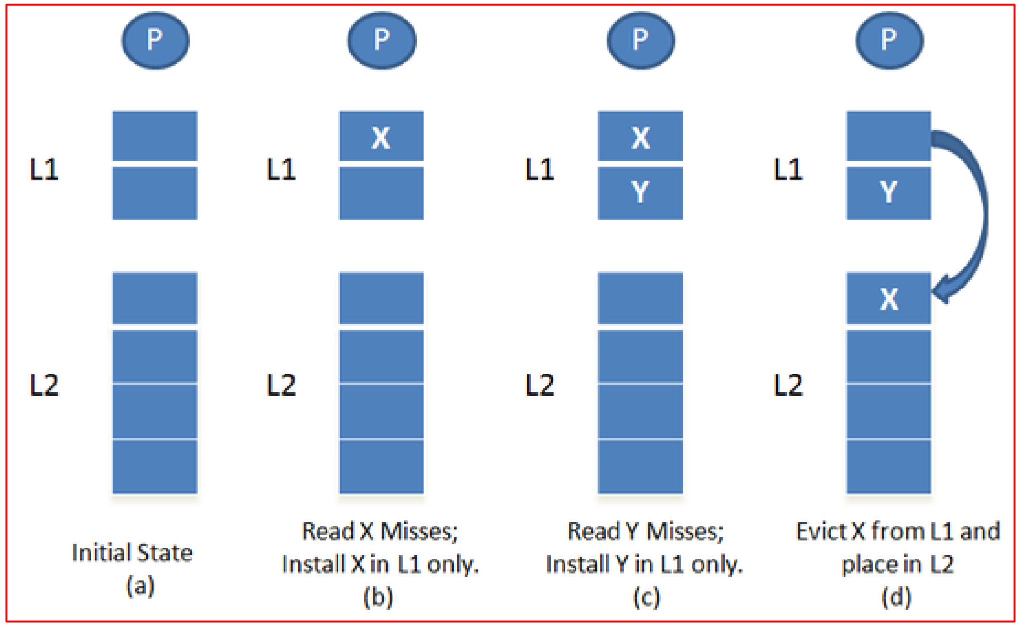
Latency vs. Bandwidth

- Bandwidth can be handled by “spending” more (hardware cost)
  - Wider buses, interfaces
  - Banking/interleaving, multiporting
- Ignoring cost, a well-designed system should never be bandwidth-limited
  - Can’t ignore cost!
- Bandwidth improvement usually increases latency
  - No free lunch
- Hierarchies decrease bandwidth demand to lower levels
  - Serve as traffic filters: a hit in L1 is filtered from L2
- Parallelism puts more demand on bandwidth
- If average b/w demand is not met => infinite queues
  - Bursts are smoothed by queues
- If burst is much larger than average => long queue
  - Eventually increases delay to unacceptable levels

Inclusion versus Exclusion



Inclusive Policy



Exclusive Policy

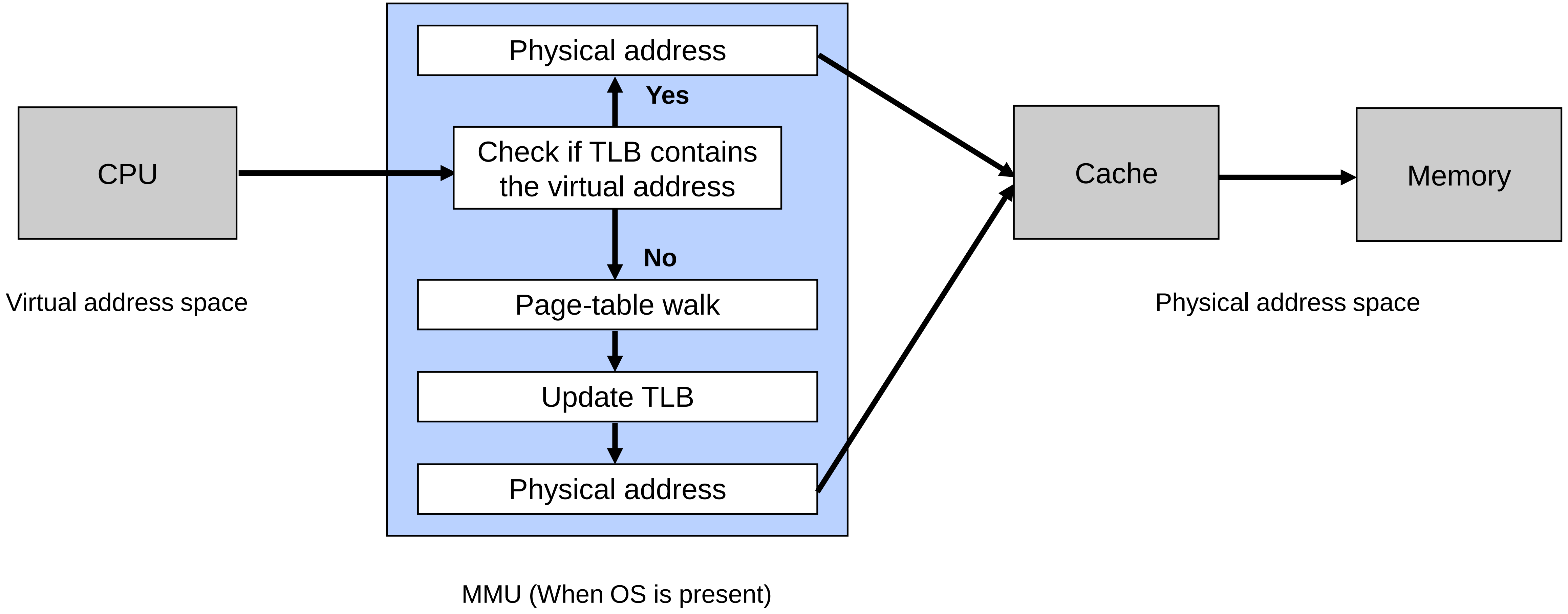
source: wiki

Split vs. Unified Caches

- **Split I$/D$**: insns and data in different caches
  - To minimize structural hazards and t_{access}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ and D$ separately
    - Not writes for I$, smaller reads for D$
  - Why is 486 I/D$ unified?

- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$
  - + Fewer capacity misses: unused insn capacity can be used for data
  - – More conflict misses: insn/data conflicts
    - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous I$/D$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core

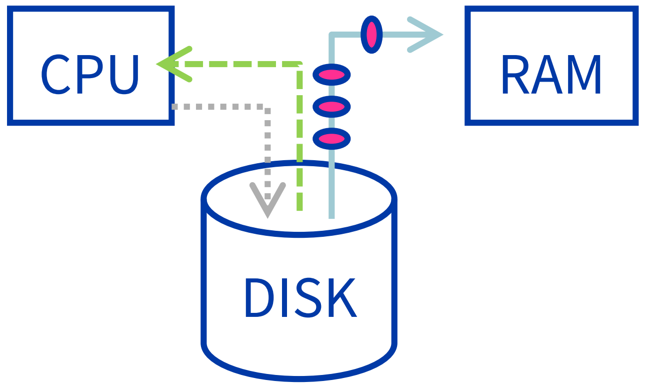
Overview of Memory Access Using an MMU



Alternative: DMA

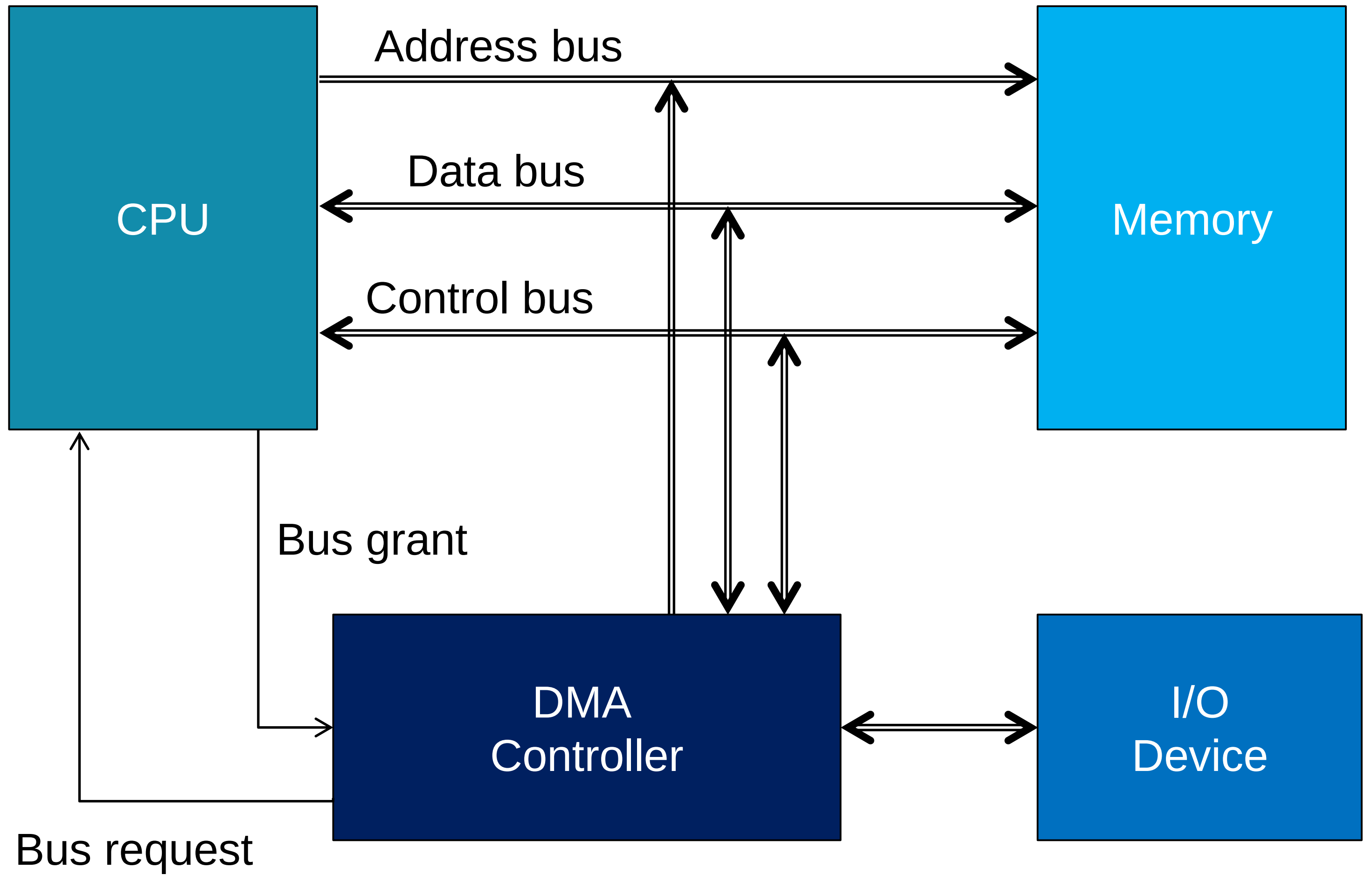
- Direct Memory Access (DMA)
- Until now CPU has sole control of main memory…
- But DMA
  - Allows I/O devices to directly read/write main memory
  - CPU sets up DMA request
  - for (i = 1 ... n)

    Device puts data on bus
    & RAM accepts it
  - Device interrupts CPU after done



