

CDA5106 - Advanced Computer Architecture

Final Exam Review

Kobee Raveendran

1 Module 1: High-Performance Microprocessor Architecture

1.1 Module 1.2: Power Wall and Dennard Scaling

1.1.1 Notes

- energy: ability of a physical system to do work on other physical systems (unit: joule)
- power: rate at which energy is transformed (unit: watt; 1 watt = 1 joule delivered per second)
 - power = $V \cdot I$ (V = voltage, I = current)
- for capacitors:
 - energy stored = $0.5 \cdot C \cdot V^2$ (C = capacitance, V = voltage)
 - if a capacitor is drained at a frequency of f per second: power = $\frac{\text{energy}}{\text{second}} = 2 \cdot 0.5CV^2 = CV^2$
- Power wall problem
 - $P_{dyn} = ACV^2f$
 - A : fraction of gates actively switching
 - C : total capacitance of all gates
 - V : supply voltage
 - f : frequency of switching
- Power wall fundamentals
 - max frequency vs. threshold voltage:
 - $f_{max} = c \cdot \frac{(V-V_{thd})^{1.3}}{V}$
- Dennard Scaling Example (old)
 - if gate length (transistor size) scales by $S = 0.7$ (both length and width), then:
 - capacitance scales by $S = 0.7$
 - original area scales by $S^2 = 0.5$
 - number of transistors scales by $\frac{1}{S^2} \approx 2$
 - supply voltage (V) scales by $S = 0.7$
 - frequency (f) scales by $\frac{1}{S} = 1.4$
 - then, **dynamic power** $P_{dyn} = ACV^2f$
 - and **new dynamic power** $P'_{dyn} = A'C'V'^2f'$
 - $P'_{dyn} = (2A)(0.7C)(0.7V)^2(1.4f) \approx 1 \cdot ACV^2f = P_{dyn}$
- Post Dennard Scaling example (new)
 - capacitance scales by $S = 0.7$
 - number of transistors scales by $\frac{1}{S^2} = 2$

- supply voltage (V) cannot scale without also scaling threshold voltage (V_{thd}), and doing that increases static power exponentially
- frequency (f) scales by $\frac{1}{S} = 1.4$
- result: dynamic power doubles every generation
- $P_{dyn} = ACV^2f$
- $P'_{dyn} = A'C'V'^2f' = (2A)(0.7C)(1 \cdot V)^2(1.4f) \approx 2 \cdot P_{dyn}$

1.1.2 Exercises

- Suppose that instead of progressing at a ratio of 0.7, Moore's law slows down and transistor gate length scales at a ratio of 0.8 instead. Find the dynamic power consumption under *unlimited* and *limited* scaling for the next process generation.
 - Unlimited/old scaling rule
 - * gate length scales by $S = 0.8$
 - * capacitance scales by $S = 0.8$
 - * original area scales by $S^2 = 0.64$
 - * num transistors thus scales by $\frac{1}{S^2} = 1.56$
 - * supply voltage scales by $S = 0.8$
 - * frequency scales by $\frac{1}{S} = 1.25$
 - * dynamic power stays constant:

$$P_{dyn} = (1.56A)(0.8C)(0.8V)^2(1.25f)$$
 - leakage-limited/new scaling
 - * capacitance scales by $S = 0.8$
 - * num transistors scales by $\frac{1}{S^2} = 1.56$
 - * supply voltage does not scale without scaling threshold voltage too, which increases static power exponentially
 - * frequency scales by $\frac{1}{S}1.25$
 - * dynamic power consumption increases:

$$P'_{dyn} = (1.56A)(0.8C)(V)^2(1.25f) = 1.56 \cdot P_{dyn}$$
- With limited voltage scaling, suppose that we want to keep the dynamic power consumption constant in the next generation by keeping frequency constant and reduce die area. How much should we reduce die area to achieve that?
 - gate length scales by $S = 0.7$
 - capacitance scales by $S = 0.7$
 - original area scales by $S^2 = 0.5$
 - supply voltage and frequency are constant
 - dynamic power consumption must stay constant: $P'_{dyn} = P_{dyn}$

$$ACV^2f = A'(0.7C)V^2f \longrightarrow A = 0.7A'$$
 - number of transistors in the next generation: $A' = 1.4A$ (instead of $2A$ like before; i.e. 70% of $2A$)
 - thus die area shrinks by 30%
- Describe the difference between energy and power.
Power is the rate of energy consumption.
- Describe the impact of threshold voltage choice on static and dynamic power consumption as transistors are scaled down.
If threshold voltage is lowered, dynamic power decreases (nearly linearly) but static power increases exponentially.
- How has processor design adapted to the power wall problem?
Stalling frequency growth, multicore, and sophisticated power management (clock gating, voltage and frequency scaling, power gating).

1.1.3 Overview of ILP Techniques

Caches example

- processor with 1-ns clock
- 64KB cache memory with 2-ns read time, 95% hitrate
- 512MB main memory with 150-ns read time
- What is the average access time (AAT) in this memory system?

