5 Solutions

Solution 5.1

5.1.1

a.	web browser, web servers; caches can be used on both sides.
b.	web browser, bank servers; caches could be employed on either.

5.1.2

a.	1. browser cache, size = fraction of client computer disk, latency = local disk latency; 2. proxy/CDN cache, size = proxy disk, latency = LAN + proxy disk latencies; 3. server-side cache, size = fraction of server disks, latency = WAN + server disk; 4. server storage, size = server storage, latency = WAN + server storage Latency is not directly related to cache size.
b.	1. web browser cache, size = % of client hard disk, latency = local hard disk latency; 2. server-side cache, size = % of server disk(s), latency = wide area network(WAN) + server disk latencies; 3. server storage, size = server storage, latency = wide area network(WAN) + server storage latencies; Latency is not directly related to the size of cache.

5.1.3

a.	Pages. Latency grows with page size as well as distance.
b.	Web pages; latency grows with page size as well as distance.

5.1.4

a.	1. browser—mainly communication bandwidth. Buying more BW costs money. 2. proxy cache—both. Buying more BW and having more proxy servers costs money. 3. server-side cache—both. Buying more BW and having larger cache costs money. 4. server storage—server bandwidth. Having faster servers costs money.
b.	1. browser—mainly communication bandwidth; obtaining more bandwidth involves greater cost; 2. server-side cache—both communication and processing bandwidth; obtaining more cache and more bandwidth involves greater cost; 3. server storage—processing bandwidth; obtaining faster processing servers involves greater cost

5.1.5

a.	Depends on the proximity between client interests. Similar clients improves both spatial and temporal locality—mutual prefetching; dissimilar clients reduces both.
b.	Client requests cannot be similar, hence not applicable to this application

5.1.6

a.	Server update the page content. Selectively caching stable content/"expires" header.
b.	Server update to financial details; selectively cache non-financial content

Solution 5.2

5.2.1 4

5.2.2

a.	I, J, B[J][0]
b.	I, J

5.2.3

a.	A[I][J]
b.	A[J][I]

5.2.4

a.	$3186 = 8 \times 800/4 \times 2 - 8 \times 8/4 + 8/4$
b.	$3596 = 8 \times 800/4 \times 2 - 8 \times 8/4 + 8000/4$

5.2.5

a.	I, J, B(J, 0)
b.	I, J

5.2.6

a.	A(I, J), A(J, I), B(J, 0)
b.	A(I, J)

Solution 5.3

5.3.1

a.	Binary address: 1_2, 10000110_2, 11010100_2, 1_2, 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2 Tag: Binary address >> 4 bits Index: Binary address mod 16 Hit/Miss: M, M, M, H, M, M, M, M, M, M, M, M
b.	Binary address: 00000110_2, 11010110_2, 10101111_2, 11010110_2, 00000110_2, 01010100_2, 01000001_2, 10101110_2, 01000000_2, 01101001_2, 01010101_2, 11010111_2 Tag: Binary address >> 4 bits Index: Binary address modulus 16 Hit/Miss: M, M, M, H, M, M, M, M, M, M, M, M

5.3.2

a.	Binary address: 1_2, 10000110_2, 11010100_2, 1_2, 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2 Tag: Binary address >> 3 bits Index: (Binary address >> 1 bit) mod 8 Hit/Miss: M, M, M, H, H, H, M, M, M, M, M, M
b.	Binary address: 00000110_2, 11010110_2, 10101111_2, 11010110_2, 00000110_2, 01010100_2, 01000001_2, 10110000_2, 01000000_2, 01101001_2, 01010101_2, 11010111_2 Tag: Binary address shift right 3 bits Index: (Binary address shift right 1 bit) modulus 8 Hit/Miss: M, M, M, H, M, M, M, H, H, M, H, M

5.3.3

a.	C1: 1 hit, C2: 3 hits, C4: 2 hits. C1: Stall time = 25 × 11 + 2 × 12 = 299, C2: Stall time = 25 × 9 + 3 × 12 = 261, C3: Stall time = 25 × 10 + 4 × 12 = 298
b.	C1: 1 hit, stall time = 25 × 11 + 2 × 12 = 299 cycles C2: 4 hits, stall time = 25 × 8 + 3 × 12 = 236 cycles C3: 4 hits, stall time = 25 × 8 + 5 × 12 = 260 cycles

5.3.4

a.	Using equation on page 351, n = 14 bits, m = 0 (1 word per block) $2^{14} \times (2^0 \times 32 + (32 - 14 - 0 - 2) + 1) = 802$ Kbits Calculating for 16 word blocks, m = 4, if n = 10 then the cache is 541 Kbits, and if n = 11 then the cache is 1 Mbit. Thus the cache has 128 KB of data. The larger cache may have a longer access time, leading to lower performance.
b.	Using equation total cache size = $2^n \times (2^m \times 32 + (32 - n - m - 2) + 1)$, n = 13 bits, m = 1 (2 words per block) $2^{13} \times (2^1 \times 32 + (32 - 13 - 1 - 2) + 1) = 2^{13} \times (64 + 17) = 663$ Kbits total cache size For m = 4 (16 word blocks), if n = 10 then the cache is 541 Kbits and if n = 11 then cache is 1 Mbits. Thus the cache has 64 KB of data. The larger cache may have a longer access time, leading to lower performance.

5.3.5 For a larger direct-mapped cache to have a lower or equal miss rate than a smaller 2-way set associative cache, it would need to have at least double the cache block size. The advantage of such a solution is less misses for near by addresses (spatial locality), but with the disadvantage of suffering longer access times.