Source: cs.cornell.edu/courses/cs3410

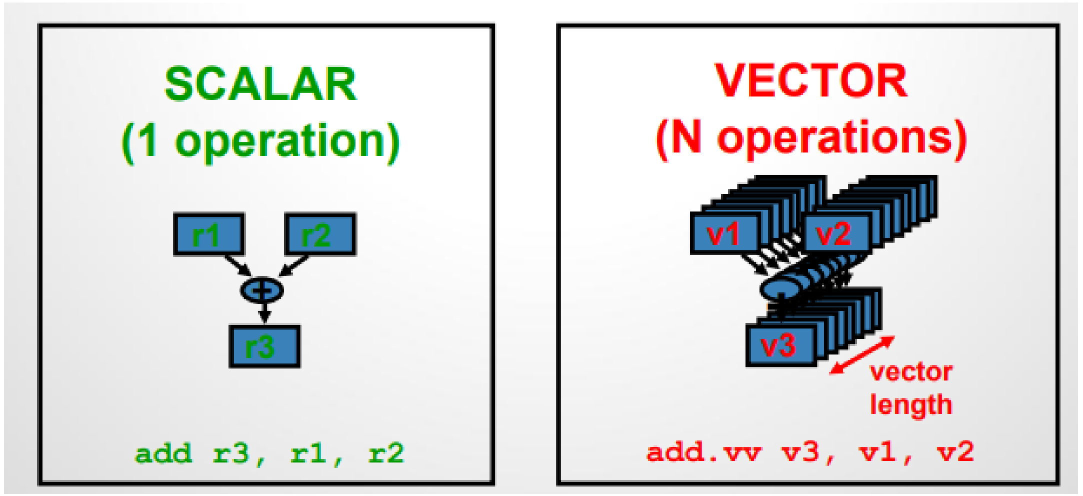
DMA Architecture



SIMD (T6)

SIMD Important Topics

- Flynn’s Classification
- Concept of SIMD
- Vector Processor
  - Components of Vector Processors
  - Vector Chaining
  - Scatter and Gather
  - Strip Mining
  - Masked Operations
  - SIMD Extensions
- GPU



GPU Architectures

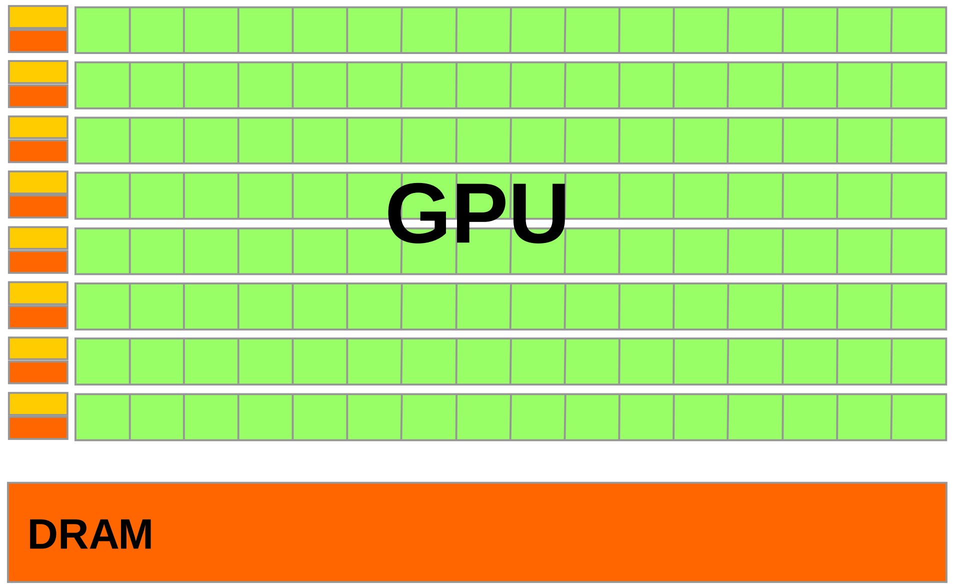
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

GPU Design Principles

- CPU design is about making a single thread run as fast as possible.
  - Pipeline stalls and memory accesses are expensive in terms of latency.
  - So increased logic was added to reduce the probability/cost of stalls.
  - Use of large cache memories to avoid memory misses
- GPU design is about maximizing computation throughput.
  - Individual thread latency not considered important
  - GPUs avoid much of the complex CPU pipeline logic for extracting ILP.
  - Instead, each thread executes on a relatively simple core with performance obtained through parallelism.
    - Single instruction, multiple threads
- Computation hides memory and pipeline latencies.
- Wide and fast bandwidth-optimized memory systems

GPUs: Throughput Oriented Design

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- Before we understand that, let's distinguish between
  - Programming Model (Software)

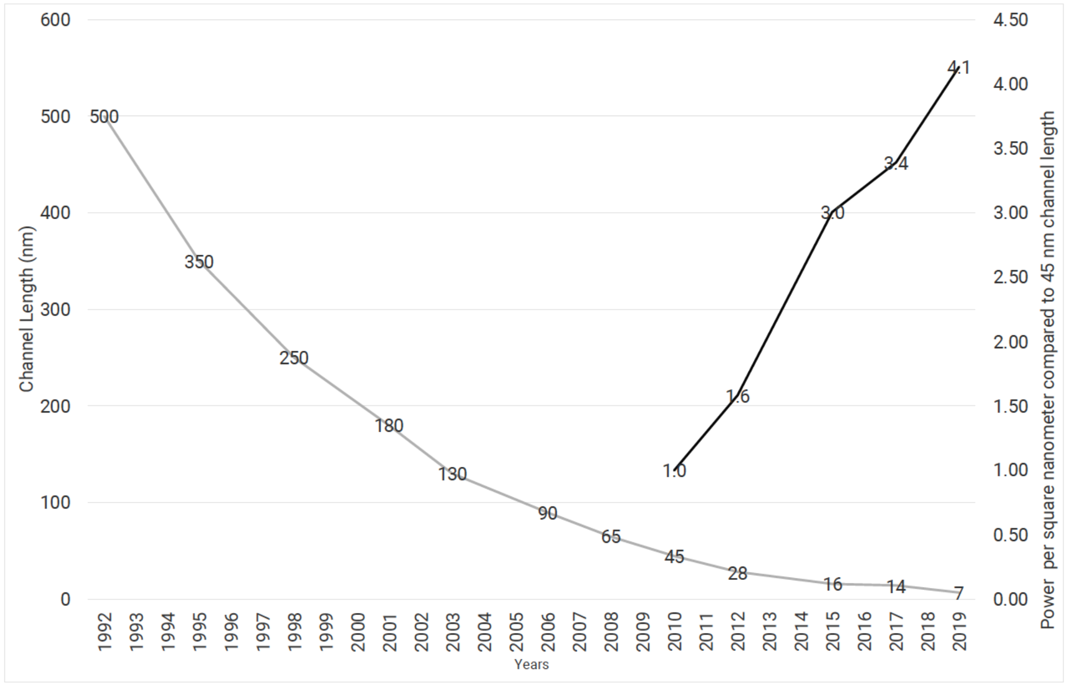
    vs.
  - Execution Model (Hardware)

