Chapter 1 Solutions

Case Study 1: Chip Fabrication Cost

1.1 a. $\text{Yield} = 1/(1 + (0.04 \times 2))^{14} = 0.34$

b. It is fabricated in a larger technology, which is an older plant. As plants age, their process gets tuned, and the defect rate decreases.

1.2 a. Phoenix:

$$\text{Dies per wafer} = \left(\pi \times (45/2)^2\right)/2 - (\pi \times 45)/\text{sqrt}(2 \times 2) = 795 - 70.7 = 724.5 = 724$$

$$\text{Yield} = 1/(1 + (0.04 \times 2))^{14} = 0.340$$

$$\text{Profit} = 724 \times 0.34 \times 30 = \$7384.80$$

b. Red Dragon:

$$\text{Dies per wafer} = \left(\pi \times (45/2)^2\right)/2 - (\pi \times 45)/\text{sqrt}(2 \times 1.2) = 1325 - 91.25 = 1234$$

$$\text{Yield} = 1/(1 + (0.04 \times 1.2))^{14} = 0.519$$

$$\text{Profit} = 1234 \times 0.519 \times 15 = \$9601.71$$

c. Phoenix chips: 25,000/724 = 34.5 wafers needed
Red Dragon chips: 50,000/1234 = 40.5 wafers needed

Therefore, the most lucrative split is 40 Red Dragon wafers, 30 Phoenix wafers.

1.3 a. Defect-free single core = Yield = $1/(1 + (0.04 \times 0.25))^{14} = 0.87$
Equation for the probability that N are defect free on a chip:
$\#\text{combinations} \times (0.87)^N \times (1 - 0.87)^{8-N}$

# defect-free	# combinations	Probability
8	1	0.32821167
7	8	0.39234499
6	28	0.20519192
5	56	0.06132172
4	70	0.01145377
3	56	0.00136919
2	28	0.0001023
1	8	4.3673E-06
0	1	8.1573E-08

Yield for Phoenix^4: $(0.39 + 0.21 + 0.06 + 0.01) = 0.57$
Yield for Phoenix^2: $(0.001 + 0.0001) = 0.0011$
Yield for Phoenix^1: 0.000004

b. It would be worthwhile to sell Phoenix^4. However, the other two have such a low probability of occurring that it is not worth selling them.

c.

$$\$20 = \frac{\text{Wafer size}}{\text{odd dpw} \times 0.28}$$

Step 1: Determine how many Phoenix4 chips are produced for every Phoenix8 chip.

There are 57/33 Phoenix4 chips for every Phoenix8 chip = 1.73

$$\$30 + 1.73 \times \$25 = \$73.25$$

Case Study 2: Power Consumption in Computer Systems

1.4 a. Energy: 1/8. Power: Unchanged.

b. Energy: $\text{Energy}_{\text{new}}/\text{Energy}_{\text{old}} = (\text{Voltage} \times 1/8)^2/\text{Voltage}^2 = 0.156$
Power: $\text{Power}_{\text{new}}/\text{Power}_{\text{old}} = 0.156 \times (\text{Frequency} \times 1/8)/\text{Frequency} = 0.00195$

c. Energy: $\text{Energy}_{\text{new}}/\text{Energy}_{\text{old}} = (\text{Voltage} \times 0.5)^2/\text{Voltage}^2 = 0.25$
Power: $\text{Power}_{\text{new}}/\text{Power}_{\text{old}} = 0.25 \times (\text{Frequency} \times 1/8)/\text{Frequency} = 0.0313$

d. 1 core = 25% of the original power, running for 25% of the time.

$$0.25 \times 0.25 + (0.25 \times 0.2) \times 0.75 = 0.0625 + 0.0375 = 0.1$$

1.5 a. Amdahl's law: 1/(0.8/4 + 0.2) = 1/(0.2 + 0.2) = 1/0.4 = 2.5

b. 4 cores, each at 1/(2.5) the frequency and voltage
Energy: $\text{Energy}_{\text{quad}}/\text{Energy}_{\text{single}} = 4 \times (\text{Voltage} \times 1/(2.5))^2/\text{Voltage}^2 = 0.64$
Power: $\text{Power}_{\text{new}}/\text{Power}_{\text{old}} = 0.64 \times (\text{Frequency} \times 1/(2.5))/\text{Frequency} = 0.256$

c. 2 cores + 2 ASICs vs. 4 cores

$$(2 + (0.2 \times 2))/4 = (2.4)/4 = 0.6$$

1.6 a. Workload A speedup: 225,000/13,461 = 16.7
Workload B speedup: 280,000/36,465 = 7.7
1/(0.7/16.7 + 0.3/7.7)

b. General-purpose: 0.70 × 0.42 + 0.30 = 0.594
GPU: 0.70 × 0.37 + 0.30 = 0.559
TPU: 0.70 × 0.80 + 0.30 = 0.886

c. General-purpose: 159 W + (455 W − 159 W) × 0.594 = 335 W
GPU: 357 W + (991 W − 357 W) × 0.559 = 711 W
TPU: 290 W + (384 W − 290 W) × 0.86 = 371 W

d.

Speedup	A	B	C
GPU	2.46	2.76	1.25
TPU	41.0	21.2	0.167
% Time	0.4	0.1	0.5

GPU: $1/(0.4/2.46+0.1/2.76+0.5/1.25)=1.67$
TPU: $1/(0.4/41+0.1/21.2+0.5/0.17)=0.33$

e. General-purpose: $14{,}000/504 = 27.8 \geq 28$
GPU: $14{,}000/1838=7.62\geq 8$
TPU: $14{,}000/861=16.3\geq 17$

d. General-purpose: $2200/504=4.37\geq 4$, $14{,}000/(4 \times 504)=6.74\geq 7$
GPU: $2200/1838=1.2\geq 1$, $14{,}000/(1 \times 1838)=7.62\geq 8$
TPU: $2200/861=2.56\geq 2$, $14{,}000/(2 \times 861)=8.13\geq 9$

Exercises

1.7 a. Somewhere between 1.4^{10} and 1.55^{10}, or $28.9-80x$

b. 6043 in 2003, 52% growth rate per year for 12 years is 60,500,000 (rounded)

c. 24,129 in 2010, 22% growth rate per year for 15 years is 1,920,000 (rounded)

d. Multiple cores on a chip rather than faster single-core performance

e. $2=x^4$, $x=1.032$, 3.2% growth

1.8 a. 50%

b. Energy: $\text{Energy}_{\text{new}}/\text{Energy}_{\text{old}}=(\text{Voltage} \times 1/2)^2/\text{Voltage}^2=0.25$

1.9 a. 60%

b. $0.4+0.6\times 0.2=0.58$, which reduces the energy to 58% of the original energy

c. $\text{newPower}/\text{oldPower}=½\text{Capacitance}\times(\text{Voltage}\times 0.8)^2\times(\text{Frequency}\times 0.6)/½\ \text{Capacitance}\times\text{Voltage}\times\text{Frequency}=0.8^2\times 0.6=0.256$ of the original power.

d. $0.4+0.3\times 2=0.46$, which reduces the energy to 46% of the original energy

1.10 a. $10^9/100=10^7$

b. $10^7/10^7+24=1$

c. [need solution]

1.11 a. $35/10{,}000\times 3333=11.67$ days

b. There are several correct answers. One would be that, with the current system, one computer fails approximately every 5 min. 5 min is unlikely to be enough time to isolate the computer, swap it out, and get the computer back on line again. 10 min, however, is much more likely. In any case, it would greatly extend the amount of time before 1/3 of the computers have failed at once. Because the cost of downtime is so huge, being able to extend this is very valuable.

c. $\$90{,}000=(x+x+x+2x)/4$
$\$360{,}000=5x$
$\$72{,}000=x$
4th quarter $=\$144{,}000$/h

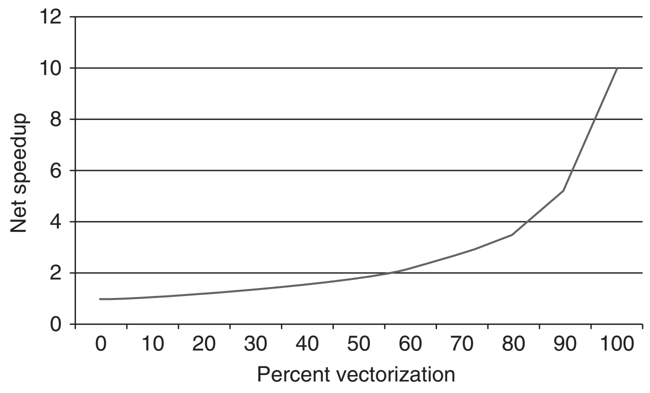


Figure S.1 Plot of the equation: $y = 100/((100 - x) + x/10)$.

1.12 a. See Figure S.1.

b. $2 = 1/((1 - x) + x/20)$
$10/19 = x = 52.6\%$

c. $(0.526/20)/(0.474 + 0.526/20) = 5.3\%$

d. Extra speedup with 2 units: $1/(0.1 + 0.9/2) = 1.82$. $1.82 \times 20 - 36.4$. Total speedup: 1.95. Extra speedup with 4 units: $1/(0.1 + 0.9/4) = 3.08$. $3.08 \times 20 - 61.5$. Total speedup: 1.97

1.13 a. old execution time = 0.5 new + 0.5 × 10 new = 5.5 new

b. In the original code, the unenhanced part is equal in time to the enhanced part (sped up by 10), therefore:
$(1 - x) = x/10$
$10 - 10x = x$
$10 = 11x$
$10/11 = x = 0.91$

1.14 a. $1/(0.8 + 0.20/2) = 1.11$

b. $1/(0.7 + 0.20/2 + 0.10 \times 3/2) = 1.05$

c. fp ops: $0.1/0.95 = 10.5\%$, cache: $0.15/0.95 = 15.8\%$

1.15 a. $1/(0.5 + 0.5/22) = 1.91$

b. $1/(0.1 + 0.90/22) = 7.10$

c. $41\% \times 22 = 9$. A runs on 9 cores. Speedup of A on 9 cores: $1/(0.5 + 0.5/9) = 1.8$ Overall speedup if 9 cores have 1.8 speedup, others none: $1/(0.6 + 0.4/1.8) = 1.22$

d. Calculate values for all processors like in c. Obtain: 1.8, 3, 1.82, 2.5, respectively.

e. $1/(0.41/1.8 + 0.27/3 + 0.18/1.82 + 0.14/2.5) = 2.12$

1.16 a. $1/(0.2 + 0.8/N)$

b. $1/(0.2 + 8 \times 0.005 + 0.8/8) = 2.94$

c. $1/(0.2 + 3 \times 0.005 + 0.8/8) = 3.17$

d. $1/(.2 + \log N \times 0.005 + 0.8/N)$

e. $d/dN\ (1/((1 - P) + \log N \times 0.005 + P/N) = 0)$

Chapter 2 Solutions

Case Study 1: Optimizing Cache Performance via Advanced Techniques

2.1 a. Each element is 8B. Because a 64B cacheline has 8 elements, and each column access will result in fetching a new line for the nonideal matrix, we need a minimum of 8×8 (64 elements) for each matrix. Hence, the minimum cache size is $128 \times 8B = 1$ KB.

b. The blocked version only has to fetch each input and output element once. The unblocked version will have one cache miss for every $64B/8B = 8$ row elements. Each column requires $64B \times 256$ of storage, or 16 KB. Thus, column elements will be replaced in the cache before they can be used again. Hence, the unblocked version will have 9 misses (1 row and 8 columns) for every 2 in the blocked version.

c.
```
for (i = 0; i < 256; i = i + B) {
     for (j = 0; j < 256; j = j + B) {
         for (m = 0; m < B; m++) {
             for (n = 0; n < B; n++) {
                 output[j + n][i + m] = input[i + m][j + n];
             }
         }
     }
  }
```

d. 2-way set associative. In a direct-mapped cache, the blocks could be allocated so that they map to overlapping regions in the cache.

e. You should be able to determine the level-1 cache size by varying the block size. The ratio of the blocked and unblocked program speeds for arrays that do not fit in the cache in comparison to blocks that do is a function of the cache block size, whether the machine has out-of-order issue, and the bandwidth provided by the level-2 cache. You may have discrepancies if your machine has a write-through level-1 cache and the write buffer becomes a limiter of performance.

2.2 Because the unblocked version is too large to fit in the cache, processing eight 8B elements requires fetching one 64B row cache block and 8 column cache blocks. Because each iteration requires 2 cycles without misses, prefetches can be initiated every 2 cycles, and the number of prefetches per iteration is more than one, the memory system will be completely saturated with prefetches. Because the latency of a prefetch is 16 cycles, and one will start every 2 cycles, $16/2 = 8$ will be outstanding at a time.

2.3 Open hands-on exercise, no fixed solution

Case Study 2: Putting it all Together: Highly Parallel Memory Systems

2.4 a. The second-level cache is 1 MB and has a 128B block size.

b. The miss penalty of the second-level cache is approximately 105 ns.

c. The second-level cache is 8-way set associative.

d. The main memory is 512 MB.

e. Walking through pages with a 16B stride takes 946 ns per reference. With 250 such references per page, this works out to approximately 240 ms per page.

2.5 a. Hint: This is visible in the graph above shown as a slight increase in L2 miss service time for large data sets, and is 4 KB for the graph above.

b. Hint: Take independent strides by the page size and look for increases in latency not attributable to cache sizes. This may be hard to discern if the amount of memory mapped by the TLB is almost the same as the size as a cache level.

c. Hint: This is visible in the graph above shown as a slight increase in L2 miss service time for large data sets, and is 15 ns in the graph above.

d. Hint: Take independent strides that are multiples of the page size to see if the TLB if fully-associative or set-associative. This may be hard to discern if the amount of memory mapped by the TLB is almost the same as the size as a cache level.

2.6 a. Hint: Look at the speed of programs that easily fit in the top-level cache as a function of the number of threads.

b. Hint: Compare the performance of independent references as a function of their placement in memory.

2.7 Open hands-on exercise, no fixed solution

Case Study 3: Studying the Impact of Various Memory System Organizations

2.8 On a row buffer miss, the time taken to retrieve a 64-byte block of data equals tRP + tRCD + CL + transfer time = 13 + 13 + 13 + 4 = 43 ns.

2.9 On a row buffer hit, the time taken to retrieve a 64-byte block of data equals CL + transfer time = 13 + 4 = 17 ns.

2.10 Each row buffer miss involves steps (i)–(iii) in the problem description. Before the Precharge of a new read operation can begin, the Precharge, Activate, and CAS latencies of the previous read operation must elapse. In other words, back-to-back read operations are separated by time tRP + tRCD + CL = 39 ns. Of this time, the memory channel is only occupied for 4 ns. Therefore, the channel utilization is 4/39 = 10.3%.

2.11 When a read operation is initiated in a bank, 39 cycles later, the channel is busy for 4 cycles. To keep the channel busy all the time, the memory controller should initiate read operations in a bank every 4 cycles. Because successive read operations to a single bank must be separated by 39 cycles, over that period, the memory controller should initiate read operations to unique banks. Therefore, at least 10 banks are required if the memory controller hopes to achieve 100% channel utilization. It can then initiate a new read operation to each of these 10 banks every 4 cycles, and then repeat the process. Another way to arrive at this answer is to divide 100 by the 10.3% utilization calculated previously.

2.12 We can assume that accesses to a bank alternate between row buffer hits and misses. The sequence of operations is as follows: Precharge begins at time 0; Activate is performed at time 13 ns; CAS is performed at time 26 ns; a second CAS (the row buffer hit) is performed at time 30 ns; a precharge is performed at time 43 ns; and so on. In the above sequence, we can initiate a row buffer miss and row buffer hit to a bank every 43 ns. So, the channel is busy for 8 out of every 43 ns, that is, a utilization of 18.6%. We would therefore need at least 6 banks to achieve 100% channel utilization.

2.13 With a single bank, the first request would be serviced after 43 ns. The second would be serviced 39 ns later, that is, at time 82 ns. The third and fourth would be serviced at times 121 and 160 ns. This gives us an average memory latency of $(43+82+121+160)/4=101.5$ ns. Note that waiting in the queue is a significant component of this average latency. If we had four banks, the first request would be serviced after 43 ns. Assuming that the four requests go to four different banks, the second, third, and fourth requests are serviced at times 47, 51, and 55 ns. This gives us an average memory latency of 49 ns. The average latency would be higher if two of the requests went to the same bank. Note that by offering even more banks, we would increase the probability that the four requests go to different banks.