Answer:

- hits: $95 \cdot 2$ ns, misses: $5 \cdot (2 + 150)$ ns
- total = hit time + miss time = $190 + (10 + 750) = 950ns$
- $AAT = \frac{total}{100} = 9.5ns$

2 Module 2: Performance, Cost, and Reliability of Micro-processors

2.1 Performance Evaluation 1

2.1.1 Amdahl's Law

- performance improvement (“speedup”) is limited by the part you can’t improve
- (s) $Speedup_{enhanced}$ = best case speedup from gizmo alone
- (f) $Fraction_{enhanced}$ = fraction of task that gizmo can enhance
- $s_{overall} = \frac{1}{(1-f) + \frac{f}{s}}$

Example:

- jet plane wing simulation, where 1 run takes 1 week on your computer
- your program is 80% parallelizable
- new supercomputer has 100,000 processors
- $s = 100,000$
- $f = 0.8$
- overall speedup: $s_{overall} = \frac{1}{(1-f) + \frac{f}{s}} = \frac{1}{(1-0.8) + \frac{0.8}{100000}} \approx \frac{1}{0.2} = 5$
- only about 5 times faster (33 hours instead of 1 week), but not worth the high price tag (using a cheaper computer with only 100 processors instead yields a 4.8X speedup!)

More examples:

Ex 1:

- $f = 0.95$
- $s = 1.10$
- $s_{overall} = \frac{1}{(1-0.95) + \frac{0.95}{1.10}} = 1.094 \approx 1.10$

Ex 2:

- $f = 0.05$
- $s \rightarrow \infty$
- $s_{overall} = 1.053$

2.1.2 Run Time

- CPU time = clock cycle count \times cycle time
- cycles per instruction (CPI) = $\frac{\text{clock cycle count}}{\text{instruction count}}$
- CPU time = IC \times CPI \times CT

2.2 Performance Evaluation 2

Determine speedup by comparing program times with respect to a reference machine.

- arithmetic mean (which one should we trust?):

	Computer A	Computer B	B vs. A
Program P1	2X faster	4X faster	2X faster
Program P2	5X faster	15X faster	3X faster
Average	3.5X	9.5X	

Speedups:

- method 1: program-wise $\rightarrow \frac{2+3}{2} = 2.5\text{X}$ faster
- method 2: machine-wise $\rightarrow \frac{9.5}{3.5} = 2.71\text{X}$ faster

- geometric mean (consistent):

$$gmean = \sqrt[n]{\prod_{i=1}^n} = \exp\left(\frac{\frac{1}{n} \sum_{i=1}^n \ln(x_i)}{n}\right)$$

	Computer A	Computer B	B vs. A
Program P1	2X faster	4X faster	2X faster
Program P2	5X faster	15X faster	3X faster
Average	$\sqrt{10}$	$\sqrt{60}$	$\sqrt{6}$

Speedups:

- method 1: program-wise \rightarrow B is $\sqrt{2 \cdot 3} = \sqrt{6}\text{X}$ faster
- method 2: machine-wise \rightarrow B is $\sqrt{60} \cdot \sqrt{10} = \sqrt{6}\text{X}$ faster

- (also important): geometric standard deviation

$$gstdev = \exp\left(\sqrt{\frac{\prod_{i=1}^n (\ln x_i - \ln gmean)^2}{n}}\right)$$

in plain English: for each “component” speedup vs. ref machine, take its natural log and subtract the natural log of the gmean from that. Square it and multiply all of these together, then divide by n. Finally take the square root of this, then take e to the power of the result.

2.2.1 Exercises

Given the following table of speedups for machines A and B relative to a reference machine:

Prog	X (secs)	A (secs)	B (secs)
App 1	30	15	10
App 2	20	15	10
App 3	40	20	30
App 4	15	20	15

Compute the following (see post-computation table below to find them all):

- geometric speedup of machine A vs. base machine X
from the table, we find that A has a 1.41X speedup over X
- geometric speedup of machine B vs. base machine X
from the table, we find that B has a 1.68X speedup over X
- geometric speedup of machine B vs. machine A
from the table, we find that B has a 1.19X speedup over A

- geometric standard deviation of the speedup of machine A over machine X

$$gstd = \exp(\sqrt{\frac{1}{4} \cdot \ln^2(\frac{2}{1.41}) \ln^2(\frac{1.33}{1.41}) \ln^2(\frac{2}{1.41}) \ln^2(\frac{0.75}{1.41})})$$

$$gstd = 1.002255... \approx 1$$

Prog	A vs. X	B vs. X	B vs. A
App 1	2X	3X	1.5X
App 2	1.33X	2X	1.5X
App 3	2X	1.33X	0.67X
App 4	0.75X	1X	1.33X
Product	4X	8X	2X
gmean	1.41X	1.68X	1.19X