5.3.6 Yes, it is possible to use this function to index the cache. However, information about the six bits is lost because the bits are XOR'd, so you must include more tag bits to identify the address in the cache.

Solution 5.4

5.4.1

a.	4
b.	8

5.4.2

a.	64
b.	128

5.4.3

a.	1 + (22/8/16) = 1.172
b.	1 + (20/8/32) = 1.078

5.4.4 3

Address	0	4	16	132	232	160	1024	30	140	3100	180	2180
Line ID	0	0	1	8	14	10	0	1	9	1	11	8
Hit/miss	M	H	M	M	M	M	M	H	H	M	M	M
Replace	N	N	N	N	N	N	Y	N	N	Y	N	Y

5.4.5 0.25

5.4.6 <Index, tag, data>:

$<000001_2, 0001_2$, mem[1024]>
$<000001_2, 0011_2$, mem[16]>
$<001011_2, 0000_2$, mem[176]>
$<001000_2, 0010_2$, mem[2176]>
$<001110_2, 0000_2$, mem[224]>
$<001010_2, 0000_2$, mem[160]>

Solution 5.5

5.5.1

a.	L1 => Write-back buffer => L2 => Write buffer
b.	L1 => Write-back buffer => L2 => Write buffer

5.5.2

a.	1. Allocate cache block for the missing data, select a replacement victim; 2. If victim dirty, put it into the write-back buffer, which will be further forwarded into L2 write buffer; 3. Issue write miss request to the L2 cache; 4. If hit in L2, source data into L1 cache; if miss, send write request to memory; 5. Data arrives and is installed in L1 cache; 6. Processor resumes execution and hits in L1 cache, set the dirty bit.
b.	1. If L1 miss, allocate cache block for the missing data, select a replacement victim; 2. If victim dirty, put it into the write-back buffer, which will be further forwarded into L2 write buffer; 3. Issue write miss request to the L2 cache; 4. If hit in L2, source data into L1 cache, goto (8); 5. If miss, send write request to memory; 6. Data arrives and is installed in L2 cache; 7. Data arrives and is installed in L1 cache; 8. Processor resumes execution and hits in L1 cache, set the dirty bit.

5.5.3

a.	Similar to 5.5.2, except that (2) If victim clean, put it into a victim buffer between the L1 and L2 caches; If victim dirty, put it into the write-back buffer, which will be further forwarded into L2 write buffer; (4) If hit in L2, source data into L1 cache, invalidate the L2 copy;
b.	Similar to 5.5.2, except that – if L1 victim clean, put it into a victim buffer between the L1 and L2 caches; – if L1 victim dirty, put it into the write-back buffer, which will be further forwarded into L2 write buffer; – if hit in L2, source data into L1 cache, invalidate copy in L2;

5.5.4

a.	0.166 reads and 0.160 writes per instruction (0.5 cycles). Minimal read/write bandwidths are 0.664 and 0.640 byte-per-cycle.
b.	0.152 reads and 0.120 writes per instruction (0.5 cycles). Minimal read/write bandwidths are 0.608 and 0.480 byte-per-cycle.

5.5.5

a.	0.092 reads and 0.0216 writes per instruction (0.5 cycles). Minimal read/write bandwidths are 0.368 and 0.0864 byte-per-cycle.
b.	0.084 reads and 0.0162 writes per instruction (0.5 cycles). Minimal read/write bandwidths are 0.336 and 0.0648 byte-per-cycle.

5.5.6

a.	Write-back, write-allocate cache saves bandwidth. Minimal read/write bandwidths are 0.4907 and 0.1152 byte-per-cycle.
b.	Write-back, write-allocate cache saves bandwidth. Minimal read/write bandwidths are 0.4478 and 0.0863 byte-per-cycle

Solution 5.6

5.6.1

12.5% miss rate. The miss rate doesn't change with cache size or working set. These are cold misses.

5.6.2

25%, 6.25% and 3.125% miss rates for 16-byte, 64-byte and 128-byte blocks. Spatial locality.

5.6.3 With next-line prefetching, miss rate will be near 0%.

5.6.4

a.	16-byte.
b.	8-byte.

5.6.5

a.	32-byte.
b.	8-byte.

5.6.6

a.	64-byte.
b.	64-byte.

Solution 5.7

5.7.1

a.	P1	1.61 GHz
	P2	1.52 GHz
b.	P1	1.04 GHz
	P2	926 MHz

5.7.2

a.	P1	8.60 ns	13.87 cycles
	P2	6.26 ns	9.48 cycles
b.	P1	3.97 ns	4.14 cycles
	P2	3.46 ns	3.20 cycles

5.7.3

a.	P1	5.63	P2
	P2	4.05	
b.	P1	2.13	P2
	P2	1.79	

5.7.4

a.	8.81 ns	14.21 cycles	Worse
b.	3.65 ns	3.80 cycles	Better

5.7.5

a.	5.76
b.	2.01

5.7.6

a.	P1 with L2 cache: CPI = 5.76. P2: CPI = 4.05. P2 is still faster than P1 even with an L1 cache
b.	P1 with L2 cache: CPI = 2.01. P2: CPI = 1.79. P2 is still faster than P1 even with an L1 cache