2.14 As seen in the previous questions, by growing the number of banks, we can support higher channel utilization, thus offering higher bandwidth to an application. We have also seen that by having more banks, we can support higher parallelism and therefore lower queuing delays and lower average memory latency. Thus, even though the latency for a single access has not changed because of limitations from physics, by boosting parallelism with banks, we have been able to improve both average memory latency and bandwidth. This argues for a memory chip that is partitioned into as many banks as we can manage. However, each bank introduces overheads in terms of peripheral logic near the bank. To balance performance and density (cost), DRAM manufacturers have settled on a modest number of banks per chip—8 for DDR3 and 16 for DDR4.

2.15 When channel utilization is halved, the power reduces from 535 to 295 mW, a 45% reduction. At the 70% channel utilization, increasing the row buffer hit rate from 50% to 80% moves power from 535 to 373 mW, a 30% reduction.

2.16 The table is as follows (assuming default system config):

DRAM chips	4×16 1 Gb	8×8 1 Gb	16×4 1 Gb	4×16 2 Gb	8×8 2 Gb	16×4 2 Gb	4×16 4 Gb	8×8 4 Gb	16×4 4 Gb
Chip power	517 mW	320 mW	734 mW	758 mW	535 mW	493 mW	506 mW	360 mW	323 mW
Rank power	2.07 W	2.56 W	11.7 W	3.03 W	4.28 W	7.89 W	2.02 W	2.88 W	5.17 W
Rank capacity	4 Gb	8 Gb	16 Gb	8 Gb	16 Gb	32 Gb	16 Gb	32 Gb	64 Gb

Note that the 4 Gb DRAM chips are manufactured with a better technology, so it is better to use fewer and larger chips to construct a rank with given capacity, if we are trying to reduce power.

Exercises

2.17 a. The access time of the direct-mapped cache is 0.86 ns, while the 2-way and 4-way are 1.12 and 1.37 ns, respectively. This makes the relative access times 1.12/0.86 = 1.30 or 30% more for the 2-way and 1.37/0.86 = 1.59 or 59% more for the 4-way.

b. The access time of the 16 KB cache is 1.27 ns, while the 32 and 64 KB are 1.35 and 1.37 ns, respectively. This makes the relative access times 1.35/1.27 = 1.06 or 6% larger for the 32 KB and 1.37/1.27 = 1.078 or 8% larger for the 64 KB.

c. Avg. access time = hit% × hit time + miss% × miss penalty, miss% = misses per instruction/references per instruction = 2.2% (DM), 1.2% (2-way), 0.33% (4-way), 0.09% (8-way).

Direct mapped access time = 0.86 ns @ 0.5 ns cycle time = 2 cycles
2-way set associative = 1.12 ns @ 0.5 ns cycle time = 3 cycles
4-way set associative = 1.37 ns @ 0.83 ns cycle time = 2 cycles
8-way set associative = 2.03 ns @ 0.79 ns cycle time = 3 cycles
Miss penalty = (10/0.5) = 20 cycles for DM and 2-way; 10/0.83 = 13 cycles for 4-way; 10/0.79 = 13 cycles for 8-way.

Direct mapped—$(1 - 0.022) \times 2 + 0.022 \times (20) = 2.39$ 6 cycles $\Rightarrow 2.396 \times 0.5 = 1.2$ ns
2-way—$(1 - 0.012) \times 3 + 0.012 \times (20) = 3.2$ cycles $\Rightarrow 3.2 \times 0.5 = 1.6$ ns
4-way—$(1 - 0.0033) \times 2 + 0.0033 \times (13) =$
2.036 cycles $\Rightarrow 2.06 \times 0.83 = 1.69$ ns
8-way—$(1 - 0.0009) \times 3 + 0.0009 \times 13 = 3$ cycles $\Rightarrow 3 \times 0.79 = 2.37$ ns

Direct mapped cache is the best.

2.18 a. The average memory access time of the current (4-way 64 KB) cache is 1.69 ns. 64 KB direct mapped cache access time = 0.86 ns @ 0.5 ns cycle time = 2 cycles Way-predicted cache has cycle time and access time similar to direct mapped cache and miss rate similar to 4-way cache.

The AMAT of the way-predicted cache has three components: miss, hit with way prediction correct, and hit with way prediction mispredict: $0.0033 \times (20) + (0.80 \times 2 + (1 - 0.80) \times 3) \times (1 - 0.0033) = 2.26$ cycles = 1.13 ns.

b. The cycle time of the 64 KB 4-way cache is 0.83 ns, while the 64 KB direct-mapped cache can be accessed in 0.5 ns. This provides 0.83/0.5 = 1.66 or 66% faster cache access.

c. With 1 cycle way misprediction penalty, AMAT is 1.13 ns (as per part a), but with a 15 cycle misprediction penalty, the AMAT becomes: $0.0033 \times 20 + (0.80 \times 2 + (1 - 0.80) \times 15) \times (1 - 0.0033) = 4.65$ cycles or 2.3 ns.

d. The serial access is 2.4 ns/1.59 ns = 1.509 or 51% slower.

2.19 a. The access time is 1.12 ns, while the cycle time is 0.51 ns, which could be potentially pipelined as finely as $1.12/0.51 = 2.2$ pipestages.

b. The pipelined design (not including latch area and power) has an area of 1.19 mm^2 and energy per access of 0.16 nJ. The banked cache has an area of 1.36 mm^2 and energy per access of 0.13 nJ. The banked design uses slightly more area because it has more sense amps and other circuitry to support the two banks, while the pipelined design burns slightly more power because the memory arrays that are active are larger than in the banked case.

2.20 a. With critical word first, the miss service would require 120 cycles. Without critical word first, it would require 120 cycles for the first 16B and 16 cycles for each of the next 3 16B blocks, or $120 + (3 \times 16) = 168$ cycles.

b. It depends on the contribution to Average Memory Access Time (AMAT) of the level-1 and level-2 cache misses and the percent reduction in miss service times provided by critical word first and early restart. If the percentage reduction in miss service times provided by critical word first and early restart is roughly the same for both level-1 and level-2 miss service, then if level-1 misses contribute more to AMAT, critical word first would likely be more important for level-1 misses.

2.21 a. 16B, to match the level 2 data cache write path.

b. Assume merging write buffer entries are 16B wide. Because each store can write 8B, a merging write buffer entry would fill up in 2 cycles. The level-2 cache will take 4 cycles to write each entry. A nonmerging write buffer would take 4 cycles to write the 8B result of each store. This means the merging write buffer would be two times faster.

c. With blocking caches, the presence of misses effectively freezes progress made by the machine, so whether there are misses or not doesn't change the required number of write buffer entries. With nonblocking caches, writes can be processed from the write buffer during misses, which may mean fewer entries are needed.

2.22 In all three cases, the time to look up the L1 cache will be the same. What differs is the time spent servicing L1 misses. In case (a), that time $= 100$ (L1 misses) $\times$ 16 cycles $+ 10$ (L2 misses) $\times$ 200 cycles $= 3600$ cycles. In case (b), that time $= 100 \times 4 + 50 \times 16 + 10 \times 200 = 3200$ cycles. In case (c), that time $= 100 \times 2 + 80 \times 8 + 40 \times 16 + 10 \times 200 = 3480$ cycles. The best design is case (b) with a 3-level cache. Going to a 2-level cache can result in many long L2 accesses (1600 cycles looking up L2). Going to a 4-level cache can result in many futile look-ups in each level of the hierarchy.

2.23 Program B enjoys a higher marginal utility from each additional way. Therefore, if the objective is to minimize overall MPKI, program B should be assigned as many ways as possible. By assigning 15 ways to program B and 1 way to program A, we achieve the minimum aggregate MPKI of $50 - (14 \times 2) + 100 = 122$.

2.24 Let's first assume an idealized perfect L1 cache. A 1000-instruction program would finish in 1000 cycles, that is, 1000 ns. The power consumption would be 1 W for the core and L1, plus 0.5 W of memory background power. The energy consumed would be $1.5\ \text{W} \times 1000\ \text{ns} = 1.5\ \mu\text{J}$.

Next, consider a PMD that has no L2 cache. A 1000-instruction program would finish in $1000 + 100$ (MPKI) $\times\ 100$ ns (latency per memory access) $= 11{,}000$ ns. The energy consumed would be $11{,}000$ ns $\times\ 1.5$ W (core, L1, background memory power) $+\ 100$ (memory accesses) $\times\ 35$ nJ (energy per memory access) $= 16{,}500 + 3500\ \text{nJ} = 20\ \mu\text{J}$.

For the PMD with a 256 KB L2, the 1000-instruction program would finish in $1000 + 100$ (L1 MPKI) $\times\ 10$ ns (L2 latency) $+\ 20$ (L2 MPKI) $\times\ 100$ ns (memory latency) $= 4000$ ns. Energy $= 1.7$ W (core, L1, L2 background, memory background power) $\times\ 4000$ ns $+\ 100$ (L2 accesses) $\times\ 0.5$ nJ (energy per L2 access) $+\ 20$ (memory accesses) $\times\ 35$ nJ $= 6800 + 50 + 700\ \text{nJ} = 7.55\ \mu\text{J}$.

For the PMD with a 1 MB L2, the 1000-instruction program would finish in $1000 + 100 \times 20\ \text{ns} + 10 \times 100\ \text{ns} = 4000$ ns.

Energy $= 2.3\ \text{W} \times 4000\ \text{ns} + 100 \times 0.7\ \text{nJ} + 10 \times 35\ \text{nJ} = 9200 + 70 + 350\ \text{nJ} = 9.62\ \mu\text{J}$.

Therefore, of these designs, lowest energy for the PMD is achieved with a 256 KB L2 cache.

2.25 (a) A small block size ensures that we are never fetching more bytes than required by the processor. This reduces L2 and memory access power. This is slightly offset by the need for more cache tags and higher tag array power. However, if this leads to more application misses and longer program completion time, it may ultimately result in higher application energy. For example, in the previous exercise, notice how the design with no L2 cache results in a long execution time and highest energy. (b) A small cache size would lower cache power, but it increases memory power because the number of memory accesses will be higher. As seen in the previous exercise, the MPKIs, latencies, and energy per access ultimately decide if energy will increase or decrease. (c) Higher associativity will result in higher tag array power, but it should lower the miss rate and memory access power. It should also result in lower execution time, and eventually lower application energy.

2.26 (a) The LRU policy essentially uses recency of touch to determine priority. A newly fetched block is inserted at the head of the priority list. When a block is touched, the block is immediately promoted to the head of the priority list. When a block must be evicted, we select the block that is currently at the tail of the priority list. (b) Research studies have shown that some blocks are not touched during their residence in the cache. An Insertion policy that exploits this observation would insert a recently fetched block near the tail of the priority list. The Promotion policy moves a block to the head of the priority list when touched. This gives a block a longer residence in the cache only when it is touched at least twice within a short window. It may also be reasonable to implement a Promotion policy that gradually moves a block a few places ahead in the priority list on every touch.

2.27 The standard approach to isolating the behavior of each program is cache partitioning. In this approach, each program receives a subset of the ways in the shared cache. When providing QoS, the ways allocated to each program can be dynamically varied based on program behavior and the service levels guaranteed to each program. When providing privacy, the allocation of ways has to be determined beforehand and cannot vary at runtime. Tuning the allocation based on the programs' current needs would result in information leakage.

2.28 A NUCA cache yields higher performance if more requests can be serviced by the banks closest to the processor. Note that we already implement a priority list (typically based on recency of access) within each set of the cache. This priority list can be used to map blocks to the NUCA banks. For example, frequently touched blocks are gradually promoted to low-latency banks while blocks that haven't been touched recently are demoted to higher-latency banks. Note that such block migrations within the cache will increase cache power. They may also complicate cache look-up.

2.29 a. A 2 GB DRAM with parity or ECC effectively has 9 bit bytes, and would require 18 1 Gb DRAMs. To create 72 output bits, each one would have to output $72/18=4$ bits.

b. A burst length of 4 reads out 32B.

c. The DDR-667 DIMM bandwidth is $667\times 8=5336$ MB/s.

The DDR-533 DIMM bandwidth is $533\times 8=4264$ MB/s.

2.30 a. This is similar to the scenario given in the figure, but tRCD and CL are both 5. In addition, we are fetching two times the data in the figure. Thus, it requires $5+5+4\times 2=18$ cycles of a 333 MHz clock, or $18\times(1/333\text{ MHz})=54.0$ ns.

b. The read to an open bank requires $5+4=9$ cycles of a 333 MHz clock, or 27.0 ns. In the case of a bank activate, this is 14 cycles, or 42.0 ns. Including 20 ns for miss processing on chip, this makes the two $42+20=61$ ns and $27.0+20=47$ ns. Including time on chip, the bank activate takes $61/47=1.30$ or 30% longer.

2.31 The costs of the two systems are $\$2\times 130+\$800=\$1060$ with the DDR2-667 DIMM and $2\times\$100+\$800=\$1000$ with the DDR2-533 DIMM. The latency to service a level-2 miss is $14\times(1/333\text{ MHz})=42$ ns 80% of the time and $9\times(1/333\text{ MHz})=27$ ns 20% of the time with the DDR2-667 DIMM.

It is $12\times(1/266\text{ MHz})=45$ ns (80% of the time) and $8\times(1/266\text{ MHz})=30$ ns (20% of the time) with the DDR-533 DIMM. The CPI added by the level-2 misses in the case of DDR2-667 is $0.00333\times 42\times 0.8+0.00333\times 27\times 0.2=0.130$ giving a total of $1.5+0.130=1.63$. Meanwhile the CPI added by the level-2 misses for DDR-533 is $0.00333\times 45\times 0.8+0.00333\times 30\times 0.2=0.140$ giving a total of $1.5+0.140=1.64$. Thus, the drop is only $1.64/1.63=1.006$, or 0.6%, while the cost is $\$1060/\$1000=1.06$ or 6.0% greater. The cost/performance of the DDR2-667 system is $1.63 \times 1060 = 1728$ while the cost/performance of the DDR2-533 system is $1.64 \times 1000 = 1640$, so the DDR2-533 system is a better value.

2.32 The cores will be executing 8 cores × 3 GHz/2.0CPI = 12 billion instructions per second. This will generate $12 \times 0.00667 = 80$ million level-2 misses per second. With the burst length of 8, this would be $80 \times 32\text{B} = 2560$ MB/s. If the memory bandwidth is sometimes 2X this, it would be 5120 MB/s. From Fig. 2.14, this is just barely within the bandwidth provided by DDR2-667 DIMMs, so just one memory channel would suffice.

2.33 We will assume that applications exhibit spatial locality and that accesses to consecutive memory blocks will be issued in a short time window. If consecutive blocks are in the same bank, they will yield row buffer hits. While this reduces Activation energy, the two blocks have to be fetched sequentially. The second access, a row buffer hit, will experience lower latency than the first access. If consecutive blocks are in banks on different channels, they will both be row buffer misses, but the two accesses can be performed in parallel. Thus, interleaving consecutive blocks across different channels and banks can yield lower latencies, but can also consume more memory power.

2.34 a. The system built from 1 Gb DRAMs will have twice as many banks as the system built from 2 Gb DRAMs. Thus, the 1 Gb-based system should provide higher performance because it can have more banks simultaneously open.

b. The power required to drive the output lines is the same in both cases, but the system built with the x4 DRAMs would require activating banks on 18 DRAMs, versus only 9 DRAMs for the x8 parts. The page size activated on each x4 and x8 part are the same, and take roughly the same activation energy. Thus, because there are fewer DRAMs being activated in the x8 design option, it would have lower power.

2.35 a. With policy 1,
Precharge delay Trp = 5 × (1/333 MHz) = 15 ns
Activation delay Trcd = 5 × (1/333 MHz) = 15 ns
Column select delay Tcas = 4 × (1/333 MHz) = 12 ns
Access time when there is a row buffer hit

$$T_h = \frac{r(\text{Tcas} + \text{Tddr})}{100}$$

Access time when there is a miss

$$T_m = \frac{(100 - r)(\text{Trp} + \text{Trcd} + \text{Tcas} + \text{Tddr})}{100}$$

With policy 2,
Access time = $Trcd + Tcas + Tddr$
If A is the total number of accesses, the tip-off point will occur when the net access time with policy 1 is equal to the total access time with policy 2.
that is,

$$\begin{aligned}\frac{\mathrm{r}}{100}(\mathrm{Tcas}+\mathrm{Tddr})\mathrm{A}+\frac{100-\mathrm{r}}{100}(\mathrm{Trp}+\mathrm{Trcd}+\mathrm{Tcas}+\mathrm{Tddr})\mathrm{A} \\ =(\mathrm{Trcd}+\mathrm{Tcas}+\mathrm{Tddr})\mathrm{A} \\ \Rightarrow \mathrm{r}=\frac{100\times\mathrm{Trp}}{\mathrm{Trp}\times\mathrm{Trcd}} \\ r=100\times(15)/(15+15)=50\%\end{aligned}$$

If r is less than 50%, then we have to proactively close a page to get the best performance, else we can keep the page open.

b. The key benefit of closing a page is to hide the precharge delay Trp from the critical path. If the accesses are back to back, then this is not possible. This new constrain will not impact policy 1.
The new equations for policy 2,
Access time when we can hide precharge delay $= Trcd + Tcas + Tddr$
Access time when precharge delay is in the critical path $= Trcd + Tcas + Trp + Tddr$
Equation 1 will now become,

$$\begin{aligned}\frac{r}{100}(Tcas+Tddr)A+\frac{100-r}{100}(Trp+Trcd+Tcas+Tddr)A \\ =0.9\times(Trcd+Tcas+Tddr)A+0.1\times(Trcd+Tcas+Trp+Tddr) \\ \Rightarrow r=90\times\left(\frac{Trp}{Trp\times Trcd}\right) \\ r=90\times 15130=45\%\end{aligned}$$

c. For any row buffer hit rate, policy 2 requires additional $r\times(2+4)$ nJ per access. If $r=50\%$, then policy 2 requires 3 nJ of additional energy.