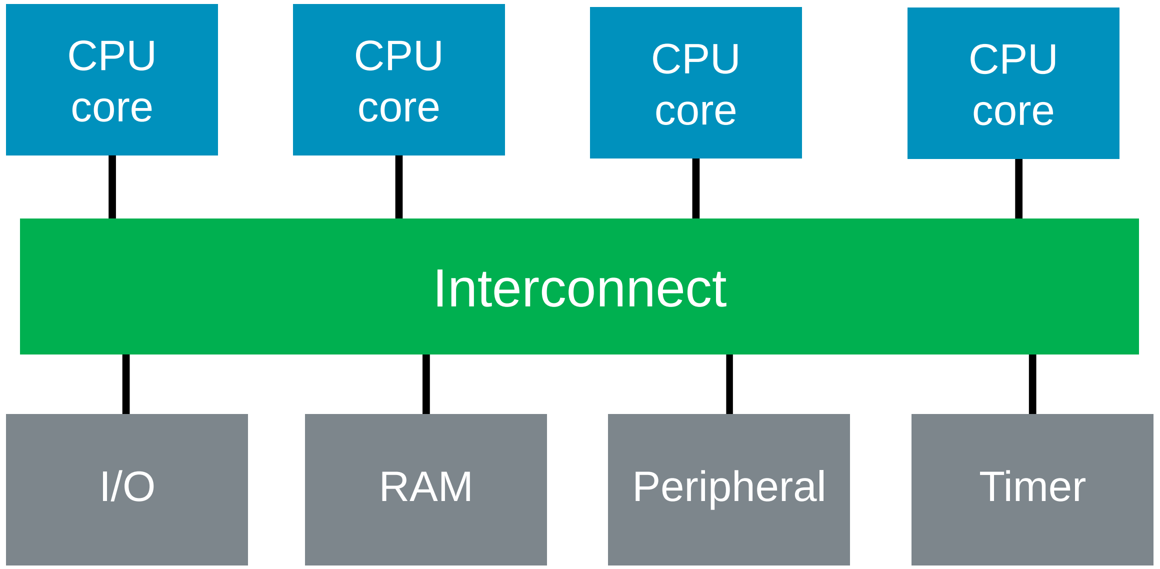
Slide Credit: Prof. Onur Mutlu, ETH

Multicore (T7)

Multicore

- Why, how and what?
  - Dennard Scaling
  - Multicore vs. SMT
- Distributed vs shared caches
- Cache Coherence

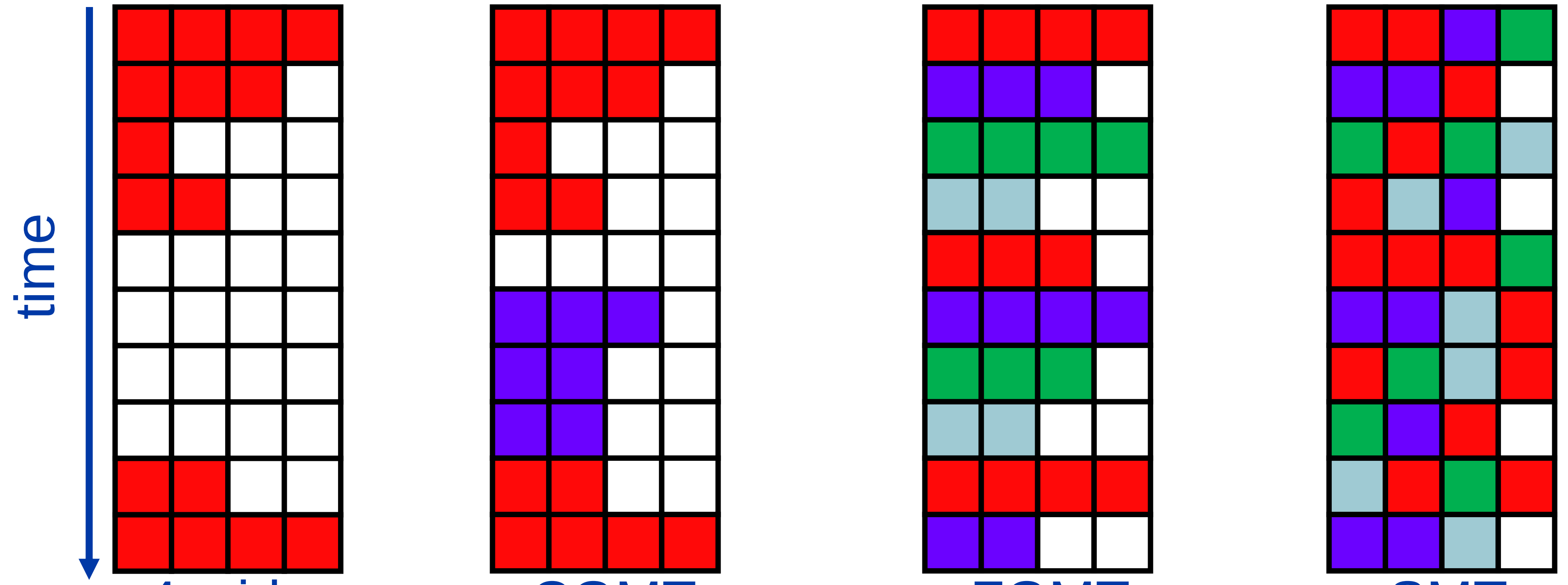




Standard Multithreading Picture

Time evolution of issue slots

- Color = thread, white = no instruction (bubble)