Solution 5.8

5.8.1

a.	Binary address: 1_2, 10000110_2, 11010100_2, 1_2, 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2 Tag: Binary address >> 3 bits Index: (Binary address >> 1 bit) mod 4 Hit/Miss: M, M, M, H, H, H, M, M, M, M, M, M Final contents (block addresses): Set 00: 0_2, 10100000_2, 101000_2 Set 01: 10100010_2, 10_2 Set 10: 11010100_2, 101100_2 Set 11: 10000110_2
b.	Binary address: {bits 7–3 tag, 2–1 index, 0 block offset} $00000\ 11\ 0_2$, Miss $11010\ 11\ 0_2$, Miss $10101\ 11\ 1_2$, Miss $11010\ 11\ 0_2$, Hit $00000\ 11\ 0_2$, Hit $01010\ 10\ 0_2$, Miss $01000\ 00\ 1_2$, Miss $10101\ 11\ 0_2$, Hit $01000\ 00\ 0_2$, Miss $01101\ 00\ 1_2$, Miss $01010\ 10\ 1_2$, Hit $11010\ 11\ 1_2$ Hit Tag: Binary address >> 3 bits Index(or set#): (Binary address >> 1 bit) mod 4 Final cache contents (_block_addresses, in base 2): set: blocks (3 slots for 2-word blocks per set) 00 : 01000000_2, 01000000_2, 01101000_2 01 : 10 : 01010100_2 11 : 00000110_2, 11010110_2, 10101110_2

5.8.2

a.	Binary address: 1_2, 10000110_2, 11010100_2, 1_2, 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2 Tag: Binary address Index: None (only one set) Hit/Miss: M, M, M, H, M, M, M, M, M, M, M, M Final contents (block addresses): 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2

b.	Binary address: {bits 7–0 tag, no index or block offset} 00000110_2, Miss 11010110_2, Miss 10101111_2, Miss 11010110_2, Hit 00000110_2, Hit 01010100_2, Miss 01000001_2, Miss 10101110_2, Miss 01000000_2, Miss 01101001_2, Miss 01010101_2, Miss, (LRU discard block 10101111_2) 11010111_2, Miss, (LRU discard block 01010100_2) Tag: Binary address Final cache contents (*block* addresses): (8 cache slots, 1-word per cache slot) 00000110_2 11010110_2 01010101_2 11010111_2 01000001_2 10101110_2 01000000_2 01101001_2 01010101_2 11010111_2

5.8.3

a.	Binary address: 1_2, 10000110_2, 11010100_2, 1_2, 10000111_2, 11010101_2, 10100010_2, 10100001_2, 10_2, 101100_2, 101001_2, 11011101_2 Hit/Miss, LRU: M, M, M, H, H, H, M, M, M, M, M, M Hit/Miss, MRU: M, M, M, H, H, H, M, M, M, M, M, M Given 2 word blocks, the best miss rate is 9/12.
b.	Binary address: {bits 7–1 tag, 1 block offset} (8 cache slots, 2-words per cache slot) $0000011\ 0_2$, Miss $1101011\ 0_2$, Miss $1010111\ 1_2$, Miss $1101011\ 0_2$, Hit $0000011\ 0_2$, Hit $0101010\ 0_2$, Miss $0100000\ 1_2$, Miss $1010111\ 0_2$, Hit $0100000\ 0_2$, Hit $0110100\ 1_2$, Miss $0101010\ 1_2$, Hit $1101011\ 1_2$, Hit No need for LRU or MRU replacement policy, hence best miss rate is 6/12.

5.8.4

a.	Base CPI: 2.0 Memory miss cycles: 125 cycles/(1/3) ns/clock = 375 clock cycles 1. Total CPI: 2.0 + 375 × 5% = 20.75/39.5/11.375 (normal/double/half) 2. Total CPI: 2.0 + 15 × 5% + 375 × 3% = 14/25.25/8.375 3. Total CPI: 2.0 + 25 × 5% + 375 × 1.8% = 10/16.75/6.625
b.	Base CPI: 2.0 Memory miss cycles: 100 clock cycles 1. Total CPI = base CPI + memory miss cycles × 1st level cache miss rate 2. Total CPI = base CPI + memory miss cycles × global miss rate w/2nd level direct-mapped cache + 2nd level direct-mapped speed × 1st level cache miss rate 3. Total CPI = base CPI + memory miss cycles × global miss rate w/2nd level 8-way set assoc cache + 2nd level 8-way set assoc speed × 1st level cache miss rate 1. Total CPI (using 1st level cache): 2.0 + 100 × 0.04 = 6.0 1. Total CPI (using 1st level cache): 2.0 + 200 × 0.04 = 10.0 1. Total CPI (using 1st level cache): 2.0 + 50 × 0.04 = 4.0 2. Total CPI (using 2nd level direct-mapped cache): 2.0 + 100 × 0.04 + 10 × 0.04 = 6.4 2. Total CPI (using 2nd level direct-mapped cache): 2.0 + 200 × 0.04 + 10 × 0.04 = 10.4 2. Total CPI (using 2nd level direct-mapped cache): 2.0 + 50 × 0.04 + 10 × 0.04 = 4.4 3. Total CPI (using 2nd level 8-way set assoc cache): 2.0 + 100 × 0.016 + 20 × 0.04 = 4.4 3. Total CPI (using 2nd level 8-way set assoc cache): 2.0 + 200 × 0.016 + 20 × 0.04 = 6.0 3. Total CPI (using 2nd level 8-way set assoc cache): 2.0 + 50 × 0.016 + 20 × 0.04 = 3.6