2.3 Cost and Reliability

2.3.1 Failure Rates (λ)

- λ = the number of failures that occur per unit time in a component/system
- FIT (failure in time) = number of failures in 10^9 hours
- example: 10,000 microprocessor chips used for 1,000 hours, and 8 of them fail. Failure rate is thus $\frac{8}{10,000 \cdot 1,000} = 8 \cdot 10^{-7}$ (failures per hour per chip) $\cdot 10^9$ hours = 800 FITs

2.3.2 Reliability Metrics

- $R(t)$ = probability that the system still works correctly at time t
- $W_N(t)$ = number of items (of the same kind) that would still be working at time t
- if λ is constant, then $R(t) = e^{-\lambda t}$
- Mean Time Between Failure (MTBF) = $\frac{1}{\lambda}$

2.3.3 System Reliability

Assume that:

- M components are in the system with failure rates $\lambda_1, \lambda_2, \dots, \lambda_m$
- for the system to work properly, all components must also work properly
- a component's reliability is independent of any other component's reliability
- then, system failure rate = sum of component's failure rates
- $R_{sys}(t) = R_1(t) \cdot R_2(t) \cdot \dots \cdot R_m(t) = e^{-\lambda_1 t} \dots e^{-\lambda_m t} = e^{-(\lambda_1 + \lambda_2 + \dots + \lambda_m)t} = e^{-\lambda_{sys} t}$

Other metrics:

- Mean Time To Repair (MTTR): mean time to repair/recover from a fault
- Mean Time Between Failure (MTBF): mean time between 2 consecutive failures
- if each failure is repaired, then $MTBF = MTTF + MTTR$
- usually, $MTTF \gg MTTR$, so MTBF and MTTF are often interchangeable

2.3.4 Examples

Assume a disk subsystem with:

- 10 disks each rated at 10^6 -hour MTTF
- 1 SCSI controller rated at $5 \cdot 10^5$ -hour MTTF
- 1 power supply rated at $2 \cdot 10^5$ -hour MTTF
- 1 fan rated at $2 \cdot 10^5$ -hour MTTF
- 1 SCSI cable rated at 10^6 -hour MTTF

Find the failure rate of the entire disk subsystem.
 $R_{sys}(t) = 10 \cdot \frac{1}{10^6} + \frac{1}{5 \cdot 10^5} + \frac{2}{2 \cdot 10^5} + \frac{1}{10^6} = \frac{10+2+5+5+1}{10^6} = \frac{23}{10^6} = \frac{23,000}{10^9} = 23,000 \text{ FIT}$
Thus, MTTF = $\frac{1}{\lambda_{sys}} = \frac{1}{23,000} \cdot 10^9 \approx 43,500 \text{ hours}$

3 Instruction Set Design

3.1 Instruction Set Architecture 1

3.1.1 Styles of ISAs

- Stack:
 - push <addr>, pop <addr> (or ALU instructions)
 - ALU: pop two entries, perform ALU operation, push result onto stack
 - compact instruction format
 - * all calculation operations take 0 operands
 - * flexible; used for compiling Java bytecode (not dependent on registers in architecture [but all have stacks])
- Accumulator:
 - load/store/ALU <addr> (result affects accumulator register)
 - also very compact (all operations take 1 operand, the other is implicitly the accumulator register)
 - less dependence on memory than stack-based
- Register-memory:
 - load/store/ALU <reg>, <reg/addr>
 - * at most one operand can be a memory address
 - * leftmost register is the destination (if applicable)
- Load-Store:
 - load/store <reg>, <addr>
 - ex: ALU <reg1>, <reg2>, <reg3>: reg1 is destination, other 2 are source registers

Example for adding two numbers:

Stack	Accumulator	Register-memory	Load-Store
push A	load A	load R1 A	load R1 A
push B	add B	add R1 B	load R2 B
add	store C	store C R1	add R3 R1 R2
pop C			store C R3

Pros/cons:

- load-store
 - (+) fixed length instructions possible (allows for easy fetch/decode)
 - (+) simpler hardware: efficient pipeline and potentially lower cycle time
 - (-) higher instruction count

- (-) fixed-length instructions can be wasteful (more bits than needed for some instructions)
- register-memory
 - (+) no need for extra loads
 - (+) better usage of bits (these pros lead to better code density)
 - (-) destroys source operand(s) (i.e. `add R1 R2`)
 - (-) may impact cycles per instruction
- memory-memory
 - (+) most compact (code density)
 - (-) high memory traffic (thus bottlenecked by memory)