4-wide Superscalar

CGMT

Switch to thread B on thread A L2 miss

FGMT

Switch threads every cycle

SMT

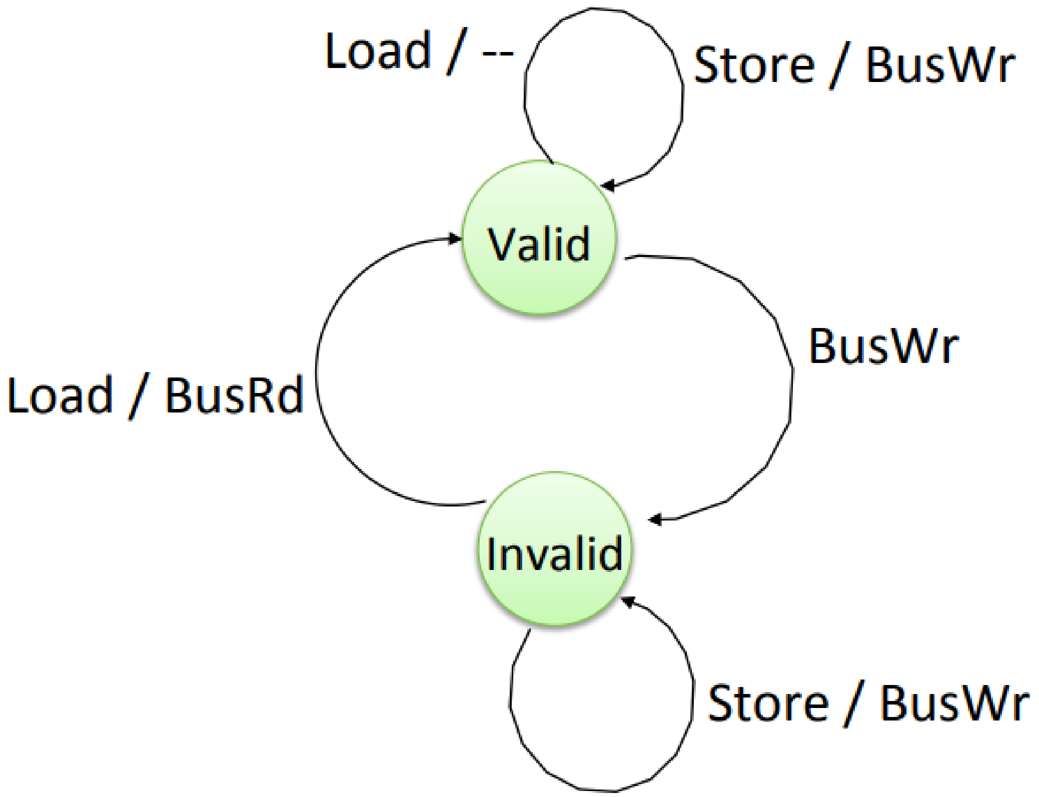
Insns from multiple threads coexist

CG: coarse grain
FG: fine grain

Approaches to Cache Coherence

- Software-based solutions
  - Mechanisms
    - Mark cache blocks/memory pages as cacheable/non-cacheable
    - Add “Flush” and “Invalidate” instructions
  - Could be done by compiler or runtime system
  - Difficult to get perfect
- Hardware solutions are far more common
  - Simple schemes rely on broadcast over a bus (Snooping Protocol)

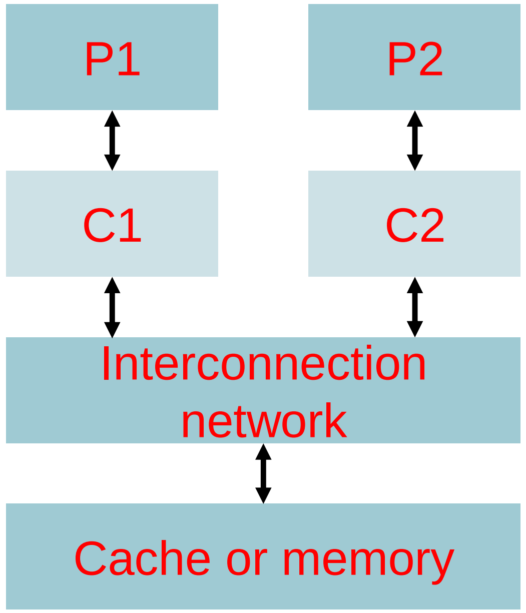
Valid-Invalid (VI) Snooping Protocol



Actions:
Ld, St, BusRd, BusWr

Write-through, no-write-allocate cache

1 bit of storage overhead per cache frame



- If *you* load/store a block: transition to **V**
- If anyone *else* wants to read/write block:
  - Give it up: transition to **I** state
  - Write-back if your own copy is dirty