5.8.5

a.	Base CPI: 2.0 Memory miss cycles: 125 cycles/(1/3) ns/clock = 375 clock cycles Total CPI: 2.0 + 15 × 5% + 50 × 3% + 375 × 1.3% = 9.125 This would provide better performance, but may complicate the design of the processor. This could lead to: more complex cache coherency, increased cycle time, larger and more expensive chips.
b.	Base CPI: 2.0 Memory miss cycles: 100 clock cycles 1. Total CPI = base CPI + memory miss cycles × global miss rate w/2nd level direct-mapped cache + 2nd level direct-mapped speed × 1st level cache miss rate 2. Total CPI = base CPI + memory miss cycles × global miss rate w/3rd level direct-mapped cache + 2nd level direct-mapped speed × 1st level cache miss rate + 3rd level direct-mapped speed × 2nd level cache miss rate 1. Total CPI (using 2nd level direct-mapped cache): 2.0 + 100 × 0.04 + 10 × 0.04 = 6.4 2. Total CPI (using 3rd level direct-mapped cache): 2.0 + 100 × 0.013 + 10 × 0.04 + 50 × 0.04 = 5.7 This would provide better performance, but may complicate the design of the processor. This could lead to: more complex cache coherency, increased cycle time, larger and more expensive chips.

5.8.6

a.	Base CPI: 2.0 Memory miss cycles: 125 cycles/(1/3) ns/clock = 375 clock cycles Total CPI: 2.0 + 50 × 5% + 375 × (4% – 0.7% × n) = 14/10 n = 3 => 2 MB L2 cache to match DM n = 4 => 2.5 MB L2 cache to match 2-way
b.	Base CPI: 2.0 Memory miss cycles: 100 clock cycles 1) Total CPI (using 2nd level direct-mapped cache): 2.0 + 100 × 0.04 + 10 × 0.04 = 6.4 2) Total CPI (using 2nd level 8-way set assoc cache): 2.0 + 100 × 0.016 + 20 × 0.04 = 4.4 3) Total CPI = base CPI + cache access time × 1st level cache miss rate + memory miss cycles × (global miss rate – 0.7% × n) where n = further unit blocks of 512 KB cache size beyond base 512 KB cache 4) Total CPI: 2.0 + 50 × [0.04] + [100] × (0.04 – 0.007 × n) for n = 0, CPI: 8 for n = 1, CPI: 7.3 for n = 2, CPI: 6.6 for n = 3, CPI: 5.9 for n = 4, CPI: 5.7 for n = 5, CPI: 5.0 Hence, to match 2nd level direct-mapped cache CPI, n = 2 or 1.5 MB L2 cache, and to match 2nd level 8-way set assoc cache CPI, n = 5 or 3 MB L2 cache

Solution 5.9

Instructors can change the disk latency, transfer rate and optimal page size for more variants. Refer to Jim Gray's paper on the five-minute rule ten years later.

5.9.1 32 KB.

5.9.2 Still 32 KB.

5.9.3 64 KB. Because the disk bandwidth grows much faster than seek latency, future paging cost will be more close to constant, thus favoring larger pages.

5.9.4 1987/1997/2007: 205/267/308 seconds. (or roughly five minutes)

5.9.5 1987/1997/2007: 51/533/4935 seconds. (or 10 times longer for every 10 years).

5.9.6 (1) DRAM cost/MB scaling trend dramatically slows down; or (2) disk $/access/sec dramatically increase. (3) is more likely to happen due to the emerging flash technology.

Solution 5.10

5.10.1

a.

Virtual page number: Address >> 12 bits
H: Hit in TLB, M: Miss in TLB hit in page table, PF: Page Fault

0, 7, 3, 3, 1, 1, 2 (M, H, M, H, PF, H, PF)

TLB

Valid	Tag	Physical Page Number
1	3	6
1	7	4
1	1	13
1	2	14

Page table

Valid	Physical page or in disk
1	5
1	13
1	14
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	12

b.

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}
2 4EC, Page Fault, disk => physical page D, (=> TLB slot 3)
7 8F4, Hit in TLB
4 AC0, Miss in TLB, (=> TLB slot 0)
B 5A6, Miss in TLB, (=> TLB slot 2)
9 4DE, Page Fault, disk => physical page E, (=> TLB slot 3)
4 10D, Hit in TLB
B D60 Hit in TLB

TLB

Valid	Tag	Physical Page
1	4	9
1	7	4
1	B	C
1	9	E

Page table

Valid	Physical page
1	5
0	disk
1	D
1	6
1	9
1	B
0	disk
1	4
0	disk
1	E
1	3
1	C

5.10.2

a.

Virtual page number: Address >> 14 bits
H: Hit in TLB, M: Miss in TLB hit in page table, PF: Page Fault

0, 3, 1, 1, 0, 0, 1 (M, H, PF, H, H, H, H)

TLB

Valid	Tag	Physical Page Number
1	1	13
1	7	4
1	3	6
1	0	5

Page table

Valid	Physical page or in disk
1	5
1	13
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

Larger page sizes allow for more addresses to be stored in a single page, potentially decreasing the amount of pages that must be brought in from disk and increasing the coverage of the TLB. However, if a program uses addresses in a sparse fashion (for example randomly accessing a large matrix), then there will be an extra penalty from transferring larger pages compared to smaller pages.

b.

Binary address: (all hexadecimal), {bits 15–14 virtual page, 13–0 page offset}
0 24EC, Miss in TLB, (=> TLB slot 3)
1 38F4, Page Fault, disk => physical page D, (=> TLB slot 1)
1 0AC0, Hit in TLB
2 35A6, Page Fault, disk => physical page E, (=> TLB slot 2)
2 14DE, Hit in TLB
1 010D, Hit in TLB
2 3D60, Hit in TLB

TLB

Valid	Tag	Physical Page
1	B	C
1	1	D
1	2	E
1	0	5

Page table

Valid	Physical page
1	5
1	D
1	E
1	6
1	9
1	B
0	disk
1	4
0	disk
0	disk
1	3
1	C

Larger page sizes allow for more addresses to be stored in a single page, potentially decreasing the amount of pages that must be brought in from disk and increasing the coverage of the TLB. However, if a program uses addresses in a sparse fashion (for example randomly accessing a large matrix), then there will be an extra penalty from transferring larger pages compared to smaller pages.