2.36 Hibernating will be useful when the static energy saved in DRAM is at least equal to the energy required to copy from DRAM to Flash and then back to DRAM. DRAM dynamic energy to read/write is negligible compared to Flash and can be ignored.

$$\begin{aligned}Time&=\frac{8\times10^{9}\times2\times2.56\times10^{-6}}{64\times1.6} \\ &=400\,seconds\end{aligned}$$

The factor 2 in the above equation is because to hibernate and wakeup, both Flash and DRAM have to be read and written once.

2.37 a. Yes. The application and production environment can be run on a VM hosted on a development machine.

b. Yes. Applications can be redeployed on the same environment on top of VMs running on different hardware. This is commonly called business continuity.

c. No. Depending on support in the architecture, virtualizing I/O may add significant or very significant performance overheads.

d. Yes. Applications running on different virtual machines are isolated from each other.

e. Yes. See "Devirtualizable virtual machines enabling general, single-node, online maintenance," David Lowell, Yasushi Saito, and Eileen Samberg, in the Proceedings of the 11th ASPLOS, 2004, pages 211–223.

2.38 a. Programs that do a lot of computation, but have small memory working sets and do little I/O or other system calls.

b. The slowdown above previously was 60% for 10%, so 20% system time would run 120% slower.

c. The median slowdown using pure virtualization is 10.3, while for para virtualization, the median slowdown is 3.76.

d. The null call and null I/O call have the largest slowdown. These have no real work to outweigh the virtualization overhead of changing protection levels, so they have the largest slowdowns.

2.39 The virtual machine running on top of another virtual machine would have to emulate privilege levels as if it was running on a host without VT-x technology.

2.40 a. As of the date of the computer paper, AMD-V adds more support for virtualizing virtual memory, so it could provide higher performance for memory-intensive applications with large memory footprints.

b. Both provide support for interrupt virtualization, but AMD's IOMMU also adds capabilities that allow secure virtual machine guest operating system access to selected devices.

2.41 Open hands-on exercise, no fixed solution

2.42 An aggressive prefetcher brings in useful blocks as well as several blocks that are not immediately useful. If prefetched blocks are placed in the cache (or in a prefetch buffer for that matter), they may evict other blocks that are imminently useful, thus potentially doing more harm than good. A second significant downside is an increase in memory utilization, that may increase queuing delays for demand accesses. This is especially problematic in multicore systems where the bandwidth is nearly saturated and at a premium.

2.43 a. These results are from experiments on a 3.3GHz Intel® Xeon® Processor X5680 with Nehalem architecture (westmere at 32 nm). The number of misses per 1 K instructions of L1 Dcache increases significantly by more than 300X when input data size goes from 8 to 64 KB, and keeps relatively constant around 300/1 K instructions for all the larger data sets. Similar behavior with different flattening points on L2 and L3 caches are observed.

b. The IPC decreases by 60%, 20%, and 66% when input data size goes from 8 to 128 KB, from 128 KB to 4 MB, and from 4 to 32 MB, respectively. This shows the importance of all caches. Among all three levels, L1 and L3 caches are more important. This is because the L2 cache in the Intel® Xeon® Processor X5680 is relatively small and slow, with capacity being 256 KB and latency being around 11 cycles.

c. For a recent Intel i7 processor (3.3GHz Intel® Xeon® Processor X5680), when the data set size is increased from 8 to 128 KB, the number of L1 Dcache misses per 1 K instructions increases by around 300, and the number of L2 cache misses per 1 K instructions remains negligible. With a 11 cycle miss penalty, this means that without prefetching or latency tolerance from out-of-order issue, we would expect there to be an extra 3300 cycles per 1 K instructions due to L1 misses, which means an increase of 3.3 cycles per instruction on average. The measured CPI with the 8 KB input data size is 1.37. Without any latency tolerance mechanisms, we would expect the CPI of the 128 KB case to be $1.37 + 3.3 = 4.67$. However, the measured CPI of the 128 KB case is 3.44. This means that memory latency hiding techniques such as OOO execution, prefetching, and nonblocking caches improve the performance by more than 26%.

Chapter 3 Solutions

Case Study 1: Exploring the Impact of Microarchitectural Techniques

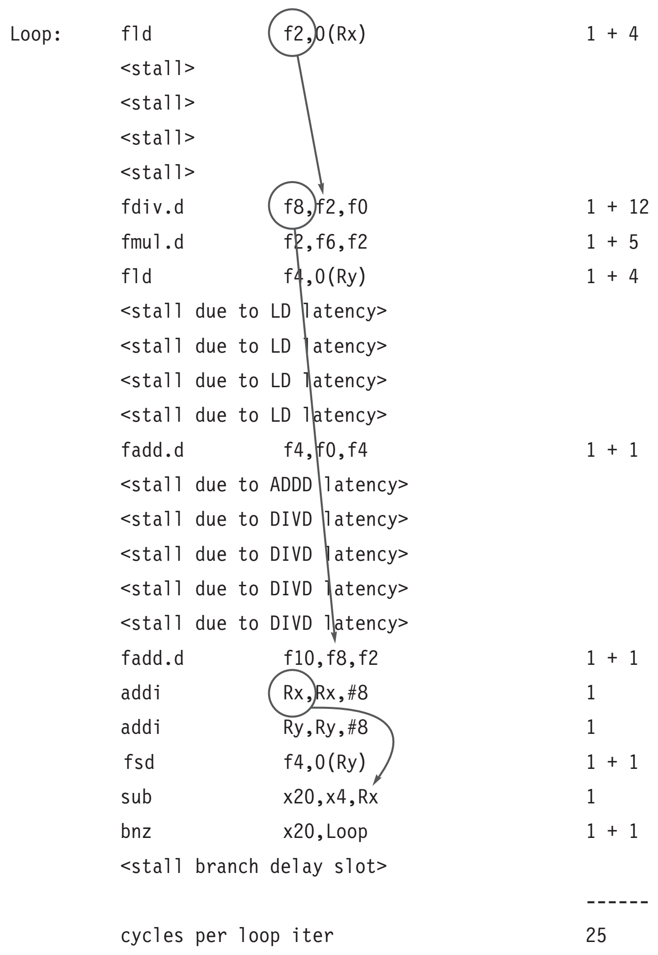
3.1 The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 38. Each instruction requires one clock cycle of execution (a clock cycle in which that instruction, and only that instruction, is occupying the execution units; since every instruction must execute, the loop will take at least that many clock cycles). To that base number, we add the extra latency cycles. Don't forget the branch shadow cycle.

3.2 How many cycles would the loop body in the code sequence in Figure 3.47 require if the pipeline detected true data dependencies and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? The answer is 25, as shown in Figure S.1. Remember, the point of the extra latency cycles is to allow an instruction to complete whatever actions it needs, in order to produce its correct output. Until that output is ready, no dependent instructions can be executed. So, the first `fld` must stall the next instruction for three clock cycles. The `fmul.d` produces a result for its successor, and therefore must stall 4 more clocks, and so on.

3.3 Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require? The answer is 22. The `fld` goes first, as before, and the `fdiv.d` must wait for it through four extra latency cycles. After the `fdiv.d` comes the `fmul.d`, which can run in the second pipe along with the `fdiv.d`, since there's no dependency between them. (Note that they both need the same input, F2, and they must both wait on F2's readiness, but there is no constraint between them.) The `fld` following the `fmul.d` does not depend on the `fdiv.d` nor the `fmul.d`, so had this been a superscalar-order-3 machine, that `fld` could conceivably have been executed concurrently with the `fdiv.d` and the `fmul.d`. Since this problem posited a two-execution-pipe machine, the `fld` executes in the cycle following the `fdiv.d`/`fmul.d`. The loop overhead instructions at the loop's bottom also exhibit some potential for concurrency because they do not depend on any long-latency instructions.

3.4 Possible answers:

1. If an interrupt occurs between N and $N+1$, then $N+1$ must not have been allowed to write its results to any permanent architectural state. Alternatively, it might be permissible to delay the interrupt until $N+1$ completes.



```
Loop:   fld                      f2,0(Rx)             1 + 4
        <stall>
        <stall>
        <stall>
        <stall>
        fdiv.d                   f8,f2,f0             1 + 12
        fmul.d                   f2,f6,f2             1 + 5
        fld                      f4,0(Ry)             1 + 4
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        fadd.d                   f4,f0,f4             1 + 1
        <stall due to ADDD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        fadd.d                   f10,f8,f2            1 + 1
        addi                     Rx,Rx,#8             1
        addi                     Ry,Ry,#8             1
        fsd                      f4,0(Ry)             1 + 1
        sub                      x20,x4,Rx            1
        bnz                      x20,Loop             1 + 1
        <stall branch delay slot>
                                                      ------
        cycles per loop iter                          25
```

Figure S.1 Number of cycles required by the loop body in the code sequence in Figure 3.47

2. If N and $N+1$ happen to target the same register or architectural state (say, memory), then allowing N to overwrite what $N+1$ wrote would be wrong.
3. N might be a long floating-point op that eventually traps. $N+1$ cannot be allowed to change arch state in case N is to be retried.

Long-latency ops are at highest risk of being passed by a subsequent op. The `fdiv.d` instr will complete long after the `fld f4,0(Ry)`, for example.

3.5 Figure S.2 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 3.47. The number of cycles that this reordered code takes is 22.

cycle	pipeline 1	pipeline 2
1	Loop: fld f2,0(Rx)	fld f4,0(Ry)
2	addi Rx,Rx,#8	<stall due to LD latency>
3	<stall due to LD latency>	<stall due to LD latency>
4	<stall due to LD latency>	<stall due to LD latency>
5	<stall due to LD latency>	<stall due to LD latency>
6	fmul.d f2,f6,f2	fadd.d f4,f0,f4
7	<stall due to MULD latency>	<stall due to ADDD latency>
8	<stall due to MULD latency>	<stall due to ADDD latency>
9	<stall due to MULD latency>	fsd f4,0(Ry)
10	<stall due to MULD latency>	addi Ry,Ry,#8
11	fdiv.d f8,f2,f0	<no ready instructions>
12	sub x20,x4,Rx	<no ready instructions>
13	<stall due to DIVD latency>	<no ready instructions>
14	<stall due to DIVD latency>	<no ready instructions>
15	<stall due to DIVD latency>	<no ready instructions>
16	<stall due to DIVD latency>	<no ready instructions>
17	<stall due to DIVD latency>	<no ready instructions>
18	<stall due to DIVD latency>	<no ready instructions>
19	<stall due to DIVD latency>	<no ready instructions>
20	<stall due to DIVD latency>	<no ready instructions>
21	bnz x20,Loop	<no ready instructions>
22	fadd.d f10,f8,f2	<no ready instructions>

Figure S.2 Number of cycles taken by reordered code.

3.6 a. Fraction of all cycles, counting both pipes, wasted in the reordered code shown in Figure S.2:

$$\begin{aligned} &11 \text{ ops out of } 2 \times 20 \text{ opportunities} \\ &1 - 11/40 = 1 - 0.275 \\ &= 0.725 \end{aligned}$$

b. Results of hand-unrolling two iterations of the loop from code shown in Figure S.3:

c.

$$\text{Speedup} = \frac{\text{exec time w/o enhancement}}{\text{exec time with enhancement}}$$

$$\text{Speedup} = 20/(22/2)$$

$$= 1.82$$

3.7 Consider the code sequence in Figure 3.48. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src (source) registers accordingly, so that true data dependencies are maintained. Show the resulting code. (*Hint:* see Figure 3.49.)

3.8 See Figure S.4. The rename table has arbitrary values at clock cycle $N-1$. Look at the next two instructions (I0 and I1): I0 targets the F1 register, and I1 will write the F4 register. This means that in clock cycle N, the rename table will have had its

cycle	pipeline 1	pipeline 2
1	Loop: fld f2,0(Rx)	fld f4,0(Ry)
2	fld f12,8(Rx)	fld f14,8(Ry)
3	addi Rx,Rx,#16	<stall due to LD latency>
4	<stall due to LD latency>	<stall due to LD latency>
5	<stall due to LD latency>	<stall due to LD latency>
6	fmul.d f2,f6,f2	fadd.d f4,f0,f4
7	fmul.d f12,f6,f12	fadd.d f14,f0,f14
8	<stall due to MULD latency>	<stall due to ADDD latency>
9	<stall due to MULD latency>	fsd f4,0(Ry)
10	<stall due to MULD latency>	fsd f14,0(Ry)
11	fdiv.d f8,f2,f0	addi Ry,Ry,#16
12	fdiv.d f18,f12,f0	<stall due to DIVD latency>
13	sub x20,x4,Rx	<stall due to DIVD latency>
14	<stall due to DIVD latency>	<stall due to DIVD latency>
15	<stall due to DIVD latency>	<stall due to DIVD latency>
16	<stall due to DIVD latency>	<stall due to DIVD latency>
17	<stall due to DIVD latency>	<stall due to DIVD latency>
18	<stall due to DIVD latency>	<stall due to DIVD latency>
19	<stall due to DIVD latency>	<stall due to DIVD latency>
20	<stall due to DIVD latency>	<stall due to DIVD latency>
21	bnz x20,Loop	<stall due to DIVD latency>
22	fadd.d f20,f18,f12	fadd.d f10,f8,f2

Figure S.3 Hand-unrolling two iterations of the loop from code shown in Figure S.2

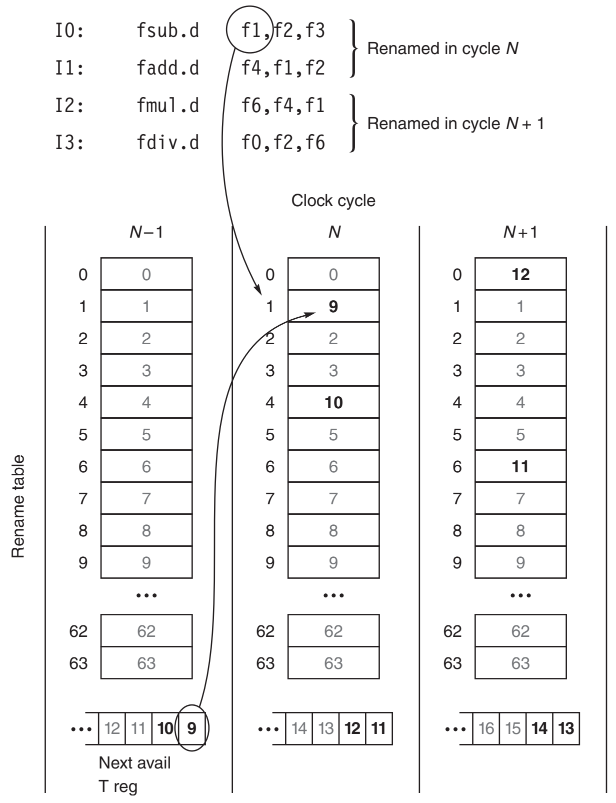


Figure S.4 Cycle-by-cycle state of the rename table for every instruction of the code in Figure 3.50.

```
addi T0, x1, x1
addi T1, T0, T0
addi x1, T1, T1

Value in X1 should be 40
```

Figure S.5 Value of X1 when the sequence has been executed.