"Valid/Invalid" Cache Coherence

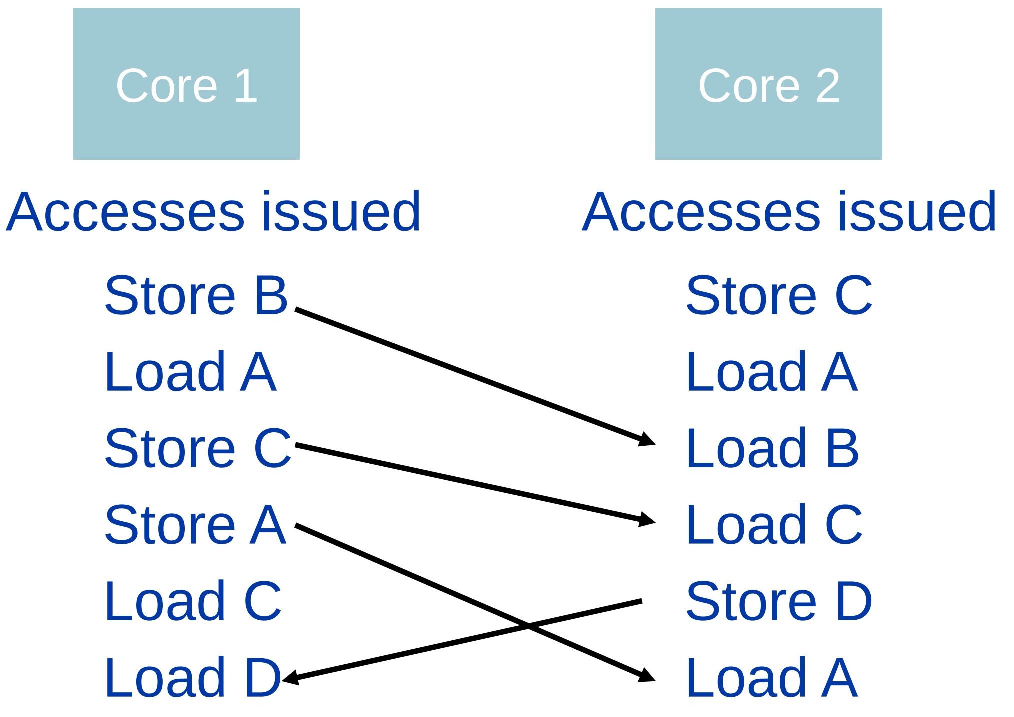
- To enforce the shared memory invariant…
  - "Loads read the value written by the most recent store"
- Enforce the invariant…
  - **"At most one valid copy of the block"**
  - Simplest form is a **two-state "valid/invalid" protocol**
  - If a core wants a copy, must find and "invalidate" it
- On a cache miss, how is the valid copy found?
  - Option #1 "**Snooping**": broadcast to all, whoever has it responds
  - Option #2: "**Directory**": track sharers with separate structure
- **Problem**: multiple copies can't exist, even if read-only
  - Consider mostly-read data structures, instructions, etc.

Modified-Shared-Invalid (MSI) Protocol

- Allows for many readers…
- Three states tracked per-block at each cache
  - **M**odified – cache has the only copy; writable; dirty (read/write permission)
    - Dirty == memory is out of date
  - **S**hared – cache has a read-only copy; clean (read-only permission)
    - Clean == memory is up to date
  - **I**nvalid – cache does not have a copy (blocked, no permission)
- Three processor actions
  - Load, Store, Evict
- Five bus messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
  - Could combine some of these

Memory Consistency vs Cache Coherence

Cache coherence



Data propagation

Memory consistency

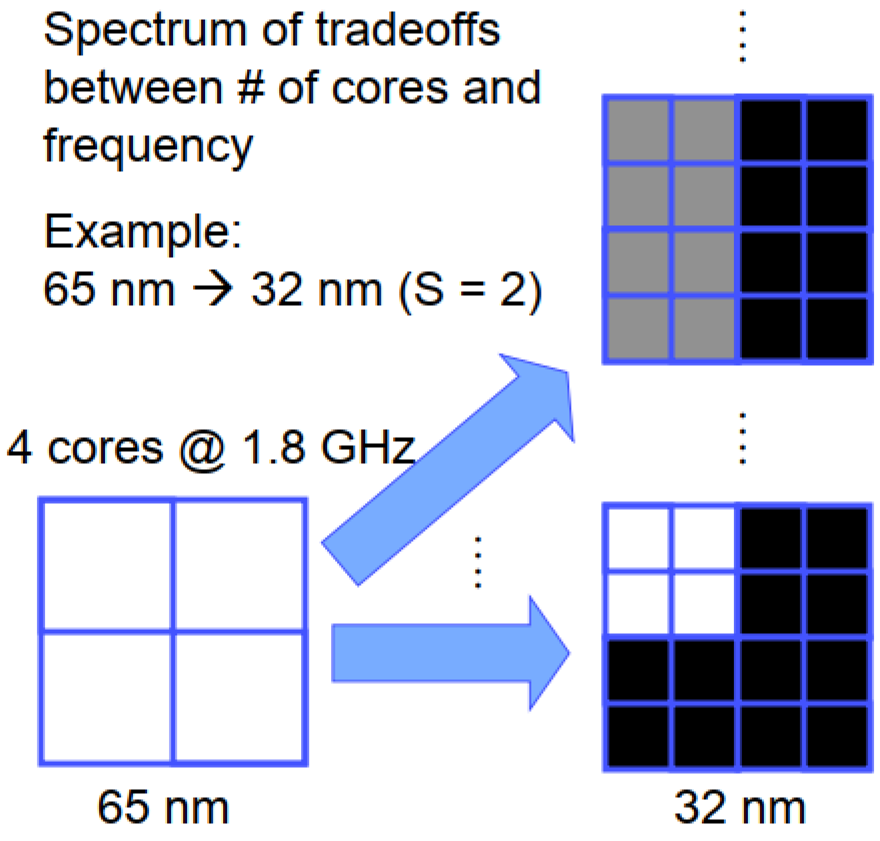
Core 1		Core 2
Accesses issued		Core 1's accesses seen
Store B	Reordered	Load A
Load A		Load D
Store C		Store B
Store A		Store C
Load C		Load C
Load D		Store A

Dark Silicon

Multicore has hit the Utilization Wall

Spectrum of tradeoffs between # of cores and frequency

Example:
65 nm → 32 nm (S = 2)



4x4 cores @ .9 GHz
(*GPUs of future?*)

2x4 cores @ 1.8 GHz
(8 cores dark, 8 dim)