5.10.3

a.

Virtual page number: Address >> 12 bits

0, 7, 3, 3, 1, 1, 2 => 0, 111, 011, 011, 001, 001, 010

2 way set associative:
Tag: VPN >> 1 bit

TLB

Valid	Tag	PPN	Valid	Tag	PPN
1	0	5	1	01	14
1	0	13	1	01	6

Direct-mapped:
Tag: VPN >> 2 bits

TLB

Valid	Tag	Physical Page Number
1	0	5
1	0	13
1	0	14
1	0	6

The TLB is important to avoiding paying high access times to memory in order to translate virtual addresses to physical addresses. If memory accesses are frequent, then the TLB will become even more important. Without a TLB, the page table would have to be referenced upon every access using a virtual addresses, causing a significant slowdown.

b.

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}

2 4EC, Page Fault, disk => physical page D, (=> TLB set 0, slot 1)
7 8F4, Miss in TLB, (=> TLB set 1, slot 1)
4 AC0, Miss in TLB, (=> TLB set 0, slot 0)
B 5A6, Miss in TLB, (=> TLB set 1, slot 0)
9 4DE, Page Fault, disk => physical page E, (=> TLB set 1, slot 1)
4 10D, Hit in TLB
B D60 Hit in TLB

2-way set associative TLB {bits 15–12 virtual page => bits 15–13 tag, bits 12 set}
(note: time stamps according to at start physical page numbers)

Valid	Tag(/Time)	Physical Page
1	4 /4	9
1	2 /2	D
1	B /5	C
1	9 /4	E

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}
2 4EC, Page Fault, disk => physical page D, (=> TLB slot 2)
7 8F4, Hit in TLB
4 AC0, Miss in TLB, (=> TLB slot 0)
B 5A6, Miss in TLB, (=> TLB slot 3)
9 4DE, Page Fault, disk => physical page F, (=> TLB slot 1)
4 10D, Hit in TLB
B D60 Hit in TLB

direct-mapped TLB {bits 15–12 virtual page => bits 13–12 TLB slot}

Valid	Tag	Physical Page
1	4	9
1	9	F
1	2	D
1	B	C

The TLB is important to avoiding paying high access times to memory in order to translate virtual addresses to physical addresses. If memory accesses are frequent, then the TLB will become even more important. Without a TLB, the page table would have to be referenced upon every access using a virtual addresses, causing a significant slowdown.

5.10.4

a.	4 KB page = 12 offset bits, 20 page number bits 2^{20} = 1 M page table entries 1 M entries × 4 bytes/entry = ~4 MB (2^{22} bytes) page table per application 2^{22} bytes × 5 apps = 20.97 MB total
b.	virtual address size of 64 bits 16 KB (2^{14}) page size, 8 (2^{3}) bytes per page table entry 64 – 14 = 40 bits or 2^{40} page table entries with 8 bytes per entry, yields total of 2^{43} bytes for each page table Total for 5 applications = 5 × 2^{43} bytes

5.10.5

a.	4 KB page = 12 offset bits, 20 page number bits 256 entries (8 bits) for first level => 12 bits => 4096 entries per second level Minimum: 128 first level entries used per app 128 entries × 4096 entries per second level = ~524K (2^{19}) entries ~524K × 4 bytes/entry = ~2 MB (2^{21}) second level page table per app 128 entries × 6 bytes/entry = 768 bytes first level page table per app ~10 MB total for all 5 apps Maximum: 256 first level entries used per app 256 entries × 4096 entries per second level = ~1M (2^{20}) entries ~1M × 4 bytes/entry = ~4 MB (2^{22}) second level page table per app 256 entries × 6 bytes/entry = 1536 bytes first level page table per app ~20.98 MB total for all 5 apps
b.	virtual address size of 64 bits 64 – 14 = 40 bits or 2^{40} page table entries 256 (2^{8}) entries in main table at 8 (2^{3}) bytes per page table entry (6 rounded up to nearest power of 2) total of 2^{11} bytes or 2 KB for main page table 40 – 8 = 32 bits or 2^{32} page table entries for 2nd level table with 8 bytes per entry, yields total of 2^{35} bytes for each page table Total for 5 applications = 5 × (2 KB + 2^{35} bytes) [maximum, with minimum as half this figure]

5.10.6

a.	16 KB Direct-Mapped cache 2 words per blocks => 8 bytes/block => 3 bits block offset 16 KB/8 bytes/block => 2K sets => 11 bits for indexing With a 4 KB page, the lower 12 bits are available for use for indexing prior to translation from VA to PA. However, a 16 KB Direct-Mapped cache needs the lower 14 bits to remain the same between VA to PA translation. Thus it is not possible to build this cache. If the cache's data size is to be increased, a higher associativity must be used. If the cache has 2 words per block, only 9 bits are available for indexing (thus 512 sets). To make a 16 KB cache, a 4-way associativity must be used.
b.	virtual address size of 64 bits 16 KB (2^{14}) page size, 8 (2^3) bytes per page table entry 16 KB direct-mapped cache 2 words or 8 (2^3) bytes per block, means 3 bits for cache block offset 16 KB/8 bytes per block = 2K sets or 11 bits for indexing With a 16 KB page, the lower 14 bits are available for use for indexing prior to translation from virtual to physical. Considering, a 16 KB direct-mapped cache requires the lower 14 bits to remain the same between translation. Hence, it is possible to build this cache.