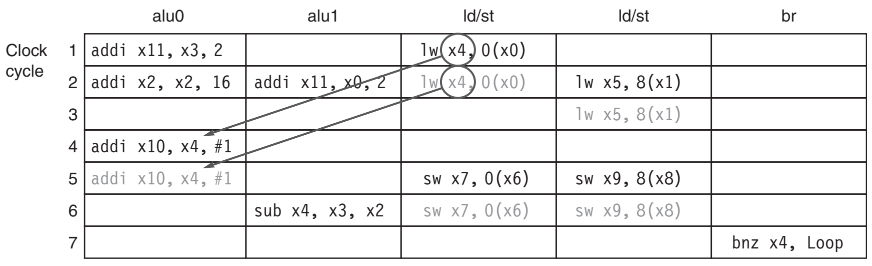
entries 1 and 4 overwritten with the next available Temp register designators. I0 gets renamed first, so it gets the first T reg (9). I1 then gets renamed to T10. In clock cycle *N*, instructions I2 and I3 come along; I2 will overwrite F6, and I3 will write F0. This means the rename table's entry 6 gets 11 (the next available T reg), and rename table entry 0 is written to the T reg after that (12). In principle, you don't have to allocate T regs sequentially, but it's much easier in hardware if you do.

3.9 See Figure S.5.

3.10 An example of an event that, in the presence of self-draining pipelines, could disrupt the pipelining and yield wrong results is shown in Figure S.6.

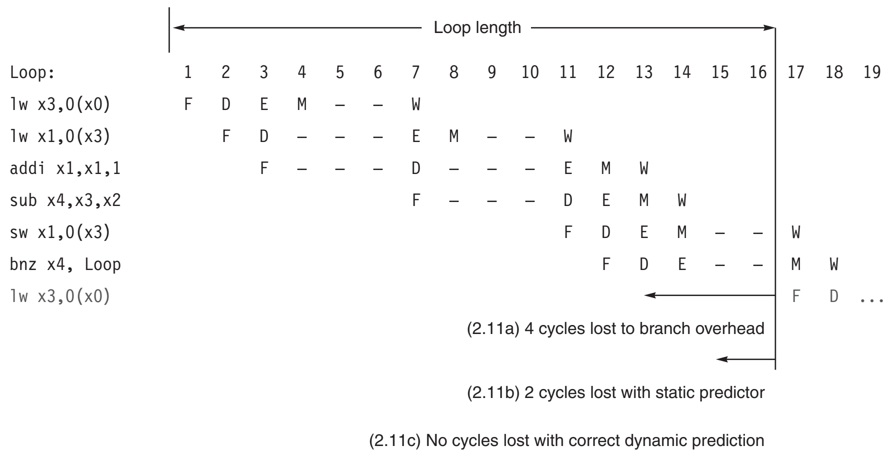
3.11 See Figure S.7. The convention is that an instruction does not enter the execution phase until all of its operands are ready. So, the first instruction, `ld x1,0(x0)`, marches through its first three stages (F, D, E) but that M stage that comes next requires the usual cycle plus two more for latency. Until the data from a `ld` is available at the execution unit, any subsequent instructions (especially that `addi x1,x1,#1`, which depends on the 2nd `ld`) cannot enter the E stage, and must therefore stall at the D stage.

a. Four cycles lost to branch overhead. Without bypassing, the results of the `sub` instruction are not available until the `sub`'s W stage. That tacks on an extra 4 clock cycles at the end of the loop, because the next loop's `ld x1` can't begin until the branch has completed.



Clock cycle	alu0	alu1	ld/st	ld/st	br
1	addi x11, x3, 2		lw x4, 0(x0)		
2	addi x2, x2, 16	addi x11, x0, 2	lw x4, 0(x0)	lw x5, 8(x1)	
3				lw x5, 8(x1)	
4	addi x10, x4, #1				
5	addi x10, x4, #1		sw x7, 0(x6)	sw x9, 8(x8)	
6		sub x4, x3, x2	sw x7, 0(x6)	sw x9, 8(x8)	
7					bnz x4, Loop

Figure S.6 Example of an event that yields wrong results. What could go wrong with this? If an interrupt is taken between clock cycles 1 and 4, then the results of the LW at cycle 2 will end up in R1, instead of the LW at cycle 1. Bank stalls and ECC stalls will cause the same effect—pipes will drain, and the last writer wins, a classic WAW hazard. All other "intermediate" results are lost.


Loop length
(2.11a) 4 cycles lost to branch overhead
(2.11b) 2 cycles lost with static predictor
(2.11c) No cycles lost with correct dynamic prediction

Loop:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
lw x3,0(x0)	F	D	E	M	–	–	W												
lw x1,0(x3)		F	D	–	–	–	E	M	–	–	W								
addi x1,x1,1			F	–	–	–	D	–	–	–	E	M	W						
sub x4,x3,x2							F	–	–	–	D	E	M	W					
sw x1,0(x3)											F	D	E	M	–	–	W		
bnz x4, Loop												F	D	E	–	–	M	W	
lw x3,0(x0)																	F	D	...

Figure S.7 Phases of each instruction per clock cycle for one iteration of the loop.

b. Two cycles lost w/ static predictor. A static branch predictor may have a heuristic like "if branch target is a negative offset, assume it's a loop edge, and loops are usually taken branches." But we still had to fetch and decode the branch to see that, so we still lose 2 clock cycles here.

c. No cycles lost w/ correct dynamic prediction. A dynamic branch predictor remembers that when the branch instruction was fetched in the past, it eventually turned out to be a branch, and this branch was taken. So, a "predicted taken" will occur in the same cycle as the branch is fetched, and the next fetch after that will be to the presumed target. If correct, we've saved all of the latency cycles seen in 3.11 (a) and 3.11 (b). If not, we have some cleaning up to do.

3.12 a. See Figure S.8.

b. See Figure S.9. The number of clock cycles taken by the code sequence is 25.

c. See Figures S.10 and S.11. The bold instructions are those instructions that are present in the RS, and ready for dispatch. Think of this exercise from the Reservation Station's point of view: at any given clock cycle, it can only "see" the instructions that were previously written into it, that have not already dispatched. From that pool, the RS's job is to identify and dispatch the two eligible instructions that will most boost machine performance.

d. See Figure S.12.
1. Another ALU: 0% improvement
2. Another LD/ST unit: 0% improvement
3. Full bypassing: critical path is `fld` -> `fdiv.d` -> `fmult.d` -> `fadd.d`. Bypassing would save 1 cycle from latency of each, so 4 cycles total.
4. Cutting longest latency in half: divider is longest at 12 cycles. This would save 6 cycles total.

e. See Figure S.13.

```
fld         f2,0(Rx)
fdiv.d      f8,f2,f0
fmul.d      f2,f8,f2      ; reg renaming doesn't really help here, due to
                          ; true data dependencies on F8 and F2
fld         F4,0(Ry)      ; this LD is independent of the previous 3
                          ; instrs and can be performed earlier than
                          ; pgm order. It feeds the next ADDD, and ADDD
                          ; feeds the SD below. But there's a true data
                          ; dependency chain through all, so no benefit
fadd.d      f4,f0,f4
fadd.d      f10,f8,f2     ; This ADDD still has to wait for DIVD latency,
                          ; no matter what you call their rendezvous reg
addi        Rx,Rx,#8      ; rename for next loop iteration
addi        Ry,Ry,#8      ; rename for next loop iteration
fsd         f4,0(Ry)      ; This SD can start when the ADDD's latency has
                          ; transpired. With reg renaming, doesn't have
                          ; to wait until the LD of (a different) F4 has
                          ; completed.
sub         x20,x4,Rx
bnz         x20,Loop
```

Figure S.8 Instructions in code where register renaming improves performance.

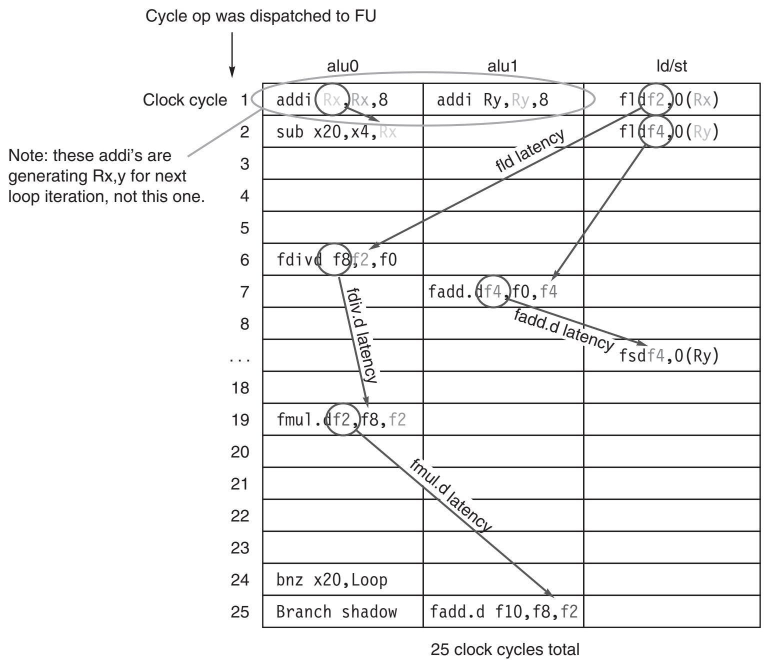
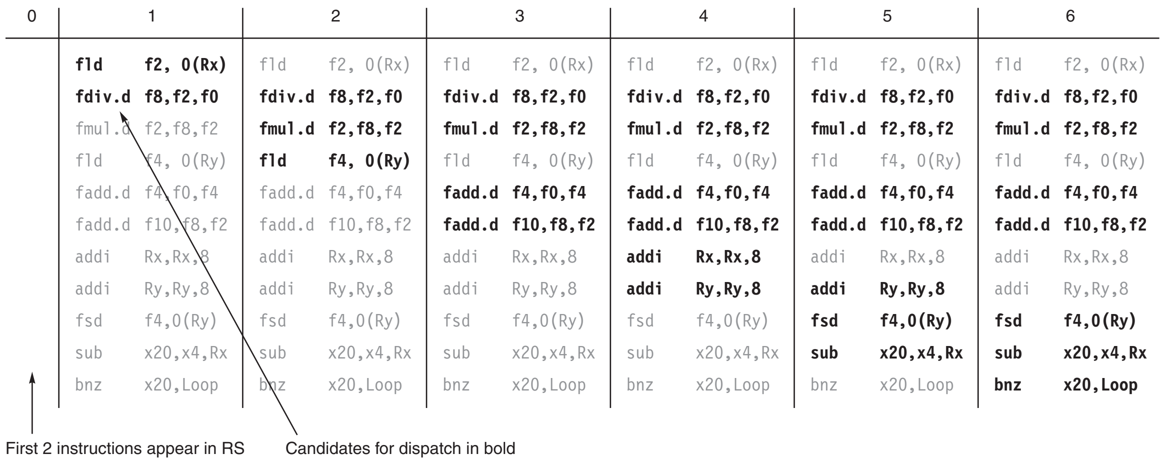


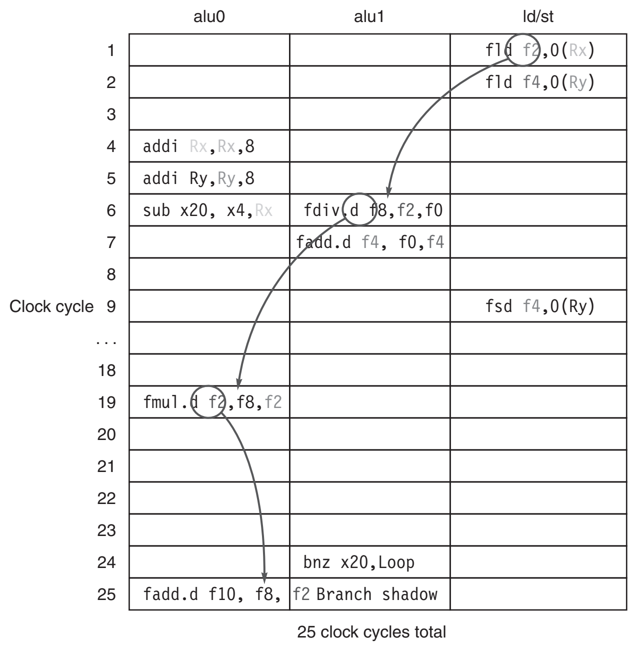
Figure S.9 Number of clock cycles taken by the code sequence.



0	1	2	3	4	5	6
	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)
	fdiv.d f8,f2,f0	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**
	fmul.d f2,f8,f2	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**
	fld f4, 0(Ry)	**fld f4, 0(Ry)**	fld f4, 0(Ry)	fld f4, 0(Ry)	fld f4, 0(Ry)	fld f4, 0(Ry)
	fadd.d f4,f0,f4	fadd.d f4,f0,f4	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**
	fadd.d f10,f8,f2	fadd.d f10,f8,f2	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**
	addi Rx,Rx,8	addi Rx,Rx,8	addi Rx,Rx,8	**addi Rx,Rx,8**	addi Rx,Rx,8	addi Rx,Rx,8
	addi Ry,Ry,8	addi Ry,Ry,8	addi Ry,Ry,8	**addi Ry,Ry,8**	**addi Ry,Ry,8**	addi Ry,Ry,8
	fsd f4,0(Ry)	fsd f4,0(Ry)	fsd f4,0(Ry)	fsd f4,0(Ry)	**fsd f4,0(Ry)**	**fsd f4,0(Ry)**
	sub x20,x4,Rx	sub x20,x4,Rx	sub x20,x4,Rx	sub x20,x4,Rx	**sub x20,x4,Rx**	**sub x20,x4,Rx**
	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	**bnz x20,Loop**

First 2 instructions appear in RS
Candidates for dispatch in bold

Figure S.10 Candidates for dispatch.



Clock cycle	alu0	alu1	ld/st
1			fld f2,0(Rx)
2			fld f4,0(Ry)
3			
4	addi Rx,Rx,8		
5	addi Ry,Ry,8		
6	sub x20, x4,Rx	fdiv.d f8,f2,f0	
7		fadd.d f4, f0,f4	
8			
9			fsd f4,0(Ry)
...			
18			
19	fmul.d f2,f8,f2		
20			
21			
22			
23			
24		bnz x20,Loop	
25	fadd.d f10, f8, f2	Branch shadow	

Figure S.11 5 Number of clock cycles required.

Cycle op was dispatched to FU

Clock cycle

	alu0	alu1	ld/st
1			`fld f2,0(Rx)`
2			`fld f4,0(Ry)`
3			
4	`addi Rx,Rx,8`		
5	`addi Ry,Ry,8`		
6	`sub x20,x4,Rx`	`fdiv.d f8,f2,f0`	
7		`fadd.d f4,f0,f4`	
8			
9			`fsd f4,0(Ry)`
. . .			
18			
19	`fmul.d f2,f8,f2`		
20			
21			
22			
23			
24		`bnz x20,Loop`	
25	`fadd.d f10, f8,f2`	`Branch shadow`	

25 clock cycles total

Figure S.12 Speedup is (execution time without enhancement)/(execution time with enhancement) = 25/(25 − 6) = 1.316.

Cycle op was dispatched to FU

Clock cycle

	alu0	alu1	ld/st
1			`fld f2,0(Rx)`
2			`fld f2,0(Rx)`
3			`fld f4,0(Ry)`
4	`addi Rx,Rx,8`		
5	`addi Ry,Ry,8`		
6	`sub x20,x4,Rx`	`fdiv.d f8,f2,f0`	
7		`fdiv.d f8,f2,f0`	
8		`fadd.d f4,f0,f4`	
9			
. . .			`fsd f4,0(Ry)`
18			
19	`fmul.d f2,f8,f2`		
20	`fmul.d f2,f8, f2`		
21			
22			
23			
24			
25	`fadd.d f10,f8,f2`	`bnz x20,Loop`	
26	`fadd.d f10,f8,f2`	`Branch shadow`	

26 clock cycles total

Figure S.13 Number of clock cycles required to do two loops' worth of work. Critical path is LD -> DIVD -> MULTD -> ADDD. If RS schedules 2nd loop's critical LD in cycle 2, then loop 2's critical dependency chain will be the same length as loop 1's is. Since we're not functional-unit-limited for this code, only one extra clock cycle is needed.