3.2 Instruction Set Architecture 2

3.2.1 RISC and Common Addressing Modes

- register
 - `add R4 R3 // R4 = R4 + R3`
 - used when value is in a register
- immediate
 - `add R4 #3 // R4 = R4 + 3`
 - used for small constants (which occur frequently)
- displacement
 - `add R4 100(R1) // R4 = R4 + MEM[100 + R1]`
 - accesses the frame (arguments, local variables)
 - access the global data segment
 - accesses the fields of a data struct
- register deferred/register indirect
 - `add R3 (R1) // R3 = R3 + MEM[R1]`
 - accesses using a computed memory address
- indexed
 - `add R3 (R1 + R2) // R3 = R3 + MEM[R1 + R2]`
 - array accesses: $R1 = \text{base}$, $R2 = \text{index}$
- direct/absolute
 - `add R1 (1001) // R1 = R1 + MEM[1001]`
 - accessing global (“static”) data
- memory deferred/memory indirect
 - `add R1 @(R3) // R1 = R1 + MEM[MEM[R3]]`
 - pointer dereferencing: `x = *p`; (if `p` is not register-allocated)
- autoincrement/postdecrement
 - `add R1 (R2)+ // R1 = R1 + MEM[R2]; R2 = R2 + d` (d is the size of the operation)
 - looping through arrays, stack pop
- autodecrement/predecrement
 - `add R1 -(R2) // R1 = R1 + MEM[R2]; R2 = R2 - d` (d is the size of the operation)

- same uses as autoincrement, stack push
- scaled
 - `add R1 100(R2) [R3] // R1 = R1 + MEM[100 + R2 + R3 * d]`
 - array accesses for non-byte-sized elements

3.3 Instruction Set Architecture 3

3.3.1 Condition Codes (for branch instructions)

- Z
 - zero flag
 - indicates the result of an arithmetic/logical expression is zero
- C
 - carry flag
 - indicates that an operation has a carry out. Enables numbers larger than a single word to be added/subtracted
- S/N
 - sign/negative flag
 - indicates the result of an operation is negative
- V/O/W
 - overflow flag
 - indicates the result of an operation is too large to fit in a register (using 2's complement representation)

3.3.2 Instruction Encoding Tradeoffs

- variable width
 - (+) very versatile, uses memory efficiently
 - (-) instruction words must be decoded before number of bytes is known (harder to fetch/decode)
- fixed width
 - (+) every instruction word is an instruction, thus easier to fetch/decode
 - (-) uses memory inefficiently (same num. bits even for short instructions)
- hybrid
 - primarily for embedded processors to conserve memory
 - often use a subset of instructions, fewer registers, or even instruction compression

3.4 ISA Examples

3.4.1 MIPS

Characteristics:

- load/store ISA: only loads/stores can have memory operands, makes for easy pipelining and uniform instruction width
- fixed instruction width = easy fetch and decode
- small number of addressing modes = easy pipelining
- large register file: 32 integer and 32 floating point registers
- aligned memory = easy data fetching

- quantitatively designed

Instruction format:

- R: op, rs, rt, rd, shamt, funct (shamt = shift amount, funct = ALU function)
- I: op, rs, rt, 16-bit address
- J: op, 26-bit address

4 Memory Hierarchy

4.1 Basics of Cache Architecture

4.1.1 Cache Organization

A cache is similar to a table

- a **set** in cache = a row in table
- a **way** in cache = a column in table
- a **line** in cache = a cell in table

4.1.2 Choices on Cache Associativities

- direct mapped cache: a block can be placed in only one line in the cache (i.e. vertical array table)
 - rigid placement of blocks in cache set
 - usually has higher miss rate
 - but power efficient (no need to search an entire way if the way is only one cell)
- fully associative cache: a block can be placed in any line in the cache (i.e. a horizontal array table)
 - flexible placement of blocks in cache set
 - has the lowest miss rate
 - but power hungry (have to search entire set [aka the whole cache] to find a block)
- set-associative cache: a block can be placed in one of the ways in a set (i.e. a 2D array table)

4.1.3 Cache Parameters

- SIZE = total amount of cache data storage in bytes
- BLOCKSIZE = total number of bytes in a single block
- ASSOC = associativity (number of lines in a set)

Formulas:

$$\# \text{ of blocks in a cache} = \frac{SIZE}{BLOCKSIZE}$$

$$\# \text{ of sets in a cache} = \frac{\# \text{ cache blocks}}{ASSOC} = \frac{SIZE}{BLOCKSIZE \cdot ASSOC}$$

$$\# \text{ of index bits} = \log_2(\#sets)$$

$$\# \text{ of block offset bits} = \log_2(BLOCKSIZE)$$

$$\# \text{ of tag bits} = 32 - \#index \text{ bits} - \#offset \text{ bits}$$

4.1.4 Examples

Example 1: Processor accesses a 256B direct-mapped cache, which has a block size of 32B, with the below sequence of addresses. Show the contents of the cache after each access, and count the number of hits and replacements.

index bits = $\log_2(8) = 3$
offset bits = $\log_2(32) = 5$
tag bits = $32 - 3 - 5 = 24$

Address (hex)	Tag (hex)	Index and offset bits (binary)	Index (decimal)	Comment
0xFF0040E0	0xFF0040	1110 0000	7	miss
0xBEEF005C	0xBEEF00	0101 1100	2	miss
0xFF0040E2	0xFF0040	1110 0010	7	hit
0xFF0040E8	0xFF0040	1110 1000	7	hit
0x00101078	0x001010	0111 1000	3	miss
0x002183E0	0x002183	1110 0000	7	miss/rep
0x00101064	0x001010	0110 0100	3	hit
0x0012255C	0x001225	0101 1100	2	miss/rep
0x00122544	0x001225	0100 0100	2	hit