Solution 5.11

5.11.1

a.	virtual address 32 bits, physical memory 4 GB page size 8 KB or 13 bits, page table entry 4 bytes or 2 bits #PTE = 32 – 13 = 19 bits or 512K entries PT physical memory = 512K × 4 bytes = 2 MB
b.	virtual address 64 bits, physical memory 16 GB page size 4 KB or 12 bits, page table entry 8 bytes or 3 bits #PTE = 64 – 12 = 52 bits or 2^{52} entries PT physical memory = $2^{52} \times 2^3 = 2^{55}$ bytes

5.11.2

a.	virtual address 32 bits, physical memory 4 GB page size 8 KB or 13 bits, page table entry 4 bytes or 2 bits #PTE = 32 – 13 = 19 bits or 512K entries 8 KB page/4 byte PTE = 2^{11} pages indexed per page Hence with 2^{19} PTEs will need 2-level page table setup. Each address translation will require at least 2 physical memory accesses.
b.	virtual address 64 bits, physical memory 16 GB page size 4 KB or 12 bits, page table entry 8 bytes or 3 bits #PTE = 64 – 12 = 52 bits or 2^{52} entries 4 KB page/8 byte PTE = 2^9 pages indexed per page Hence with 2^{52} PTEs will need 6-level page table setup. Each address translation will require at least 6 physical memory accesses.

5.11.3

a.	Since there are only 4 GB physical DRAM, only 512K PTEs are really needed to store the page table. Common-case: no hash conflict, so one memory reference per address translation; worst case: almost 512K memory references are needed if hash table degrade into a link list.
b.	virtual address 64 bits, physical memory 16 GB page size 4 KB or 12 bits, page table entry 8 bytes or 3 bits #PTE = 64 – 12 = 52 bits or 2^{52} entries Since there are only 16 GB physical memory, only $2^{(34-12)}$ PTEs are really needed to store the page table. Common-case: no hash conflict, so one memory reference per address translation; Worst case: almost $2^{(34-12)}$ memory references are needed if hash table degrade into a link list.

5.11.4 TLB initialization, or process context switch.

5.11.5 TLB miss. When most missed TLB entry is cached in processor caches.

5.11.6 Write protection exception.

Solution 5.12

5.12.1

a.	0 hits
b.	2 hits

5.12.2

a.	3 hits
b.	3 hits

5.12.3

a.	3 hits or fewer
b.	3 hits or fewer

5.12.4 Any address sequence is fine so long as the number of hits are correct.

a.	3 hits
b.	3 hits

5.12.5 The best block to evict is the one that will cause the fewest misses in the future. Unfortunately, a cache controller cannot know the future! Our best alternative is to make a good prediction.

5.12.6 If you knew that an address had limited temporal locality and would conflict with another block in the cache, it could improve miss rate. On the other hand, you could worsen the miss rate by choosing poorly which addresses to cache.

Solution 5.13

5.13.1 Shadow page table: (1) VM creates page table, hypervisor updates shadow table; (2) nothing; (3) hypervisor intercepts page fault, creates new mapping, and invalidates the old mapping in TLB; (4) VM notifies the hypervisor to invalidate the process's TLB entries. Nested page table: (1) VM creates new page table, hypervisor adds new mappings in PA to MA table. (2) Hardware walks both page tables to translate VA to MA; (3) VM and hypervisor update their page tables, hypervisor invalidates stale TLB entries; (4) same as shadow page table.

5.13.2

Native: 4; NPT: 24 (instructors can change the levels of page table)

Native: L; NPT: $L \times (L + 2)$

5.13.3

Shadow page table: page fault rate.

NPT: TLB miss rate.

5.13.4

Shadow page table: 1.03

NPT: 1.04

5.13.5 Combining multiple page table updates

5.13.6 NPT caching (similar to TLB caching)

Solution 5.14

5.14.1

a.	CPI: 2.0 + (100/10,000 × (20 + 150)) = 3.7 CPI: 2.0 + (100/10,000 × (20 + 300)) = 5.2 CPI: 2.0 + (100/10,000 × (20 + 75)) = 2.95 To obtain a 10% performance degradation, we must solve: 1.1 × (2.0 + (100/10,000 × 20)) = 2.0 + (100/10,000 × (20 + n)) We find that n = 22 cycles
b.	CPI for the system with no accesses to I/O CPI: BaseCPI + ((priv OS access/10000) × (perf impact trap guestOS + perf impact trap VMM)) CPI: BaseCPI + ((priv OS access/10000) × (perf impact trap guestOS + 2 × perf impact trap VMM)) CPI: BaseCPI + ((priv OS access/10000) × (perf impact trap guestOS + 0.5 × perf impact trap VMM)) CPI: 1.5 + (110/10,000 × (25 + 160)) = 3.535 CPI: 1.5 + (110/10,000 × (25 + 320)) = 5.295 CPI: 1.5 + (110/10,000 × (25 + 80)) = 2.655 To obtain a 10% performance degradation, we must solve: 1.1 × (BaseCPI + (priv OS access/10000 × perf impact trap guestOS)) = BaseCPI + ((priv OS access/10000) × (perf impact trap guestOS + n)) 1.1 × (1.5 + (110/10000 × 25)) = (1.5 + (110/10000 × (25 + n))) 1.1 × 1.775 = 1.5 + (0.011 × (25 + n)) 1.9525 – 1.5 = 0.011 × (25 + n) 0.4525/0.011 = 25 + n we find that n = 20 cycles is longest possible penalty to trap to the VMM