3.13 Processor A:

Cycle	Slot 1	Slot 2	Notes
1	fld x1(thread 0)	fld x1(thread 1)	threads 1,2 stalled until cycle 5
2	fld x1 (thread 2)	fld x1 (thread 3)	threads 3,4 stalled until cycle 6
3	stall	stall	
4	stall	stall	
5	fld x2(thread 0)	fld x2(thread 1)	threads 1,2 stalled until cycle 9
6	fld x2 (thread 2)	fld x2 (thread 3)	threads 3,4 stalled until cycle 10
7	stall	stall	
8	stall	stall	
…			
33	beq (thread 0)	beq (thread 1)	threads 1,2 stalled until cycle 37
34	beq (thread 2)	beq (thread 3)	threads 3,4 stalled until cycle 38
35	stall	stall	
36	stall	stall	
…			
65	addi (thread 0)	addi (thread 1)	
66	addi (thread 2)	addi (thread 3)	
67	blt (thread 0)	blt (thread 1)	
68	blt (thread 2)	blt (thread 3)	
69	stall	stall	
70	stall	stall	first iteration ends
71			second iteration begins
140			second iteration ends

Processor B:

Cycle	Slot 1	Slot 2	Slot 3	Slot 4	Notes
1	fld x1 (th 0)	fld x2 (th 0)	LD x3 (th 0)	LD x4 (th 0)	
2	fld x1 (th 1)	fld x2 (th 1)	fld x3 (th 1)	fld x4 (th 1)	
3	fld x1 (th 2)	fld x2 (th 2)	fld x3 (th 2)	fld x4 (th 2)	
4	fld x1 (th 3)	fld x2 (th 3)	fld x3 (th 3)	fld x4 (th 3)	
5	fld x5 (th 0)	fld x6 (th 0)	fld x7 (th 0)	fld x8 (th 0)	
6	fld x5 (th 1)	fld x6 (th 1)	fld x7 (th 1)	fld x8 (th 1)	
7	fld x5 (th 2)	fld x6 (th 2)	fld x7 (th 2)	fld x8 (th 2)	
8	fld x5 (th 3)	fld x6 (th 3)	fld x7 (th 3)	fld x8 (th 3)	
9	beq (th 0)				first beq of each thread
10	beq (th 1)				

11	beq (th 2)	
12	beq (th 3)	
13	beq (th 0)	second beq if each thread
14	beq (th 1)	
15	beq (th 2)	
16	beq (th 3)	
…		
41	addi (th 0)	
42	addi (th 1)	
43	addi (th 2)	
44	addi (th 3)	
45	blt (th 0)	
46	blt (th 1)	
47	blt (th 2)	
48	blt (th 3)	end of first iteration
…		
96		second iteration ends

Processor C:

Cycle	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7	Slot 8	Notes
1	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	
2	stall								
3	stall								
4	stall								
5	beq x1 (th0)								
6	stall								
7	stall								
8	stall								
9	beq x2 (th0)								
10	stall								
11	stall								
12	stall								
…									
37	addi (th 0)								

38	blt (th0)								
39	stall								
40	stall								
41	stall								
42	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	start of second iteration (th 0)
43	stall								
44	stall								
45	stall								
…									
83	fld (th 1)	fld (th 2)	fld (th 3)	fld (th 4)	fld (th 5)	fld (th 6)	fld (th 7)	fld (th 8)	start of first iteration (th 1)
…									
328									end of second iteration (th 3)

3.14. a.

Clock cycle	Unscheduled code	Scheduled code
1	addi x4,x1,#800	addi x4,x1,#800
2	fld f2,0(x1)	fld f2,0(x1)
3	stall	fld f6,0(x2)
4	fmul.d f4,f2,f0	fmul.d f4,f2,f0
5	fld f6,0(x2)	addi x1,x1,#8
6	stall	addi x2,x2,#8
	stall	sltu x3,x1,x4
	stall	stall
	stall	stall
7	fadd.d f6,f4,f6	fadd.d f6,f4,f6
8	stall	stall
9	stall	stall
10	stall	bnez x3,foo
11	fsd f6,0(x2)	fsd f6,-8(x2)
12	addi x1,x1,#8	
13	addi x2,x2,#8	
14	sltu x3,x1,x4	
15	stall	
16	bnez x3,foo	
17	stall	

The execution time per element for the unscheduled code is 16 clock cycles and for the scheduled code is 10 clock cycles. This is 60% faster, so the clock must be 60% faster for the unscheduled code to match the performance of the scheduled code on the original hardware.

b.

Clock cycle	Scheduled code
1	addi x4,x1,#800
2	fld f2,0(x1)
3	fld f6,0(x2)
4	fmul.d f4,f2,f0
5	fld f2,8(x1)
6	fld f10,8(x2)
7	fmul.d f8,f2,f0
8	fld f2,8(x1)
9	fld f14,8(x2)
10	fmul.d f12,f2,f0
11	fadd.d f6,f4,f6
12	addi x1,x1,#24
13	fadd.d f10,f8,f10
14	addi x2,x2,#24
15	sltu x3,x1,x4
16	fadd.d f14,f12,f14
17	fsd f6,-24(x2)
18	fsd f10,-16(x2)
19	bnez x3,foo
20	fsd f14,-8(x2)

The code must be unrolled three times to eliminate stalls after scheduling.

c. Unrolled six times:

Cycle	Memory reference 1	Memory reference 2	fP operation 1	fP operation 2	Integer operation/ branch
1	fld f1, 0(x1)	fld f2, 8(x1)			
2	fld f3, 16(x1)	fld f4, 24(x1)			
3	fld f5, 32(x1)	fld f6, 40(x1)	fmul.d f1,f1,f0	fmul.d f2,f2,f0	
4	fld f7, 0(x2)	fld f8, 8(x2)	fmul.d f3,f3,f0	fmul.d f4,f4,f0	
5	fld f9, 16(x2)	fld f10, 24(x2)	fmul.d f5,f5,f0	fmul.d f6,f6,f0	

6	fld f11 32(x2)	fld f12, 40(x2)			
7					addi x1,x1,48
8					addi x2,x2,48
9			fadd.d f7,f7,f1	fadd.d f8,f8,f2	
10			fadd.d f9,f9,f3	fadd.d f10,f10, f4	
11			fadd.d f11,f11, f5	fadd.d f12,f12, f6	
12					sltu x3,x1,x4
13	fsd f7, -48(x2)	fsd f8, -40(x2)			
14	fsd f9, -32(x2)	fsd f10, -24(x2)			
15	fsd f11 -16(x2)	fsd f12, -8(x2)			bnez x3, foo

15 cycles for 34 operations, yielding 2.67 issues per clock, with a VLIW efficiency of 34 operations for 75 slots = 45.3%. This schedule requires 12 floating-point registers.

Unrolled 10 times:

Cycle	Memory reference 1	Memory reference 2	fP operation 1	fP operation 2	Integer operation/ branch
1	fld f1, 0(x1)	fld f2, 8(x1)			
2	fld f3, 16(x1)	fld f4, 24(x1)			
3	fld f5, 32(x1)	fld f6, 40(x1)	fmul.d f1,f1,f0	fmul.d f2,f2,f0	
4	fld f7, 48(x1)	fld f8, 56(x1)	fmul.d f3,f3,f0	fmul.d f4,f4,f0	
5	fld f9, 64(x1)	fld f10, 72(x1)	fmul.d f5,f5,f0	fmul.d f6,f6,f0	
6	fld f11, 0(x2)	fld f12, 8(x2)	fmul.d f7,f7,f0	fmul.d f8,f8,f0	
7	fld f13, 16(x2)	fld f14, 24(x2)	fmul.d f9,f9,f0	fmul.d f10,f10, f0	addi x1,x1,48
8	fld f15, 32(x2)	fld f16, 40(x2)			addi x2,x2,48
9	fld f17, 48(x2)	fld f18, 56(x2)	fadd.d f11, f11,f1	fadd.d f12,f12, f2	
10	fld f19, 64(x2)	fld f20, 72(x2)	fadd.d f13, f13,f3	fadd.d f14,f14, f4	
11			fadd.d f15, f15,f5	fadd.d f16,f16, f6	
12			fadd.d f17, f17,f7	fadd.d f18,f18, f8	DSLTU x3,x1,x4
13	fsd f11, -80(x2)	fsd f12, -72(x2)	fadd.d f19, f19,f9	fadd.d f20,f20, f10	
14	fsd f13, -64(x2)	fsd f14, -56(x2)			

15	fsd f15 -48(x2)	fsd f16, -40(x2)			
16	fsd f17 -32(x2)	fsd f18, -24(x2)			
17	fsd f19 -16(x2)	fsd f20, -8(x2)			bnez x3, foo

17 cycles for 54 operations, yielding 3.18 issues per clock, with a VLIW efficiency of 54 operations for 85 slots = 63.5%. This schedule requires 20 floating-point registers.

3.15. a.

Iteration	Instruction	Issues at	Executes/memory	Write CDB at	Comment
1	fld F2,0(x1)	1	2	3	First issue
1	fmul.d F4,F2,F0	2	4	19	Wait for F2 Mult rs [3–4] Mult use [5–18]
1	fld F6,0(x2)	3	4	5	Ldbuf [4]
1	fadd.d F6,F4,F6	4	20	30	Wait for F4 Add rs [5–20] Add use [21–29]
1	fsd F6,0(x2)	5	31		Wait for F6 Stbuf1 [6–31]
1	addi x1,x1,#8	6	7	8	
1	addi x2,x2,#8	7	8	9	
1	sltu x3,x1,x4	8	9	10	
1	bnez x3,foo	9	11		Wait for x3
2	fld F2,0(x1)	10	12	13	Wait for bnez Ldbuf [11–12]
2	fmul.d F4,F2,F0	11	14 19	34	Wait for F2 Mult busy Mult rs [12–19] Mult use [20–33]
2	fld F6,0(x2)	12	13	14	Ldbuf [13]
2	fadd.d F6,F4,F6	13	35	45	Wait for F4 Add rs [14–35] Add use [36–44]
2	fsd F6,0(x2)	14	46		Wait for F6 Stbuf [15–46]
2	addi x1,x1,#8	15	16	17	
2	addi x2,x2,#8	16	17	18	
2	sltu x3,x1,x4	17	18	20	
2	bnez x3,foo	18	20		Wait for x3
3	fld F2,0(x1)	19	21	22	Wait for bnez Ldbuf [20–21]

3	fmul.d F4,F2,F0	20	23 34	49	Wait for F2 Mult busy Mult rs [21–34] Mult use [35–48]
3	fld F6,0(x2)	21	22	23	Ldbuf [22]
3	fadd.d F6,F4,F6	22	50	60	Wait for F4 Add rs [23–49] Add use [51–59]
3	fsd F6,0(x2)	23	55		Wait for F6 Stbuf [24–55]
3	addi x1,x1,#8	24	25	26	
3	addi x2,x2,#8	25	26	27	
3	sltu x3,x1,x4	26	27	28	
3	bnez x3,foo	27	29		Wait for x3

b.

Iteration	Instruction	Issues at	Executes/memory	Write CDB at	Comment
1	fld F2,0(x1)	1	2	3	
1	fmul.d F4,F2,F0	1	4	19	Wait for F2 Mult rs [2–4] Mult use [5]
1	fld F6,0(x2)	2	3	4	Ldbuf [3]
1	fadd.d F6,F4,F6	2	20	30	Wait for F4 Add rs [3–20] Add use [21]
1	fsd F6,0(x2)	3	31		Wait for F6 Stbuf [4–31]
1	addi x1,x1,#8	3	4	5	
1	addi x2,x2,#8	4	5	6	
1	sltu x3,x1,x4	4	6	7	INT busy INT rs [5–6]
1	bnez x3,foo	5	7		INT busy INT rs [6–7]
2	fld F2,0(x1)	6	8	9	Wait for BEQZ
2	fmul.d F4,F2,F0	6	10	25	Wait for F2 Mult rs [7–10] Mult use [11]
2	fld F6,0(x2)	7	9	10	INT busy INT rs [8–9]
2	fadd.d F6,F4,F6	7	26	36	Wait for F4 Add xS [8–26] Add use [27]
2	fsd F6,0(x2)	8	37		Wait for F6
2	addi x1,x1,#8	8	10	11	INT busy INT rs [8–10]

2	addi x2,x2,#8	9	11	12	INT busy INT rs [10–11]
2	sltu x3,x1,x4	9	12	13	INT busy INT rs [10–12]
2	bnez x3,foo	10	14		Wait for x3
3	fld F2,0(x1)	11	15	16	Wait for bnez
3	fmul.d F4,F2,F0	11	17	32	Wait for F2 Mult rs [12–17] Mult use [17]
3	fld F6,0(x2)	12	16	17	INT busy INT rs [13–16]
3	fadd.d F6,F4,F6	12	33	43	Wait for F4 Add rs [13–33] Add use [33]
3	fsd F6,0(x2)	14	44		Wait for F6 INT rs full in 15
3	addi x1,x1,#8	15	17		INT rs full and busy INT rs [17]
3	addi x2,x2,#8	16	18		INT rs full and busy INT rs [18]
3	sltu x3,x1,x4	20	21		INT rs full
3	bnez x3,foo	21	22		INT rs full

3.16.

Instruction	Issues at	Executes/memory	Write CDB at
fadd.d F2,F4,F6	1	2	12
add x1,x1,x2	2	3	4
add x1,x1,x2	3	5	6
add x1,x1,x2	4	7	8
add x1,x1,x2	5	9	10
add x1,x1,x2	6	11	12 (CDB conflict)

3.17. Correlating predictor

Branch PC mod 4	Entry	Prediction	Outcome	Mispredict?	Table update
2	4	T	T	No	None
3	6	NT	NT	No	Change to “NT”
1	2	NT	NT	No	None

3	7	NT	NT	No	None
1	3	T	NT	Yes	Change to “T with one misprediction”
2	4	T	T	No	None
1	3	T	NT	Yes	Change to “NT”
2	4	T	T	No	None
3	7	NT	T	Yes	Change to “NT with one misprediction”

Misprediction rate = 3/9 = 0.33

Local predictor

Branch PC mod 2	**Entry**	**Prediction**	**Outcome**	**Mispredict?**	**Table update**
0	0	T	T	no	Change to “T”
1	4	T	NT	yes	Change to “T with one misprediction”
1	1	NT	NT	no	None
1	3	T	NT	yes	Change to “T with one misprediction”
1	3	T	NT	yes	Change to “NT”
0	0	T	T	no	None
1	3	NT	NT	no	None
0	0	T	T	no	None
1	5	T	T	no	Change to “T”

Misprediction rate = 3/9 = 0.33

3.18. For this problem we are given the base CPI without branch stalls. From this we can compute the number of stalls given by no BTB and with the BTB: $\text{CPI}_{\text{noBTB}}$ and CPI_{BTB} and the resulting speedup given by the BTB:

$$\text{Speedup} = \frac{\text{CPI}_{\text{noBTB}}}{\text{CPI}_{\text{BTB}}} = \frac{\text{CPI}_{\text{base}} + \text{Stalls}_{\text{base}}}{\text{CPI}_{\text{base}} + \text{Stalls}_{\text{BTB}}}$$

$$\text{Stalls}_{\text{noBTB}} = 15\% \times 2 = 0.30$$

To compute $\text{Stalls}_{\text{BTB}}$, consider the following table:

BTB result	**BTB prediction**	**Frequency (per instruction)**	**Penalty (cycles)**
Miss		15% × 10% = 1.5%	3
Hit	Correct	15% × 90% × 90% = 12.1%	0
Hit	Incorrect	15% × 90% × 10% = 1.3%	4

Therefore:

$$\text{Stalls}_{\text{BTB}} = (1.5\% \times 3) + (12.1\% \times 0) + (1.3\% \times 4) = 1.2$$

$$\text{Speed up} = \frac{1.0 + 0.30}{1.0 + 0.097} = 1.2$$

3.19. a. Storing the target instruction of an unconditional branch effectively removes one instruction. If there is a BTB hit in instruction fetch and the target instruction is available, then that instruction is fed into decode in place of the branch instruction. The penalty is −1 cycle. In other words, it is a performance gain of 1 cycle.

b. If the BTB stores only the target address of an unconditional branch, fetch has to retrieve the new instruction. This gives us a CPI term of 5% x (90% × 0 + 10% × 2) of 0.01. The term represents the CPI for unconditional branches (weighted by their frequency of 5%). If the BTB stores the target instruction instead, the CPI term becomes 5% x (90% × (−1) + 10% × 2) or −0.035. The negative sign denotes that it reduces the overall CPI value. The hit percentage to just break even is simply 20%.