4.1.5 Write Updates

- Write-through (WT) policy
 - writing to some level in the cache also means writing through to subsequent levels in the cache (i.e. next level in the memory hierarchy)
- Write-back (WB) policy
 - write only to the specified cache level, and set its dirty bit
 - when the block you wrote to is replaced (evicted), write the block to the next level of memory hierarchy
- Write-allocate (WA) policy
 - bring block into cache if the write misses (just like in read misses)
 - typically used in conjunction with write-back (WBWA)
- Write-no-allocate (NA) policy
 - do not bring the block into the cache if write misses
 - *must* be used in conjunction with write-through (WTNA)

4.1.6 Victim Cache

- small fully-associative cache that sits alongside the primary cache
- when main cache evicts a block, the victim cache takes the evicted block (called the “victim”)
- when the main cache misses, it searches the victim cache for recently discarded blocks; a victim cache hit means the main cache doesn’t have to go to memory to search for a block
- example:
 - L1 cache (initially a set contains just A)
 - 2-entry victim cache that contains X, and Y (current LRU)
 - B misses in L1, evicts A, A goes to victim cache and replaces Y (previous LRU); X is new LRU
 - A then misses in L1 but hits in victim cache, so A and B *swap* positions (A goes to L1, B goes to VC; note that X (prev LRU) is not replaced in the case of victim cache hits)
 - thus a victim cache is useful in cases of repeated conflicts and gives the illusion of set-associativity

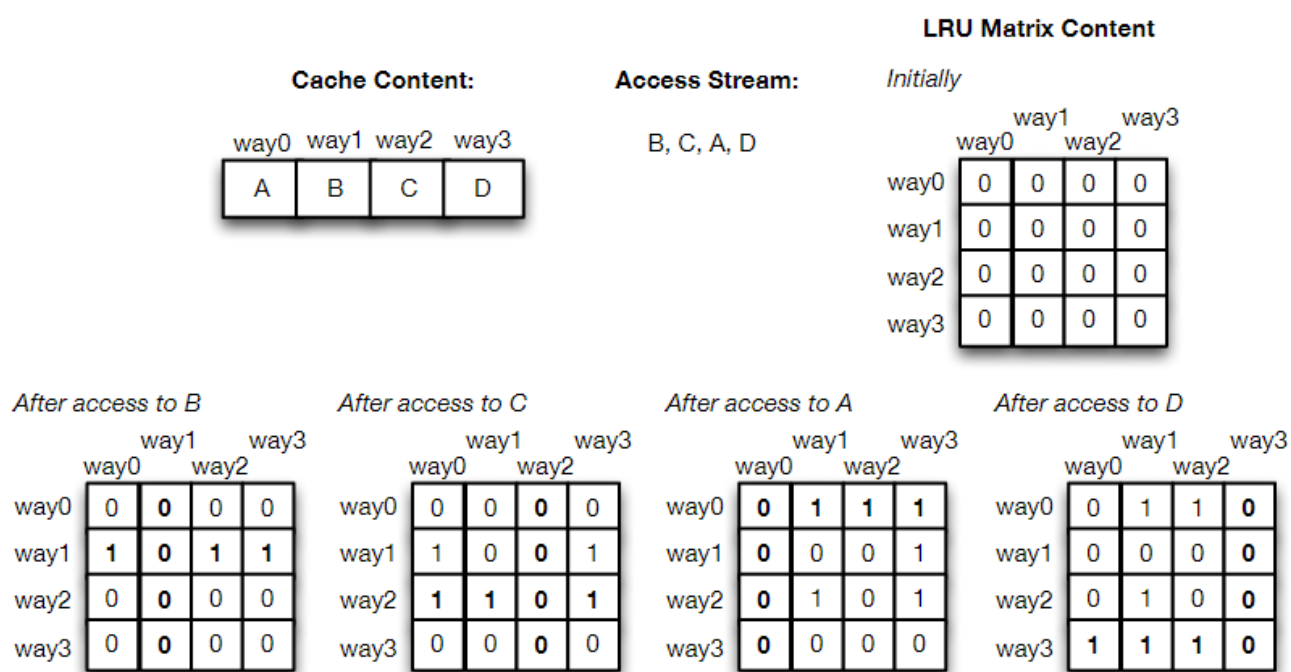
4.2 Replacement Policies

4.2.1 Optimal Replacement Policy

- look into future and determine when each block in a set is needed again (if at all)
- replace the block needed farthest in the future
- note: not practical since we don't know in advance when blocks are needed, but this is the theoretical gold standard with which other replacement policies are evaluated against