5.14.2

a.	CPI, non virtualized = 2.0 + 80/10,000 × 20 + 20/10,000 × 1000 = 2.0 + 0.16 + 2.0 = 4.16 CPI, virtualized = 2.0 + 80/10,000 × (20 + 150) + 20/10,000 × (1000 + 150) = 2.0 + 1.36 + 2.3 = 5.66 I/O bound applications have a smaller impact from virtualization because, comparatively, a much longer time is spent on waiting for the I/O accesses to complete.
b.	CPI (non virtualised): BaseCPI + (priv OS access-I/O accesses)/10000 × perf impact trap guestOS) + I/O accesses/10000 × I/O access time CPI (virtualised): BaseCPI + (priv OS access-I/O accesses)/10000 × (perf impact trap guestOS + perf impact trap VMM)) + (I/O accesses/10000 × (I/O access time + perf impact trap VMM)) CPI (non virtualised): 1.5 + (110 – 10)/10000 × 25 + 10/10000 × 1000 = 1.5 + 0.225 + 1 = 2.725 CPI (virtualised): 1.5 + (110 – 10)/10000 × (25 + 160) + 10/10000 × (1000 + 160) = 1.5 + 1.665 + 1.16 = 4.325 I/O bound applications have a smaller impact from virtualization because, comparatively, a much longer time is spent on waiting for the I/O accesses to complete.

5.14.3 Virtual memory aims to provide each application with the illusion of the entire address space of the machine. Virtual machines aims to provide each operating system with the illusion of having the entire machine to its disposal. Thus they both serve very similar goals, and offer benefits such as increased security. Virtual memory can allow for many applications running in the same memory space to not have to manage keeping their memory separate.

5.14.4 Emulating a different ISA requires specific handling of that ISA's API. Each ISA has specific behaviors that will happen upon instruction execution, interrupts, trapping to kernel mode, etc. that therefore must be emulated. This can require many more instructions to be executed to emulate each instruction than was originally necessary in the target ISA. This can cause a large performance impact and make it difficult to properly communicate with external devices. An emulated system can potentially run faster than on its native ISA if the emulated code can be dynamically examined and optimized. For example, if the underlying machine's ISA has a single instruction that can handle the execution of several of the emulated system's instructions, then potentially the number of instructions executed can be reduced. This is similar to the recent Intel processors that do micro-op fusion, allowing several instructions to be handled by fewer instructions.

Solution 5.15

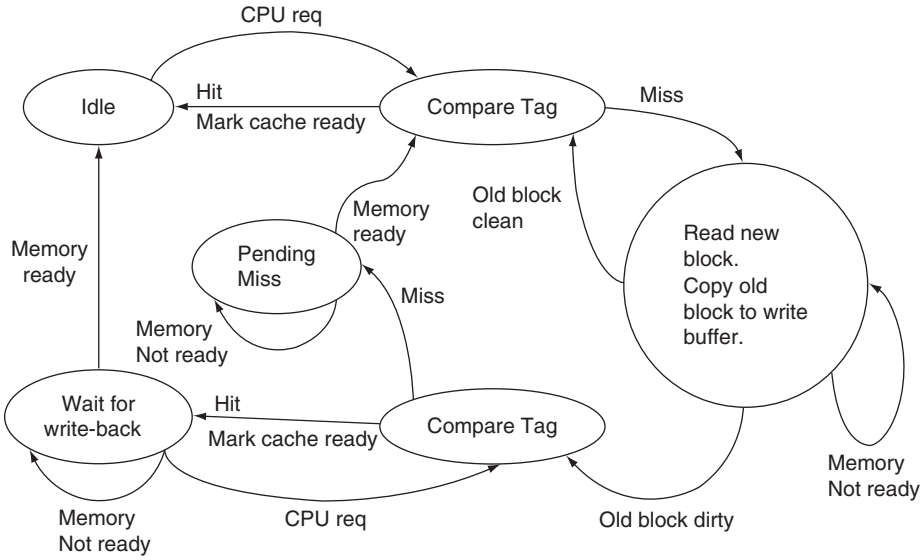
5.15.1 The cache should be able to satisfy the request since it is otherwise idle when the write buffer is writing back to memory. If the cache is not able to satisfy hits while writing back from the write buffer, the cache will perform little or no better than the cache without the write buffer, since requests will still be serialized behind writebacks.

5.15.2 Unfortunately, the cache will have to wait until the writeback is complete since the memory channel is occupied. Once the memory channel is free, the cache is able to issue the read request to satisfy the miss.

5.15.3 Correct solutions should exhibit the following features:

1. The memory read should come before memory writes.
2. The cache should signal "Ready" to the processor before completing the write.