Chapter 4 Solutions

Case Study: Implementing a Vector Kernel on a Vector Processor and GPU

4.1 RISC-V code (answers may vary)

```
      li      x1, #0              # initialize k
loop: flw     f0, 0 (RtipL)       # load all values for first expression
      flw     f1, 0 (RclL)
      flw     f2, 4 (RtipL)
      flw     f3, 4 (RclL)
      flw     f4, 8 (RtipL)
      flw     f5, 8 (RclL)
      flw     f6, 12 (RtipL)
      flw     f7, 12 (RclL)
      flw     f8, 0 (RtipR)
      flw     f9, 0 (RclR)
      flw     f10, 4 (RtipR)
      flw     f11, 4 (RclR)
      flw     f12, 8 (RtipR)
      flw     f13, 8 (RclR)
      flw     f14, 12 (RtipR)
      flw     f15, 12 (RclR)
      fmul.s   f16, f0, f1         # first four multiplies
      fmul.s   f17, f2, f3
      fmul.s   f18, f4, f5
      fmul.s   f19, f6, f7
      fadd.s   f20, f16, f17      # accumulate
      fadd.s   f20, f20, f18
      fadd.s   f20, f20, f19
      fmul.s   f16, f8, f9         # second four multiplies
      fmul.s   f17, f10, f11
      fmul.s   f18, f12, f13
      fmul.s    f19, f14, f15
      fadd.s    f21, f16, f17       # accumulate
      fadd.s  f21, f21, f18
      fadd.s  f21, f21, f19
      fmul.s  f20, f20, f21       # final multiply
      fsw     f20, 0 (RclP)       # store result
      add     RclP, RclP, 4       # increment clP for next  expression
      addi     x1, x1, 1
      and     x2, x2, #3          # check to see if we should
                                    increment clL and clR (every
                                    4 bits)
```

```
           bneq     x2, skip
     skip: blt      x1, x3, loop # assume r3 = seq_length * 4
```

RV64V code (answers may vary)

```
           li       x1, 0                # initialize k
           vcfgd    10 * FP32             # enable 10 SP FP vregs
     loop: vld      v0, 0 (RclL)
           vld      v1, 0 (RclR)
           vld      v2, 0 (RtipL)        # load all tipL values
           vld      v3, 16 (RtipL)
           vld      v4, 32 (RtipL)
           vld      v5, 48 (RtipL)
           vld      v6, 0 (RtipR)         # load all tipR values
           vld      v7, 16 (RtipR)
           vld      v8, 32 (RtipR)
           vld      v9, 48 (RtipR)
           vmul v2, v2, v0             # multiply left sub-expressions
           vmul v3, v3, v0
           vmul v4, v4, v0
           vmul v5, v5, v0
           vmul v6, v6, v1             # multiply right sub-expression
           vmul v7, v7, v1
           vmul v8, v8, v1
           vmul v9, v9, v1
           vsum f0, v2                # reduce left sub-expressions
           vsum f1, v3
           vsum f2, v4
           vsum f3, v5
           vsum f4, v6                # reduce right sub-expressions
           vsum f5, v7
           vsum f6, v8
           vsum f7, v9
           fmul.s  f0, f0, f4            # multiply left and right sub-expressions
           fmul.s  f1, f1, f5
           fmul.s  f2, f2, f6
           fmul.s  f3, f3, f7
           fsw  f0, 0 (Rclp)          # store results
           fsw  f1, 4 (Rclp)
           fsw  f2, 8 (Rclp)
           fsw  f3, 12 (Rclp)
           add  RclP, RclP, 16        # increment clP for next expression
           add  RclL, RclL, 16        # increment clL for next expression
           add  RclR, RclR, 16        # increment clR for next expression
           addi  x1, x1, 1
           blt  x1, x3, loop     # assume x3 = seq_length
```

4.2 RISC-V: loop is 39 instructions, will iterate $500 \times 4 = 2000$ times, so roughly 78,000 instructions.

RV64V: loop is also 39 instructions, but will iterate only 500 times, so roughly 19,500 instructions.

4.3
```
1.    vld               # clL
2.    vld               # clR
3.    vld      vmul     # tiPL 0
4.    vld      vmul     # tiPL 1
5.    vld      vmul     # tiPL 2
6.    vld      vmul     # tiPL 3
7.    vld      vmul     # tiPR 0
8.    vld      vmul     # tiPR 1
9.    vld      vmul     # tiPR 2
10.   vld      vmul     # tiPR 3
11.   vsum
12.   vsum
13.   vsum
14.   vsum
15.   vsum
16.   vsum
17.   vsum
18.   vsum
```

18 chimes, 4 results, 15 FLOPS per result, $18/15 = 1.2$ cycles per FLOP

4.4 In this case we can perform, for example, four parallel instances of the eight multiplies in each line of the for-loop, but it would require that we be able to perform reductions on a subset of elements of the product vector.

4.5
```
__global__ void compute_condLike (float *clL, float *clR, float
*clP, float *tiPL, float *tiPR) {
  int i,k = threadIdx.x;
  __shared__ float clL_s[4], clR_s[4];
for (i=0;i<4;i++) {
    clL_s[i]=clL[k*4+i];
    clR_s[i]=clR[k*4+i];
  }
    clP[k*4] = (tiPL[k+AA]*clL_s[A] +
tiPL[k+AC]*clL_s[C] + tiPL[k+AG]*clL_s[G] +
tiPL[k+AT]*clL_s[T])*(tiPR[k+AA]*clR_s[A] +
tiPR[k+AC]*clR_s[C] + tiPR[k+AG]*clR_s[G] +
tiPR[k+AT]*clR_s[T]);
    clP[k*4+1] = (tiPL[k+CA]*clL_s[A] +
tiPL[k+CC]*clL_s[C] + tiPL[k+CG]*clL_s[G] +
tiPL[k+CT]*clL_s[T])*(tiPR[k+CA]*clR_s[A] +
tiPR[k+CC]*clR_s[C] + tiPR[k+CG]*clR_s[G] +
tiPR[k+CT]*clR_s[T]);
```

```
    clP[k*4+2] = (tiPL[k+GA]*clL_s[A] +
tiPL[k+GC]*clL_s[C] + tiPL[k+GG]*clL_s[G] +
tiPL[k+GT]*clL_s[T])*(tiPR[k+GA]*clR_s[A] +
tiPR[k+GC]*clR_s[C] + tiPR[k+GG]*clR_s[G] +
tiPR[k+GT]*clR_s[T]);
    clP[k*4+3] = (tiPL[k+TA]*clL_s[A] +
tiPL[k+TC]*clL_s[C] + tiPL[k+TG]*clL_s[G] +
tiPL[k+TT]*clL_s[T])*(tiPR[k+TA]*clR_s[A] +
tiPR[k+TC]*clR_s[C] + tiPR[k+TG]*clR_s[G] +
tiPR[k+TT]*clR_s[T]);
}
```

4.6

```
clP[threadIdx.x*4 + blockIdx.x+12*500*4]
clP[threadIdx.x*4+1 + blockIdx.x+12*500*4]
clP[threadIdx.x*4+2+ blockIdx.x+12*500*4]
clP[threadIdx.x*4+3 + blockIdx.x+12*500*4]
clL[threadIdx.x*4+i+ blockIdx.x*2*500*4]
clR[threadIdx.x*4+i+ (blockIdx.x*2+1)*500*4]
tipL[threadIdx.x+AA + blockIdx.x*2*500]
tipL[threadIdx.x+AC + blockIdx.x*2*500]
...
tipL[threadIdx.x+TT + blockIdx.x*2*500]
tipR[threadIdx.x+AA + (blockIdx.x*2+1)*500]
tipR[threadIdx.x+AC +1 + (blockIdx.x*2+1)*500]
...
tipR[threadIdx.x+TT +15+ (blockIdx.x*2+1)*500]
```

4.7

```
                                        # compute address of clL
mul.u64          %r1, %ctaid.x, 4000    # multiply block index by 4000
mul.u64          %r2, %tid.x, 4         # multiply thread index by 4
add.u64          %r1, %r1, %r2          # add products
ld.param.u64     %r2, [clL]             # load base address of clL
add.u64          %r1, %r2, %r2          # add base to offset
                                        # compute address of clR
add.u64          %r2, %ctaid.x, 1       # add 1 to block index
mul.u64          %r2, %r2, 4000         # multiply by 4000
mul.u64          %r3, %tid.x, 4         # multiply thread index by 4
add.u64          %r2, %r2, %r3          # add products
ld.param.u64     %r3, [clR]             # load base address of clR
add.u64          %r2, %r2, %r3          # add base to offset
ld.global.f32    %f1, [%r1+0]           # move clL and clR into shared memory
st.shared.f32    [clL_s+0], %f1         # (unroll the loop)
ld.global.f32    %f1, [%r2+0]
st.shared.f32    [clR_s+0], %f1
ld.global.f32    %f1, [%r1+4]
```

```
st.shared.f32       [clL_s+4], %f1
ld.global.f32       %f1, [%r2+4]
st.shared.f32       [clR_s+4], %f1
ld.global.f32       %f1, [%r1+8]
st.shared.f32       [clL_s+8], %f1
ld.global.f32       %f1, [%r2+8]
st.shared.f32       [clR_s+8], %f1
ld.global.f32       %f1, [%r1+12]
st.shared.f32       [clL_s+12], %f1
ld.global.f32       %f1, [%r2+12]
st.shared.f32       [clR_s+12], %f1
                                                 # compute address of tiPL:
mul.u64             %r1, %ctaid.x, 16000     # multiply block index by 4000
mul.u64             %r2, %tid.x, 64          # multiply thread index by 16 floats
add.u64             %r1, %r1, %r2            # add products
ld.param.u64        %r2, [tipL]              # load base address of tipL
add.u64             %r1, %r2, %r2            # add base to offset
add.u64             %r2, %ctaid.x, 1         # compute address of tiPR:
mul.u64             %r2, %r2, 16000          # multiply block index by 4000
mul.u64             %r3, %tid.x, 64          # multiply thread index by 16 floats
add.u64             %r2, %r2, %r3            # add products
ld.param.u64        %r3, [tipR]              # load base address of tipL
add.u64             %r2, %r2, %r3            # add base to offset
                                                 # compute address of clP:
mul.u64             %r3, %r3, 24,000         # multiply block index by 4000
mul.u64             %r4, %tid.x, 16          # multiply thread index by 4 floats
add.u64             %r3, %r3, %r4            # add products
ld.param.u64        %r4, [tipR]              # load base address of tipL
add.u64             %r3, %r3, %r4            # add base to offset
ld.global.f32       %f1, [%r1]               # load tiPL[0]
ld.global.f32       %f2, [%r1+4]             # load tiPL[1]
...
ld.global.f32       %f16, [%r1+60]           # load tiPL[15]
ld.global.f32       %f17, [%r2]              # load tiPR[0]
ld.global.f32       %f18, [%r2+4]            # load tiPR[1]
...
ld.global.f32       %f32, [%r1+60]           # load tiPR[15]
ld.shared.f32       %f33, [clL_s]            # load clL
ld.shared.f32       %f34, [clL_s+4]
ld.shared.f32       %f35, [clL_s+8]
ld.shared.f32       %f36, [clL_s+12]
ld.shared.f32       %f37, [clR_s]            # load clR
ld.shared.f32       %f38, [clR_s+4]
ld.shared.f32       %f39, [clR_s+8]
ld.shared.f32       %f40, [clR_s+12]
mul.f32             %f1, %f1, %f33           # first expression
mul.f32             %f2, %f2, %f34
mul.f32             %f3, %f3, %f35
mul.f32             %f4, %f4, %f36
add.f32             %f1, %f1, %f2
add.f32             %f1, %f1, %f3
add.f32             %f1, %f1, %f4
mul.f32             %f17, %f17, %f37
mul.f32             %f18, %f18, %f38
mul.f32             %f19, %f19, %f39
mul.f32             %f20, %f20, %f40
add.f32             %f17, %f17, %f18
add.f32             %f17, %f17, %f19
add.f32             %f17, %f17, %f20
st.global.f32       [%r3], %f17              # store result
mul.f32%            f5, %f5, %f33            # second expression
mul.f32             %f6, %f6, %f34
mul.f32             %f7, %f7, %f35
mul.f32             %f8, %f8, %f36
add.f32             %f5, %f5, %f6
add.f32             %f5, %f5, %f7
add.f32             %f5, %f5, %f8
mul.f32             %f21, %f21, %f37
mul.f32             %f22, %f22, %f38
mul.f32             %f23, %f23, %f39
mul.f32             %f24, %f24, %f40
add.f32             %f21, %f21, %f22
add.f32             %f21, %f21, %f23
add.f32             %f21, %f21, %f24
st.global.f32       [%r3+4], %f21            # store result
mul.f32             %f9, %f9, %f33           # third expression
mul.f32             %f10, %f10, %f34
mul.f32             %f11, %11, %f35
mul.f32             %f12, %f12, %f36
add.f32             %f9, %f9, %f10
add.f32             %f9, %f9, %f11
add.f32             %f9, %f9, %f12
mul.f32             %f25, %f25, %f37
mul.f32             %f26, %f26, %f38
mul.f32             %f27, %f27, %f39
mul.f32             %f28, %f28, %f40
add.f32             %f25, %f26, %f22
add.f32             %f25, %f27, %f23
add.f32             %f25, %f28, %f24
```

```
st.global.f32       [%r3+8], %f25           # store result
mul.f32             %f13, %f13, %f33        # fourth expression
mul.f32             %f14, %f14, %f34
mul.f32             %f15, %f15, %f35
mul.f32             %f16, %f16, %f36
add.f32             %f13, %f14, %f6
add.f32             %f13, %f15, %f7
add.f32             %f13, %f16, %f8
mul.f32             %f29, %f29, %f37
mul.f32             %f30, %f30, %f38
mul.f32             %f31, %f31, %f39
mul.f32             %f32, %f32, %f40
add.f32             %f29, %f29, %f30
add.f32             %f29, %f29, %f31
add.f32             %f29, %f29, %f32
st.global.f32       [%r3+12], %f29          # store result
```

4.8 It will perform well, because there are no branch divergences, all memory references are coalesced, and there are 500 threads spread across 6 blocks (3000 total threads), which provides many instructions to hide memory latency.

Exercises

4.9 a. This code reads four floats and writes two floats for every six FLOPs, so arithmetic intensity = 6/6 = 1.

b. Assume MVL = 64:

```
      li      x1, 0           # initialize index
loop: vld     v1, a_re+r1     # load a_re
      vld     v3, b_re+r1     # load b_re
      vmul    v5, v1,v3       # a+re*b_re
      vld     v2, a_im+r1     # load a_im
      vld     v4, b_im+r1     # load b_im
      vmul    v6, v2, v4      # a+im*b_im
      vsub    v5, v5, v6      # a+re*b_re - a+im*b_im
      sv      v5, c_re+r1     # store c_re
      vmul    v5, v1, v4      # a+re*b_im
      vmul    v6, v2, v3      # a+im*b_re
      addvv.s v5, v5, v6      # a+re*b_im + a+im*b_re
      vst     v5, c_im+r1     # store c_im
      bne     x1, 0, else     # check if first iteration
      addi    x1, x1, 44      # first iteration, increment by 44
      j loop                  # guaranteed next iteration
else: addi    x1, x1, 256     # not first iteration, increment by 256
skip: blt     x1, 1200, loop  # next iteration?
```

c.
```
1. vmul     vld    # a_re * b_re (assume already
                   # loaded), load a_im
2. vld      vmul   # load b_im, a_im*b_im
3. vsub     vst    # subtract and store c _re
4. vmul     vld    # a_re*b_im, load next a_re vector
5. vmul     vld    # a_im*b_re, load next b_re vector
6. vadd     vst    # add and store c_im
```

6 chimes

d. total cycles per iteration = 6 chimes × 64 elements + 15 cycles (load / store) × 6 +8 cycles (multiply) × 4 + 5 cycles (add/subtract) × 2 = 516 cycles per result = 516/128 = 4

e.
```
1. vmul                         # a_re*b_re
2. vmul                         # a_im*b_im
3. vsub vst                     # subtract and store c_re
4. vmul                         # a_re*b_im
5. vmul vld                     # a_im*b_re, load next a_re
6. addvv.s vst vld vld vld      # add, store c_im, load next b_re, a_im, b_im
```

Same cycles per result as in part c. Adding additional load / store units did not improve performance.

4.10 Vector processor requires:

- (200 MB + 100 MB)/(30 GB/s) = 10 ms for vector memory access +
- 400 ms for scalar execution.

Assuming that vector computation can be overlapped with memory access, total time = 410 ms.

The hybrid system requires:

- (200 MB + 100 MB)/(150 GB/s) = 2 ms for vector memory access +
- 400 ms for scalar execution +
- (200 MB + 100 MB)/(10 GB/s) = 30 ms for host I/O

Even if host I/O can be overlapped with GPU execution, the GPU will require 430 ms and therefore will achieve lower performance than the host.