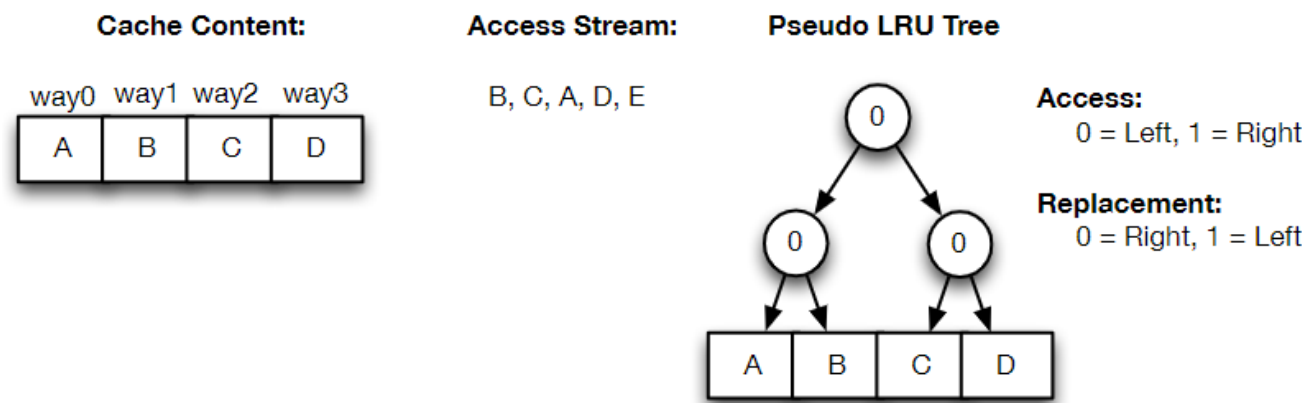
4.2.2 LRU implementation

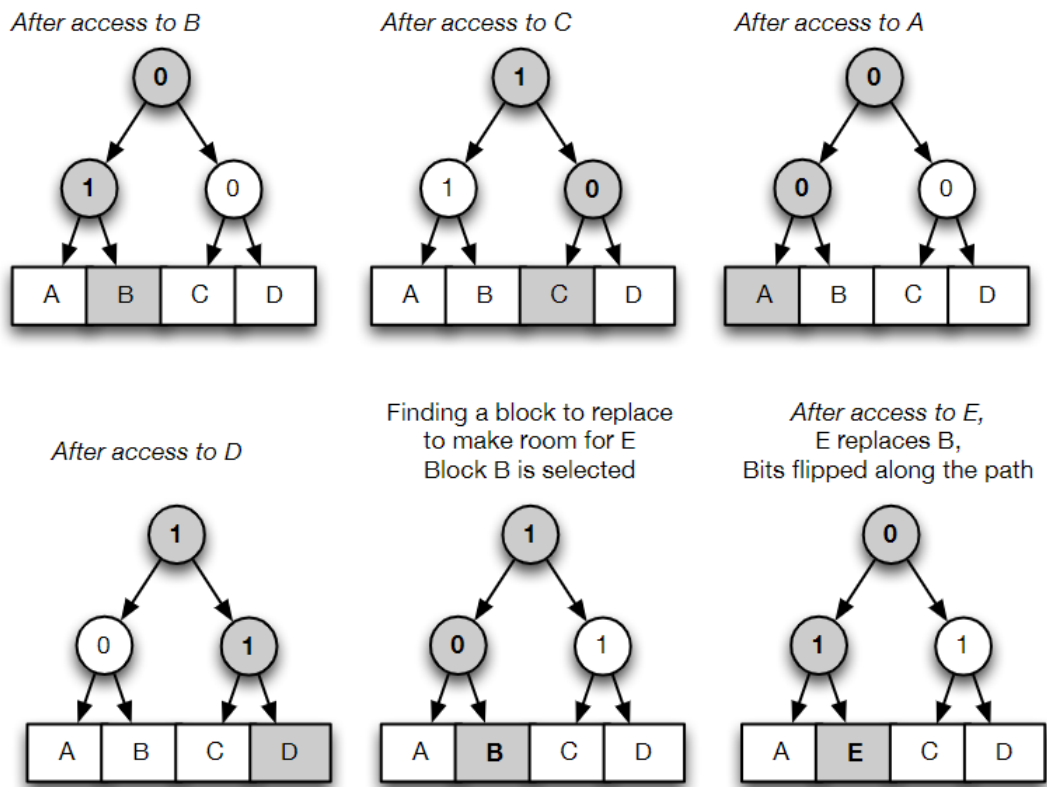
- assign a row and column in LRU matrix to each way in the set
- if there's a hit in a way, set the row corresponding to that way, and unset the column corresponding to that way
- the number of 1's in a row specifies the MRU order (thus, the LRU is the one with all 0's)



4.2.3 Pseudo-LRU implementation

- LRU implementation takes $O(way^2)$ space and time; too expensive
- PLRU approximates LRU with decent accuracy
- PLRU complexity is $O(way)$