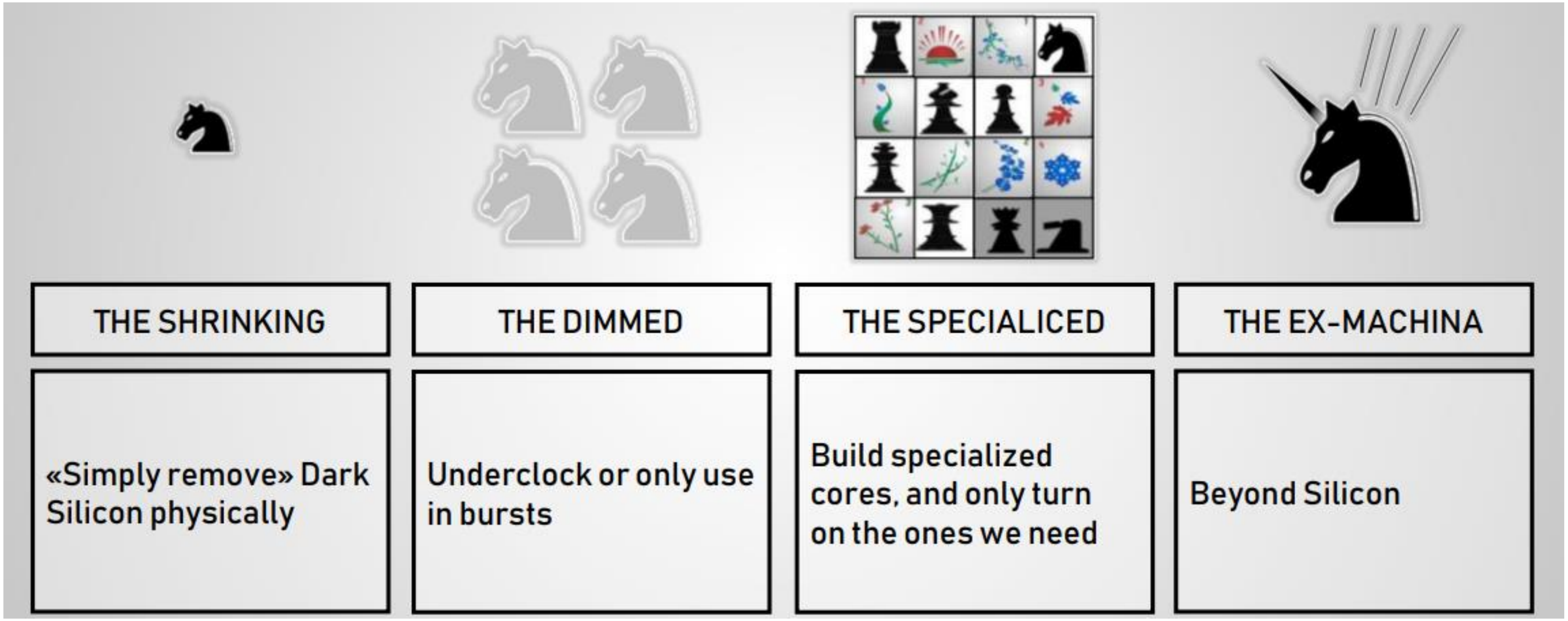
(*Intel/x86 Choice, next slide*)

4 cores @ 2x1.8 GHz
(12 cores dark)

[Goulding, Hotchips 2010, IEEE Micro 2011]
[Esmaeilzadeh ISCA 2011]
[Skadron IEEE Micro 2011]
[Hardavellas, IEEE Micro 2011]

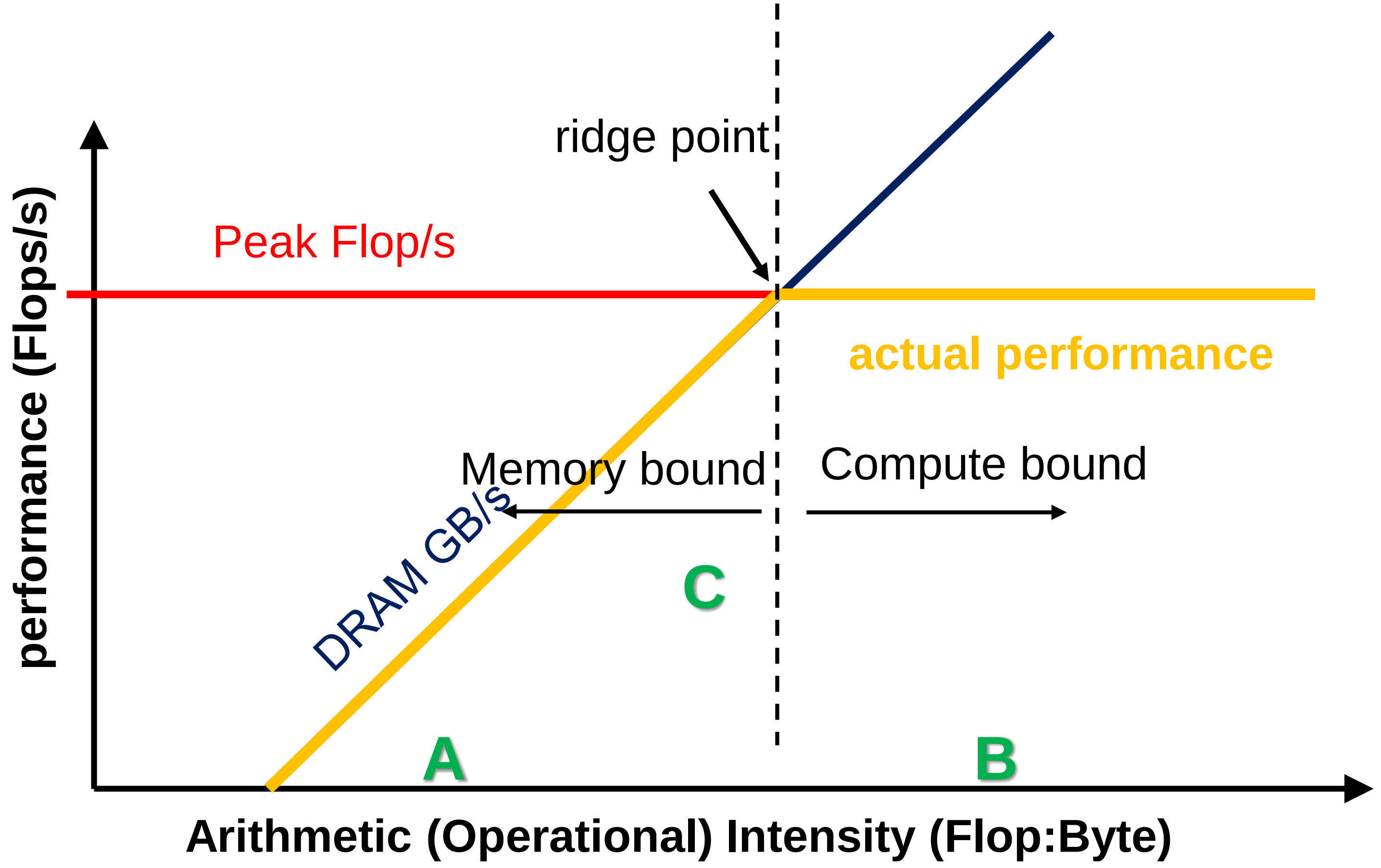
source: Prof. Michael Taylor @ UW

4 horsemen



THE SHRINKING	THE DIMMED	THE SPECIALICED	THE EX-MACHINA
«Simply remove» Dark Silicon physically	Underclock or only use in bursts	Build specialized cores, and only turn on the ones we need	Beyond Silicon