4.11 a.
```
for (i=0;i<32;i+=2) dot[i] = dot[i]+dot[i+1];
for (i=0;i<16;i+=4) dot[i] = dot[i]+dot[i+2];
for (i=0;i<8;i+=8) dot[i] = dot[i]+dot[i+4];
for (i=0;i<4;i+=16) dot[i] = dot[i]+dot[i+8];
for (i=0;i<2;i+=32) dot[i] = dot[i]+dot[i+16];
dot[0]=dot[0]+dot[32];
```

b.
```
vadd  v0(0), v0(4)
vadd  v0(8), v0(12)
vadd  v0(16), v0(20)
vadd  v0(24), v0(28)
vadd  v0(32), v0(36)
vadd  v0(40), v0(44)
vadd  v0(48), v0(52)
vadd  v0(56), v0(60)
```

c.
```
for (unsigned int s= blockDim.x/2;s>0;s/=2) {
if (tid<s) sdata[tid]=sdata[tid]+sdata[tid+s];
   __syncthreads();
 }
```

4.12 a. Reads 40 bytes and writes 4 bytes for every 8 FLOPs, thus 8/44 FLOPs/byte.

b. This code performs indirect references through the Ca and Cb arrays, as they are indexed using the contents of the IDx array, which can only be performed at runtime. While this complicates SIMD implementation, it is still possible to perform the type of indexing using gather-type load instructions. The innermost loop (iterates on z) can be vectorized: the values for Ex, dH1, dH2, Ca, and Cb could be operated on as SIMD registers or vectors. Thus, this code is amenable to SIMD and vector execution.

c. Having an arithmetic intensity of 0.18, if the processor has a peak floating-point throughout $>$ (30 GB/s) $\times$ (0.18 FLOPs/byte) $=$ 5.4 GFLOPs/s, then this code is likely to be memory-bound, unless the working set fits well within the processor's cache.

d. The single precision arithmetic intensity corresponding to the edge of the roof is 85/4 $=$ 21.25 FLOPs/byte.

4.13 a. 1.5 GHz $\times$ 0.80 $\times$ 0.85 $\times$ 0.70 $\times$ 10 cores $\times$ 32/4 $=$ 57.12 GFLOPs/s

b. **Option 1:**
1.5 GHz $\times$ 0.80 $\times$ 0.85 $\times$ 0.70 $\times$ 10 cores $\times$ 32/2 $=$ 114.24 GFLOPs/s
(speedup $=$ 114.24/57.12 $=$ 2)
Option 2:
1.5 GHz $\times$ 0.80 $\times$ 0.85 $\times$ 0.70 $\times$ 15 cores $\times$ 32/4 $=$ 85.68 GFLOPs/s
(speedup $=$ 85.68/57.12 $=$ 1.5)
Option 3:
1.5 GHz $\times$ 0.80 $\times$ 0.95 $\times$ 0.70 $\times$ 10 cores $\times$ 32/4 $=$ 63.84 GFLOPs/s
(speedup $=$ 63.84/57.12 $=$ 1.11)
Option 3 is best.

4.14 a. Using the GCD test, a dependency exists if GCD (2,4) must divide 5 $-$ 4. In this case, a loop-carried dependency does exist.

b. *Output dependencies*
S1 and S3 cause through A[i]
Anti-dependencies
S4 and S3 cause an anti-dependency through C[i]
Re-written code

```
for (i=0;i<100;i++) {
  T[i] = A[i] * B[i]; /* S1 */
  B[i] = T[i] + c; /* S2 */
  A1[i] = C[i] * c; /* S3 */
  C1[i] = D[i] * A1[i]; /* S4 */}
```

True dependencies
S4 and S3 through A[i]
S2 and S1 through T[i]

c. There is an anti-dependence between iteration i and i + 1 for array B. This can be avoided by renaming the B array in S2.

4.15 a. Branch divergence: causes SIMD lanes to be masked when threads follow different control paths.

b. Covering memory latency: a sufficient number of active threads can hide memory latency and increase instruction issue rate.

c. Coalesced off-chip memory references: memory accesses should be organized consecutively within SIMD thread groups.

d. Use of on-chip memory: memory references with locality should take advantage of on-chip memory, references to on-chip memory within a SIMD thread group should be organized to avoid bank conflicts.

4.16 This GPU has a peak throughput of $1.5 \times 16 \times 16 = 384$ GFLOPS/s of single-precision throughput. However, assuming each single precision operation requires four-byte two operands and outputs one four-byte result, sustaining this throughput (assuming no temporal locality) would require 12 bytes/FLOP $\times$ 384 GFLOPs/s = 4.6 TB/s of memory bandwidth. As such, this throughput is not sustainable, but can still be achieved in short bursts when using on-chip memory.

4.17 Reference code for programming exercise:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <cuda.h>
__global__ void life (unsigned char *d_board,int iterations) {
  int i,row,col,rows,cols;
  unsigned char state,neighbors;
  row = blockIdx.y * blockDim.y + threadIdx.y;
  col = blockIdx.x * blockDim.x + threadIdx.x;
  rows = gridDim.y * blockDim.y;
  cols = gridDim.x * blockDim.x;
  state = d_board[(row)*cols+(col)];
  for (i=0;i<iterations;i++) {
    neighbors=0;
     if (row!=0) {
      if (col!=0) if (d_board[(row-1)*cols+(col-1)]==1) neighbors++;
      if (d_board[(row-1)*cols+(col)]==1) neighbors++;
      if(col!=(cols-1))if(d_board[(row-1)*cols+(col+1)]==1)neighbors++;
     }
```

```
      if (col!=0) if (d_board[(row)*cols+(col-1)]==1) neighbors++;
      if(col!=(cols-1))if(d_board[(row)*cols+(col+1)]==1)neighbors++;
      if (row!=(rows-1)) {
       if (col!=0) if (d_board[(row+1)*cols+(col-1)]==1) neighbors++;
       if (d_board[(row+1)*cols+(col)]==1) neighbors++;
       if(col!=(cols-1))if(d_board[(row+1)*cols+(col+1)]==1)neighbors++;
      }
      if (neighbors<2) state = 0;
      else if (neighbors==3) state = 1;
      else if (neighbors>3) state = 0;
      __syncthreads();
      d_board[(row)*cols+(col)]=state;
   }
}
int main () {
   dim3 gDim,bDim;
   unsigned char *h_board,*d_board;
   int i,iterations=100;
   bDim.y=16;
   bDim.x=32;
   bDim.z=1;
   gDim.y=16;
   gDim.x=8;
   gDim.z=1;
   h_board=(unsigned char *)malloc(sizeof(unsigned char)*4096*4096);
   cudaMalloc((void **)&d_board,sizeof(unsigned char)*4096*4096);
   srand(56);
   for (i=0;i<4096*4096;i++) h_board[i]=rand()%2;
cudaMemcpy(d_board,h_board,sizeof(unsigned char)*4096*4096,
cudaMemcpyHostToDevice);
life <<<gDim,bDim>>> (d_board,iterations);
cudaMemcpy(h_board,d_board,sizeof(unsigned char)*4096*4096,
cudaMemcpyDeviceToHost);
free(h_board);
cudaFree(d_board);
}
```

Chapter 5 Solutions

Case Study 1: Single-Chip Multicore Multiprocessor

5.1 Cx.y is cache line y in core x.

a. C0: R AC20	$\rightarrow$	C0.0: (S, AC20, 0020), returns 0020
b. C0: W AC20 $\leftarrow$ 80	$\rightarrow$	C0.0: (M, AC20, 0080) C3.0: (I, AC20, 0020)
c. C3: W AC20 $\leftarrow$ 80	$\rightarrow$	C3.0: (M, AC20, 0080)
d. C1: R AC10	$\rightarrow$	C1.2: (S, AC10, 0010) returns 0010
e. C0: W AC08 $\leftarrow$ 48	$\rightarrow$	C0.1 (M, AC08, 0048) C3.1: (I, AC08, 0008)
f. C0: W AC30 $\leftarrow$ 78	$\rightarrow$	C0.2: (M, AC30, 0078) M: AC10 $\leftarrow$ 0030 (write-back to memory)
g. C3: W AC30 $\leftarrow$ 78	$\rightarrow$	C3.2 :(M, AC30, 0078)

5.2 a. C0: R AC20 Read miss, satisfied by memory
C0: R AC28 Read miss, satisfied by C1's cache
C0: R AC20 Read miss, satisfied by memory, write-back 110
Implementation 1: 100 + 40+ 10 + 100 + 10 = 260 stall cycles
Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles

b. C0: R AC00 Read miss, satisfied by memory
C0: W AC08 $\leftarrow$ 48 Write hit, sends invalidate
C0: W AC20 $\leftarrow$ 78 Write miss, satisfied by memory, write back 110
Implementation 1: 100 + 15 + 10 + 100 = 225 stall cycles
Implementation 2: 100 + 15 + 10 + 100 = 225 stall cycles

c. C1: R AC20 Read miss, satisfied by memory
C1: R AC28 Read hit
C1: R AC20 Read miss, satisfied by memory
Implementation 1: 100 + 0 + 100 = 200 stall cycles
Implementation 2: 100 + 0 + 100 = 200 stall cycles

d. C1: R AC00 Read miss, satisfied by memory
C1: W AC08 $\leftarrow$ 48 Write miss, satisfied by memory, write back AC28
C1: W AC20 $\leftarrow$ 78 Write miss, satisfied by memory
Implementation 1: 100 + 100 + 10 + 100 = 310 stall cycles
Implementation 2: 100 + 100 + 10 + 100 = 310 stall cycles

5.3 See Figure S.1

5.4 a. C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W AC00 $\leftarrow$ 40 MSI: send invalidate, MESI: silent transition from E to M
MSI: 100 + 15 = 115 stall cycles
MESI: 100 + 0 = 100 stall cycles

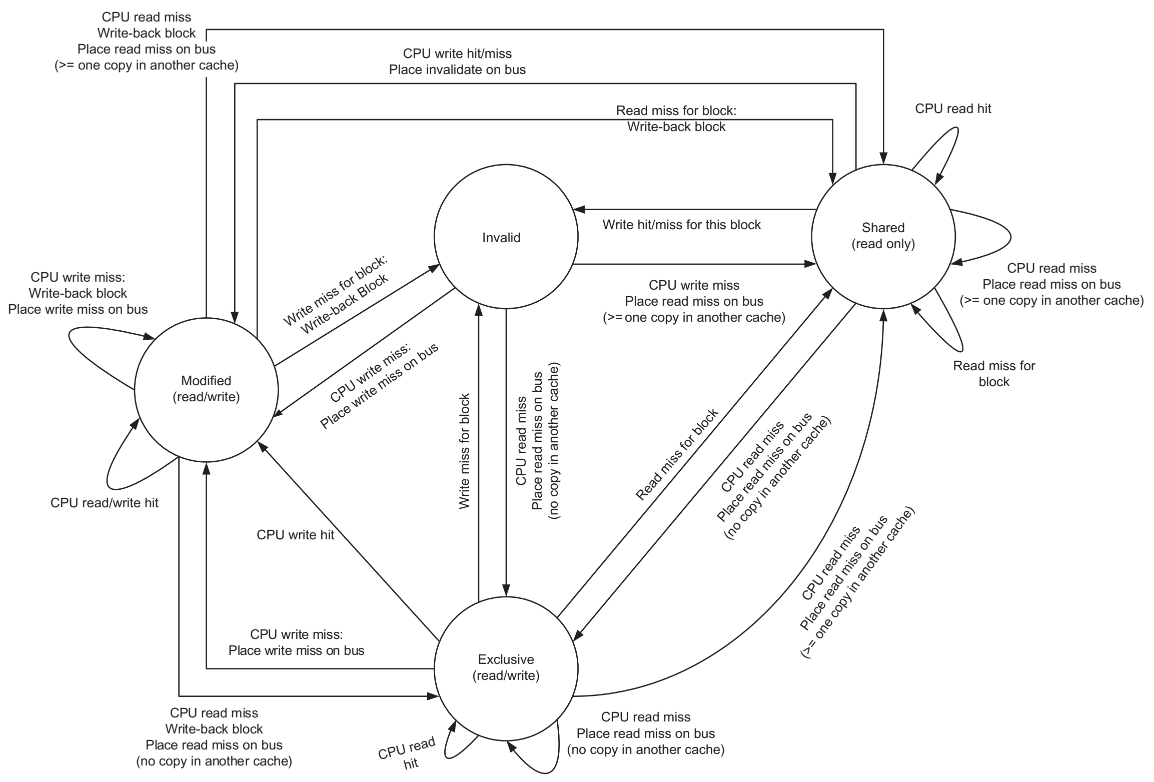


Figure S.1 Protocol diagram.

b. C0: R AC20, Read miss, satisfied in memory, sharers both to S
C0: W AC20 ← 60 both send invalidates
Both: 100 + 15 = 115 stall cycles

c. C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: R AC20, Read miss, memory, silently replace 120 from S or E
Both: 100 + 100 = 200 stall cycles, silent replacement from E

d. C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C1: W AC00 ← 60, Write miss, satisfied in memory regardless of protocol
Both: 100 + 100 = 200 stall cycles, don't supply data in E state (some protocols do)

e. C0: R AC00, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W AC00 ← 60, MSI: send invalidate, MESI: silent transition from E to M
C1: W AC00 ← 40, Write miss, C0's cache, write-back data to memory
MSI: 100 + 15 + 40 + 10 = 165 stall cycles
MESI: 100 + 0 + 40 + 10 = 150 stall cycles

5.5 **Loop 1**