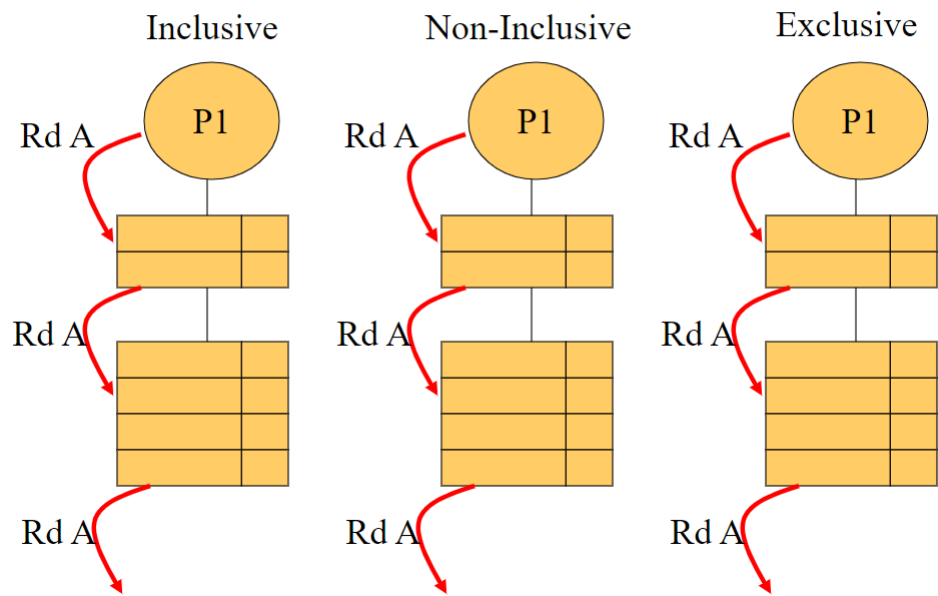




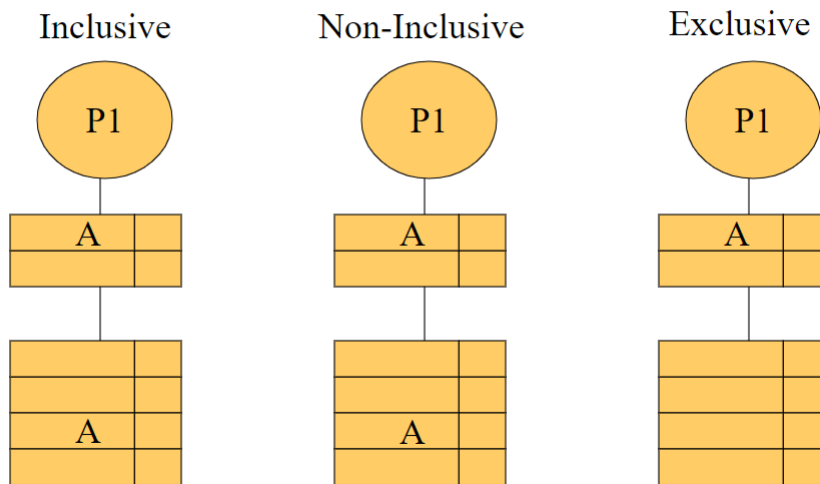
4.3 Inclusion Policies

- block in inner cache level *always* included in outer level cache too: **inclusive**
- block in inner cache level *never* included in outer level cache too: **exclusive**
- block in inner cache level *sometimes* included in outer level cache: **non-inclusive**