Example (simpler solutions exist, the state machine is somewhat underspecified in the chapter):



Solution 5.16

5.16.1

a.	Coherent: [2,4], [3,4], [2,5], [3,5]; non-coherent: [1, 1];
b.	P1: X[0] ++; X[1] += 3; P2: X[0] = 5; X[1] = 2; coherent: [5,5], [6,5], [5,2], [6,2] non-coherent: [1, 2];

5.16.2

a.	P1: X[0] ++; X[1] = 4; P2: X[0] = 2; X[1] ++; operation sequence P1: read X[0], X[0]++, write X[0]; X[1] = 4, write X[1]; P2: X[0] = 2, write X[0]; read X[1], X[1]++, write X[1];
b.	P1: X[0] ++; X[1] += 3; P2: X[0] = 5; X[1] = 2; operation sequence P1: read X[0], X[0]++, write X[0]; read X[1], X[1] += 3, write X[1]; P2: X[0] = 5, write X[0]; X[1] = 2, write X[1];

5.16.3

a.	Best-case: 1; worst-case: 6
b.	best case: 1; worst case: 6;

5.16.4

a.	Consistent: [0,0], [0,1], [1,2], [2,1], [2,2], [2,3], [3,3];
b.	P1: A = 1; B += 2; A ++; B = 4; P2: C = B; D = A; Consistent [C,D]: [0,0], [0,1], [2,1], [2,2], [4,2]

5.16.5

a.	Inconsistent: [2,0], [3,0], [3,1], [3,2];
b.	Inconsistent [C,D]: [4,0]

5.16.6 Write-through, non write allocate simplifies the most.

Solution 5.17

5.17.1

a.	Shared L2 is better for benchmark A; private L2 is better for benchmark B. Compare L1 miss latencies for various configurations.
b.	Benchmark A/B: private miss rate × memory hit latency + (1 – private miss rate) × private cache hit latency Benchmark A/B: shared miss rate × memory hit latency + (1 – shared miss rate) × shared cache hit latency Benchmark A private: 0.003 × 120 + 0.997 × 8 = 0.36 + 7.976 = 8.336 Benchmark B private: 0.0006 × 120 + 0.9994 × 8 = 0.072 + 7.9952 = 8.0672 Benchmark A shared: 0.0012 × 120 + 0.9988 × 20 = 0.144 + 19.976 = 20.12 Benchmark B shared: 0.0003 × 120 + 0.9997 × 20 = 0.036 + 19.994 = 20.03

5.17.2

a.	When shared L2 latency doubles, both benchmarks prefer private L2. When memory latency doubles, benchmark A prefers shared cache while benchmark B prefers private L2.
b.	Shared cache latency doubling: Benchmark A/B: shared miss rate × memory hit latency + (1 – shared miss rate) × 2 × shared cache hit latency Benchmark A shared: 0.0012 × 120 + 0.9988 × 2 × 20 = 0.144 + 2 × 19.976 = 40.096 Benchmark B shared: 0.0003 × 120 + 0.9997 × 2 × 20 = 0.036 + 2 × 19.994 = 40.024 Off chip memory latency doubling: Benchmark A/B: private miss rate × 2 × memory hit latency + (1 – private miss rate) × private cache hit latency Benchmark A/B: shared miss rate × 2 × memory hit latency + (1 – shared miss rate) × shared cache hit latency

5.17.3

a.	Shared L2: typically good for multithreaded benchmarks when significant amount of shared data; good for applications need more than private cache capacity. Private L2: good for applications whose working set can fit, also good for isolating negative interferences between multiprogrammed workloads.
b.	Shared L2: typically good for multithreaded benchmarks when significant amount of shared data; good for applications need more than private cache capacity. Private L2: good for applications whose working set can fit, also good for isolating negative interferences between multiprogrammed workloads.

5.17.4

a.	Over shared cache: benchmark A 4.7%, benchmark B 1.3% Over private cache: benchmark A 11.6%, benchmark B 8.1%
b.	Performance improvement over shared cache: benchmark A 4.7%, benchmark B 1.3% Performance improvement over private cache: benchmark A 11.6%, benchmark B 8.1%

5.17.5

a.	For private L2, 4X bandwidth. For shared L2, cannot determine because the aggregate miss rate is not the sum of per-workload miss rates.
b.	For private L2, 4X bandwidth. For shared L2, cannot determine because the aggregate miss rate is not the sum of per-workload miss rates.

5.17.6

Processor: out-of-order execution, larger load/store queue, multiple hardware threads;

Caches: more miss status handling registers (MSHR)

Memory: memory controller to support multiple outstanding memory requests

Solution 5.18

5.18.1

a.	`srcIP` field. 1 miss per entry.
b.	`refTime` and `status` fields. 1 miss per entry.

5.18.2

a.	Split the `srcIP` field into a separate array.
b.	Group the `refTime` and `status` fields into a separate array.

5.18.3

a.	`topK_sourceIP (int hour);` Group the `srcIP` and `refTime` fields into a separate array.
b.	`topK_sourceIP (int hour);` Group `srcIP`, `refTime` and `status` together.

5.18.4

a.		**cold**	**capacity**	**conflict (8-way)**	**conflict (4-way)**	**conflict (2-way)**	**conflict (direct map)**
	apsi	0.00%	0.746%	0.505%	0.006%	0.142%	0.740%
	facerec	0.00%	0.649%	0.144%	–0.001%	0.001%	0.083%

b.		**cold**	**capacity**	**conflict (8-way)**	**conflict (4-way)**	**conflict (2-way)**	**conflict (direct map)**
	perlbmk	0.00%	0.0011%	0.0016%	0.0017%	0.0024%	0.0061%
	ammp	0.00%	0.0166%	0.0172%	0.0175%	0.0180%	0.0196%

5.18.5

a.	3 way for shared L1 cache; 4-way for shared L2 cache.
b.	8-way for shared L1 cache of 64 KB; 8-way for shared L2 cache of 1 MB and direct-mapped for L1 cache of 64 KB;

5.18.6

a.	apsi. 512 KB 2-way LRU has higher miss rate than direct-mapped cache.
b.	apsi/mesa/ammp/mcf all have such examples.

Example cache: 4-block caches, direct-mapped versus 2-way LRU.

Reference stream (blocks): 1 2 2 6 1.