Roofline Model



Specialized Architecture (T8)

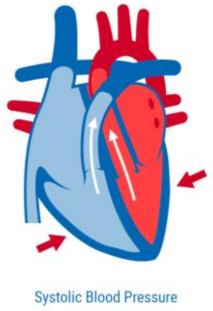
Important Topics

- Why Specialized Architecture?
- Reconfigurable Computing
  - What is RC?
  - FPGAs
    - Basic architecture
    - Limitations
- DSAs
  - Motivation
  - Systolic Array

Guidelines for DSAs

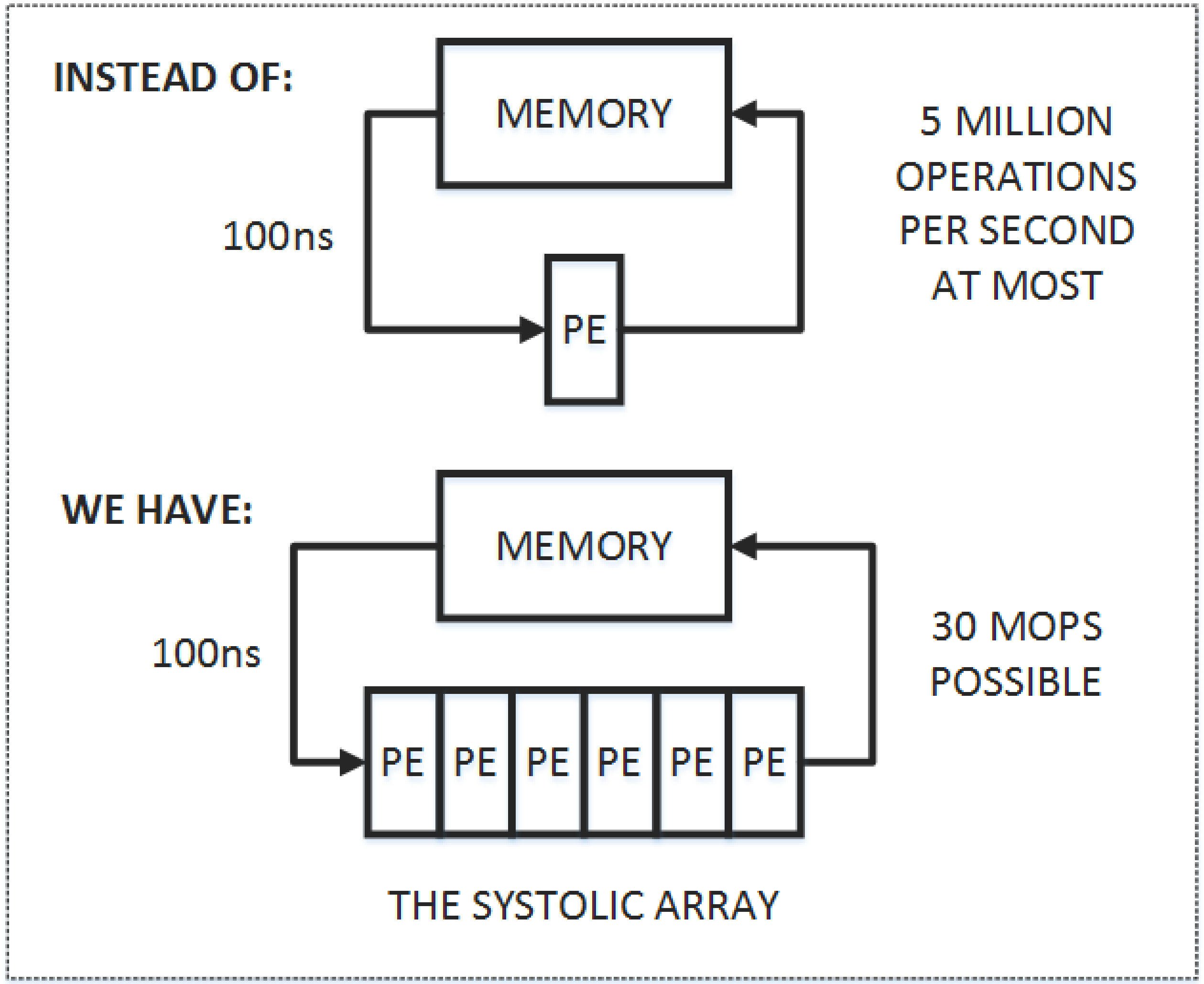
- Use dedicated memories to minimize data movement
  - **Hardware-controlled multi-level cache -> domain-specific software controlled scratch-pad**
- Invest resources into more arithmetic units or bigger memories
  - **Core optimization (OoO, speculation, threading, etc) -> more domain-specific FU/memory**
- Use the easiest form of parallelism that matches the domain
  - **MIMD -> SIMD or VLIW that matches domain**
- Reduce data size and type to the simplest needed for the domain
  - **General-purpose 32/64 integer/float -> domain-specific 8/16 int/float**
- Use a domain-specific programming language
  - **General-purpose C/C++/Fortran -> Domain-specific language (TensorFlow for DNN)**

Systolic Array



"脉动"

Memory: Heart
PEs: Cell



- Goal: design an accelerator that has
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth
- Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - such that they collectively transform a piece of input data before outputting it to memory
- Benefit: Maximizes computation done on a single piece of data element brought from memory