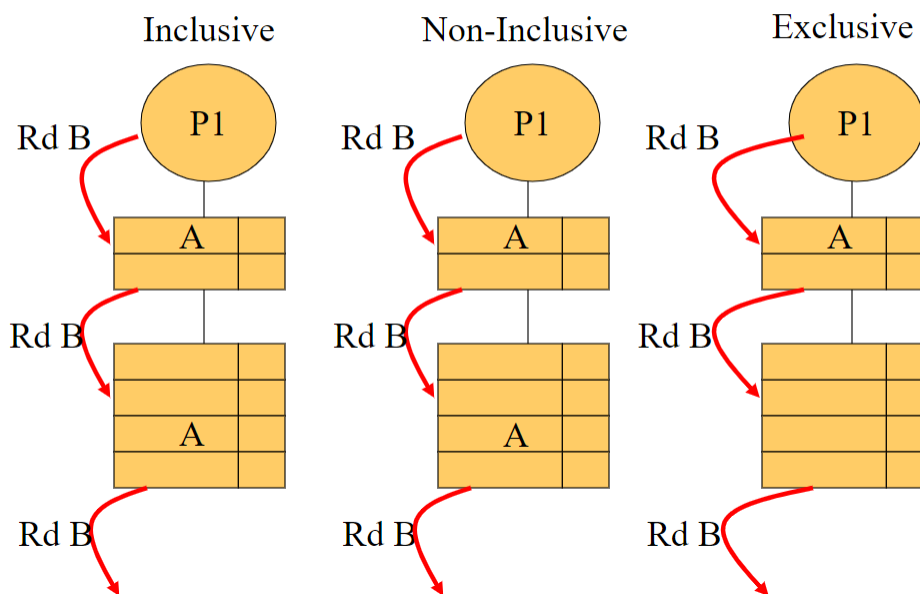
Example: Read miss(A) in L1&L2



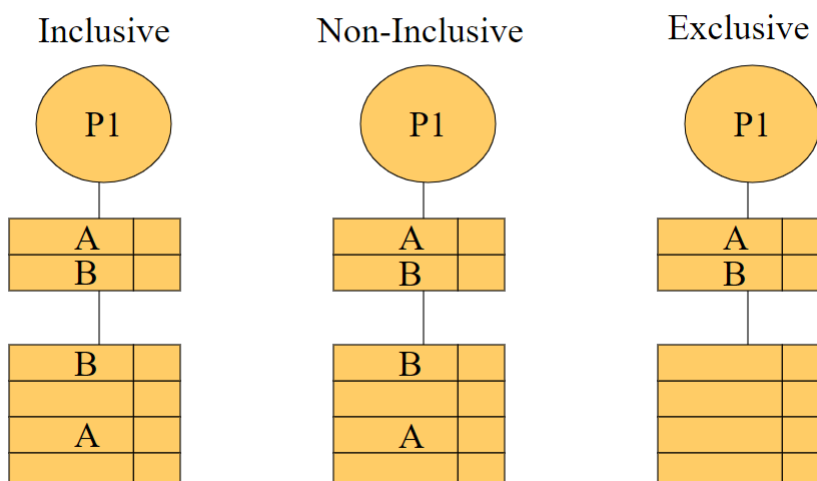
Fill at L1? Fill at L2?



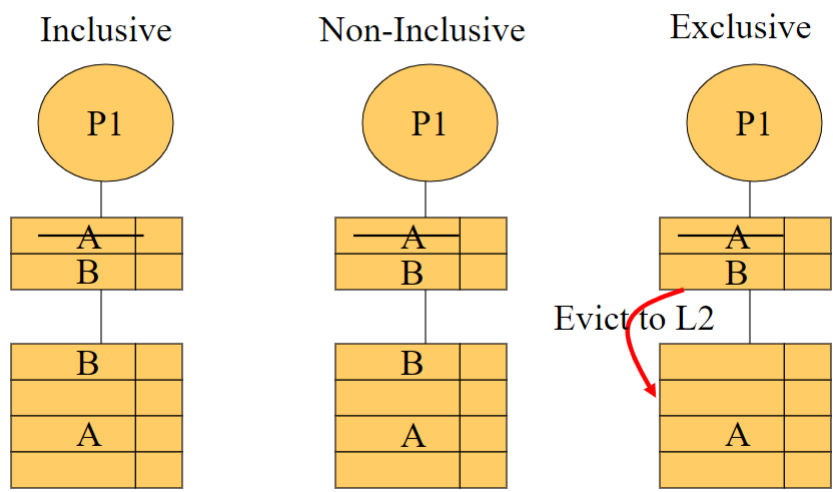
Example: Read miss(B) in L1&L2



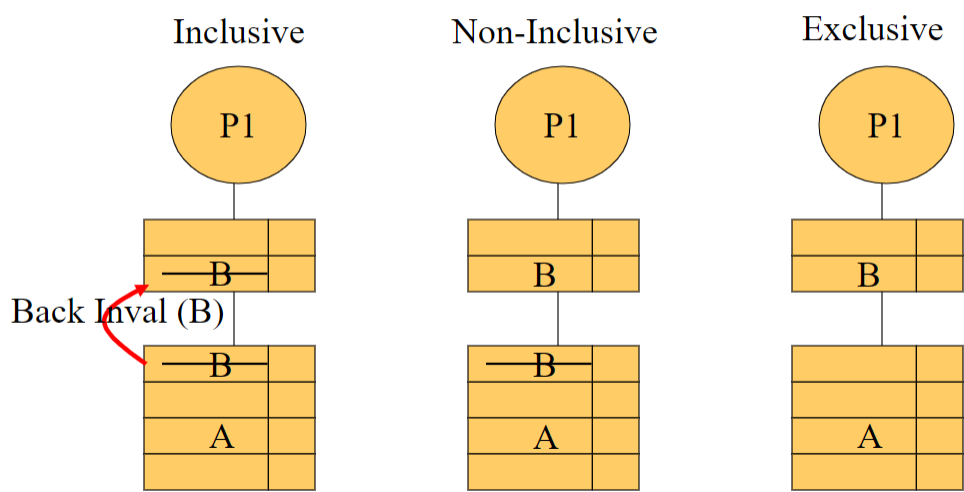
Fill at L1? Fill at L2?



Evict(A) from L1



Evict(B) from L2



4.3.1 Inclusive outer cache Pros/Cons

Pros:

- for most cases, external requests can be checked against the outer cache (if not in outer, cannot be in inner)
- reduces contention for cache tags at inner cache
- external snoop latency (and thus memory latency) is reduced significantly

Cons:

- space wasteful; forced data redundancy
- inflexible (power gating ways makes inner cache ineffective)