```
Repeat i: 1 .. n
    A[i] ← A[i-1] +B[i];
```

Loop2

```
Repeat i: 1 .. n
    A[i] ← A[i] +B[i];
```

If A, B, are larger than the cache and n is large enough, the hit/miss pattern (running on one CPU) for both loops for **large values of i** is shown in the table (hit times ignored).

Cache/ memory accesses	**Loop1**							**Loop2**						
	A[i]		**A[i-1]**		**B[i]**		**Total**	**A[i]**		**A[i]**		**B[i]**		**Total**
No coherence protocol	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit	–	Read miss + writeback	110 cycles	Read miss	110 cycles	220 cycles
MESI	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit	–	Read miss + writeback	110 cycles	Read miss	110 cycles	220 cycles
MSI	Write miss + writeback	110 cycles	Read hit	–	Read miss	100 cycles	210 cycles	Write hit + invalidate	15	Read miss + writeback	110 cycles	Read miss	110 cycles	235 cycles

When the cache line is large enough to contain multiple elements—M, the average cost of the memory accesses (ignoring cache hits) will be divided by M. When hits and non-memory accessing instructions are considered, the relative performance of Loop1 and Loop2 will get closer to 1.

5.6 See Figure S.2. To make the case clearer for the MOESI protocol, we require that the cache that owns a modified line supply to other caches on a miss.

5.7 Assuming data provided from other core's cache whenever possible

a. C1: R, AC10: C0 cache writeback + C1 cache miss $\rightarrow$ 40 +10 = 50 cycles for implementation 1.
$\rightarrow$ 130 +10 = 140 cycles for implementation 2
C3: R, AC10 C3 cache miss $\rightarrow$ 40 cycles for implementation 1.
$\rightarrow$ 130 cycles for implementation 2.
C0: R, AC10 C0 cache hit $\rightarrow$ 0 extra cycles for both implementations 1 & 2.

Total:
Implementation 1: 90 cycles, Implementation 2: 270 cycles
(Results same for MSI and MESI)

b. C1: R, AC20 : C1 cache miss $\rightarrow$ 40 cycles for implementation 1.
$\rightarrow$ 130 cycles for implementation 2.
C3: R, AC20: C3 cache hit $\rightarrow$ 0 extra cycles for both implementations 1 & 2.
C0: R, AC20: C0 cache miss $\rightarrow$ 40 cycles for implementation 1.
$\rightarrow$ 130 cycles for implementation 2.

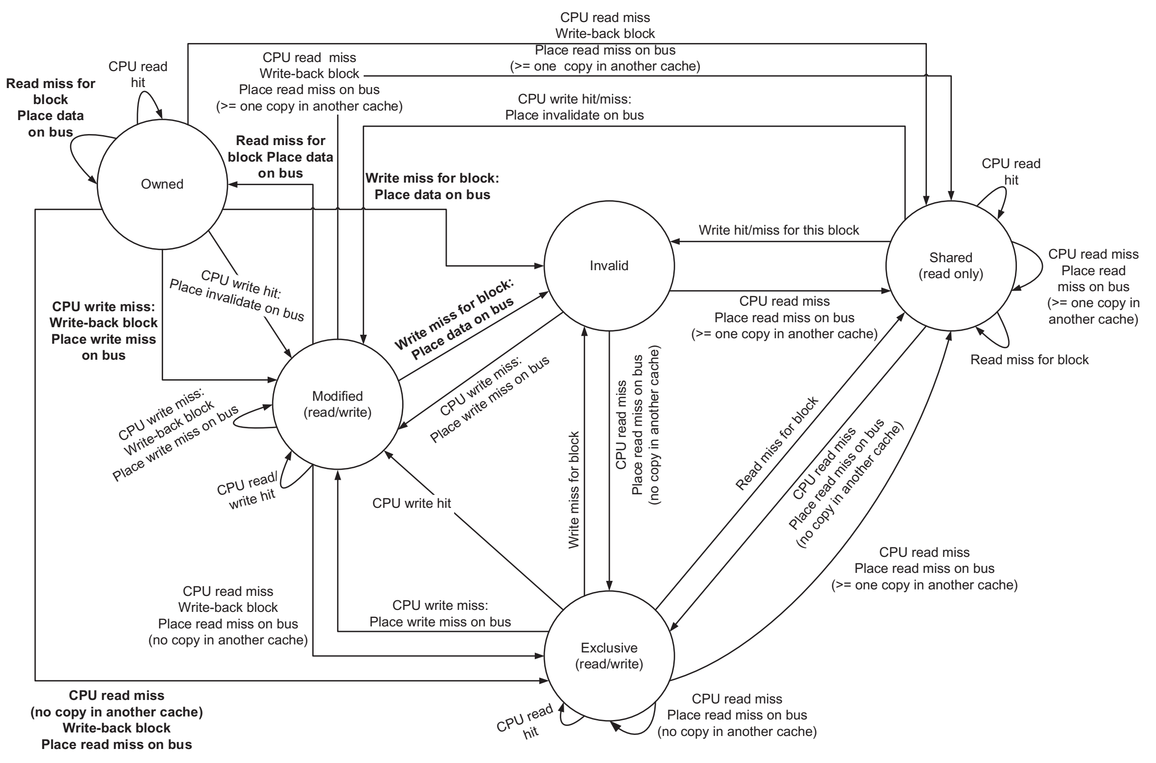

Owned
Modified (read/write)
Invalid
Shared (read only)
Exclusive (read/write)
CPU read hit
Read miss for block Place data on bus
CPU read miss Write-back block Place read miss on bus (>= one copy in another cache)
CPU write hit/miss: Place invalidate on bus
Read miss for block Place data on bus
Write miss for block: Place data on bus
Write hit/miss for this block
CPU read miss Place read miss on bus (>= one copy in another cache)
Read miss for block
CPU write hit: Place invalidate on bus
CPU write miss: Write-back block Place write miss on bus
CPU read/ write hit
CPU write miss: Place write miss on bus
CPU write hit
Write miss for block
CPU read miss Place read miss on bus (no copy in another cache)
Read miss for block
CPU read miss Write-back block Place read miss on bus (no copy in another cache)
CPU read miss (no copy in another cache) Write-back block Place read miss on bus

Figure S.2 Diagram for a MOESI protocol.

Total:
Implementation 1: 80 cycles, Implementation 2: 260 cycles
(Results same for MSI and MESI)

c. C0: W, AC20 ← 80: C0
cache miss + C3 (S → I) → 40 cycles for implementation 1.
→ 130 cycles for implementation 2.
C3: R, AC20: C0 (M → S)
writeback + C3 miss → 10 + 40 = 50 cycles for implementation 1.
→ 10 + 130 = 140 cycles for implementation 2.
C0: R, AC20: C0 hit → 0 extra cycles for both implementations 1 & 2.

Total:
Implementation 1: 90 cycles, Implementation 2: 270 cycles
(Results same for MSI and MESI)

d. C0: W, AC08 ← 88 C0 write hit – send inv (S → M), C3 (S → I) → 15 cycles for both implementations 1 & 2.

C3: R, AC08 C0 writeback
(M → S), C3 cache miss → 10 + 40 = 50 cycles implementation 1.
→ 10 + 130 = 140 cycles implementation 2.
C0: W, AC08 ← 98 C0 write hit – send inv (S → M), C3 (S → I) → 15 cycles for both implementations 1 & 2.

Total:
Implementation 1: 80 cycles, Implementation 2: 170 cycles (Results same for MSI and MESI)

Difference between MSI and MESI would show if there is a read miss followed by a write hit with no intervening accesses from other cores.

5.8 a. Assume the processors acquire the lock in order. C0 will acquire it first, incurring 100 stall cycles to retrieve the block from memory. C1 and C3 will stall until C0's critical section ends (ping-ponging the block back and forth) 1000 cycles later. C0 will stall for (about) 40 cycles while it fetches the block to invalidate it; then C1 takes 40 cycles to acquire it. C1's critical section is 1000 cycles, plus 40 to handle the write miss at release. Finally, C3 grabs the block for a final 40 cycles of stall. So, C0 stalls for 100 cycles to acquire, 10 to give it to C1, 40 to release the lock, and a final 10 to hand it off to C1, for a total of 160 stall cycles. C1 essentially stalls until C0 releases the lock, which will be 100 + 1000 + 10 + 40 = 1150 cycles, plus 40 to get the lock, 10 to give it to C3, 40 to get it back to release the lock, and a final 10 to hand it back to C3. This is a total of 1250 stall cycles. C3 stalls until C1 hands it off the released lock, which will be 1150 + 40 + 10 + 1000 + 40 = 2240 cycles. Finally, C3 gets the lock 40 cycles later, so it stalls a total of 2280 cycles.

b. The optimized spin lock will have many fewer stall cycles than the regular spin lock because it spends most of the critical section sitting in a spin loop (which while useless, is not defined as a stall cycle). Using the analysis below for the interconnect transactions, the stall cycles will be

- 3 read memory misses (300),
- 1 upgrade/invalidate (15)
- 1 write miss to a cache (40 + 10)
- 1 write miss to memory (100)
- 1 read cache miss to cache (40 + 10)
- 1 write miss to memory (100)
- 1 read miss to cache
- 1 read miss to memory (40 + 10 + 100)
- followed by an upgrade/invalidate (15)
- 1 write miss to cache (40 + 10)
- finally a write miss to cache (40 + 10)
- followed by a read miss to cache (40 + 10)
- & an upgrade/invalidate (15).

So approximately 945 cycles total.

c. Approximately 31 interconnect transactions. The first processor to win arbitration for the interconnect gets the block on its first try (1); the other two ping-pong the block back and forth during the critical section. Because the latency is 40 cycles, this will occur about 25 times (25). The first processor does a write to release the lock, causing another bus transaction (1), and the second processor does a transaction to perform its test and set (1). The last processor gets the block (1) and spins on it until the second processor releases it (1). Finally, the last processor grabs the block (1).

d. Approximately 15 interconnect transactions. Assume processors acquire the lock in order. All three processors do a test, causing a read miss, then a test and set, causing the first processor to upgrade/invalidate and the other two to write miss (6). The losers sit in the test loop, and one of them needs to get back a shared block first (1). When the first processor releases the lock, it takes a write miss (1) and then the two losers take read misses (2). Both have their test succeed, so the new winner does an upgrade/invalidate and the new loser takes a write miss (2). The loser spins on an exclusive block until the winner releases the lock (1). The loser first tests the block (1) and then test-and-sets it, which requires an upgrade/invalidate (1).

Case Study 2: Simple Directory-Based Coherence

5.9 a. i. C3:R, M4

Messages:

- Read miss request message from C3 to Dir4 (011 → 010 → 000 → 100)
- Read response (data) message from M4 to C3 (100 → 101 → 111 → 011)

C3 cache line 0: <I, x, x, …..> → <S, 4, 4, …..>
Dir4: <I, 00000000> → <S, 00001000>, M4 = 4444…..

ii. C3:R, M2

Messages:

- Read miss request message from C3 to Dir2 (011 → 010)
- Read response (data) message from M2 to C3 (010 → 011)

C3 cache line 0: <S, 4, 4, …..> → <S, 2, 2, …..>
C2 Dir: <I, 00000000> → <S, 00001000>, M4 = 4444…..
Note that Dir4 still assumes C3 is holding M4 because C3 did not notify it that it replaced line 0. C3 informing Dir4 of the replacement can be a useful upgrade to the protocol.

iii. C7: W, M4 ← 0xaaaa

Messages:

- Write miss request message from C7 to M4 (111 → 110 → 100)
- Invalidate message from Dir4 to (100 → 101 → 111 → 011)
- Acknowledge message from C3 to Dir4 (011 → 010 → 000 → 100)
- Acknowledge message from Dir4 to C7 (100 → 101 → 111)

C3 cache line 0: <S, 4, 4, …..> → <I, x, x, …..>
C7 cache line 0: <I, x, x, …..> → <M, aaaa, …..>
Dir4: <S, 00001000> → <M, 10000000>, M4 = 4444…..

iv. C1: W, M4 ← 0xbbbb

Messages:

- Write miss request message from C1 to M4 (001 → 000 → 100)
- Invalidate message from Dir4 to C7 (100 → 101 → 111)
- Acknowledge message (with data write-back) from C7 to Dir4 (111 → 110 → 100)
- Write Response (data) message from Dir4 to C1 (100 → 101 → 001)

C7 cache line 0: <M, aaaa, …..> → <I, x, x, …..>
C1 cache line 0: <I, x, x, …..> → <M, bbbb, …..>
Dir4: <M, 10000000> → <M, 00000001> M4 = aaaa…..

Example message formats:

No data message: <no data message flag, message type, destination (dir/cache, & number), block/line number>

Data message: <data message flag, message type, destination (dir/cache, & number), block/line number, data>

(b), (c) Same analysis like (a)

5.10 a. **Core Cx sends**

(a) A write miss message to directory

The directory sends:

(b) A message with shared data to the core Cx that initiated the write miss. The message has the number of read sharers *N*: <message type, destination, data, *N*>

(c) A message to every sharer to asking them to let Cx know they invalidated their caches.

Every one of N sharer sends

(d) A message to Cx acknowledging it invalidated its cache.
Cx waits until it receives *N* acknowledges before writing.

Number of messages = 1 write miss from Cx, 1 response from directory to Cx, *N* messages from directory to sharers, N messages from sharers to Cx = $2N + 2$ messages.

That is the same number of messages as in the straightforward approach.

However, in this new protocol steps (ii), (iii) and (iv) can be overlapped, thus reducing the response time.

Core Cx sends

i. A read miss message to directory.

The directory sends:

ii. A message to owner of modified block <message type, owner address, block number, sharer address (Cx)>

The owner of modified block downgrades cache line state to *shared* and sends:

iii. A message to Cx with data.

Number of messages = 3 (only one is a data transfer message)

Messages in original protocol = read miss, invalidate to owner, modified data write-back from owner to directory core, data sent to Cx), that is 4 messages, 2 of which are data transfer messages.

b. **Core Cx sends**

(a) A read miss message to directory.

The directory sends:

(b) A message to one of data sharers : <message type, data sharer address, block number, requester (Cx)>

Data sharer sends:

(c) Data message to Cx

Number of messages is 3.

Messages in original protocol is 2.

Change would be useful if Cx is much closer to sharer than it is to the directory node. In this case the data message would cross over a few number of hops. The advantage would be more pronounced in a DSM multi-core with a large numbers of cores connected in a topology where the distances to the farthest and closest neighbors vary significantly.

5.11 a. C1: W, M4 ← 0xbbbb   C3: R, M4   C7: R, M2
C3: W, M4 ← 0xaaaa

It should be noted that since both C1 and C3 are accessing M4, either of them can get access first and the behavior is non-deterministic (only decided by implementation delays). In this question we will assume simple delays—based on the Hamming distance between source and destination addresses. Hence, C1 will win over C3!
So the ordering of the transactions on M4 is

C1: W, M4 ← 0xbbbb (wins) → C3: R, M4 → C3: W, M4 (serialization on C3)

The transaction on M2 is independent of the 3 above transactions.

M4 transactions

(a) C1: W, M4 ← 0xbbbb

- Write miss request message from C1 to M4 (001 → 000 → 100)
- Write Response (data) message from Dir4 to C1 (100 → 101 → 001)

C1 cache line 0: <I, x, x, …..> → <M, bbbb, …..>
Dir4: <I, 00000000> →<M, 00000001> M4 = 4444…..

(b) C3: R, M4

- Read miss request message from C3 to Dir4 (011 → 010 → 000 → 100)
- Request data message (and move to shared state) from M4 to C1 (100 → 101 → 001)
- Write response (data) message from C1 to Dir4 (001 → 101 → 100)
- Read response (data) message from Dir4 to C3 (100 → 101 → 111 → 011)

C1 cache line 0: <M, bbbb, …..> → <S, bbbb, …..>
C3 cache line 0: <I, x, x, …..> → <S, bbbb, …..>
Dir4: <M, 00000001> →<S, 00001010>, M4 = bbbb…..

(c) C3: W, M4 ← 0xaaaa
- Write hit request message from C3 to M4 (011 → 010 → 000 → 100)
- Invalidate message from Dir4 to C1 (100 → 101 → 001)
- Acknowledge message from C1 to Dir4 (001 → 000 → 100)
- Write hit response (message from Dir4 to C3 (100 → 101 → 111 → 011)

C1 cache line 0: <S, bbbb, …..> → <I, x, x, …..>
C3 cache line 0: <S, bbbb, …..> → <M, bbbb, …..>
Dir4: <S, 00001010>, →<M, 00001000>, M4= bbbb…..

M2 transaction
C7: R, M2
- Read miss request message from C7 to Dir2 (111 → 110 → 010)
- Read response (data) message from M2 to C7 (010 → 011 → 111)

C7 cache line 0: <I, x, x, …..> → <S, 2222, …..>
Dir2: <I, 00000000> → <S, 1000000>, M4 = 222…..

(b), (c) Same analysis like (a)

5.12 a. C0:R, M7 C2: W, M2<–0xbbbb C3: R, M4 C6: W, M2<–0xaaaa
Assume the local message-free transactions taken to service a request take 1 cycle, or 8 cycles when they are data transfers. We will assume that B—the line size—is equal to 16 bytes

First step is to detail message path to appropriate directory.

RM = read miss, WM = write miss, RMR = read miss response, WMR = write miss response, L0/L1: cache lines L0 and L1

Cycle	C0:R, M7	C2: W, M2<–0xbbbb	C3: R, M4	C6: W, M2<–0xaaaa
1		Miss recognized at local cache controller		
2		Miss serviced at local directory controller		

10	msg(RM) 000 → 001	M2 written into local cache
		Dir2-status = 0000-0100, M
		Cache2-status = L0: I; L1:M[M2, 0xbbbb]
			msg(RM) 011 → 010	msg(WM) 110 → 010

11				Service at Dir, will cause invalidate to local cache

15				Local cache wrote back line to local memory, initiates data response to C6
				Dir2-status = 0100-0000, M
				Cache2-status = L0: I; L1: I

20	msg(RM) 001 → 011		msg(RM) 010 → 000	
30	msg(RM) 011 → 111		msg(RM) 000 → 100	

31	Service at directory dir7, initiates data response to C0 Dir7-status = 0100-0000, S Cache4-status = L0: I; L1: I	Service at directory dir4, initiates data response to C3 Dir4-status = 0001-0000, S Cache4-status = L0: I; L1: I	
181			Data retrieved and formed as message by Dir2
197	Data retrieved & formed as message by Dir7	Data retrieved & formed as message by Dir4	
201			msg (WMR) 010 → 110
208			Data written to cache 6 Cache2-status = L0: I; L1: M[M2, 0xaaaa]
217	msg (RMR) 111 → 110	msg (RMR) 100 → 101	
237	msg (RMR) 110 → 100	msg (RMR) 101 → 111	
257	msg (RMR) 100 → 000	msg (RMR) 111 → 110	
264	Data written to cache 0 Cache0-status = L0: I; L1: S[M7, 0x…]	Data written to cache 3 Cache3-status= L0: I; L1: M[M4, 0x….]	

b. Repeat same procedure as part (a)

5.13 If two messages from the same source core to the same destination core (dir/cache) are adaptively rerouted, then they may reach the destination out-of-order causing unintended behavior.

For example, C1 issues a read miss for block M4, while C3 independently issues a write miss for the block. If Dir4 receives C1's request first, it will issue a data response to C1. When Dir4 receives C3's request—later—it will issue an invalidate to C1. If the data response and the invalidate messages arrive out of order then C1 will be asked to invalidate a block it does not yet own!

5.14 C3: R, M4
C3: R, M2
C2: W, M4 <–0xabcd

In the normal case, the M4 directory will record that C3 shares the block. When C2 needs to write the block, Dir4 will send an invalidate message to C3 (realistically will have to wait for an acknowledge from C3 before it responds to C2).

If the protocol arranges for C3 to send a hint (an information update) to Dir4 stating that it replaced the M4 block and is not sharing it any more, Dir4 will not need to send the invalidate when C2 needs to write the block. Considering that in a real example, the second and third accesses may be reasonably separated in time, the hint will have arrived to M4 before C2 makes its request, and C2 will be spared from waiting for the unnecessary C3-Dir4 message exchange.

Case Study III

5.15

Initial values of A, B are 0	P2:
P1:	While (A==0);
While (B == 0);	B = 1;
A = 1;	

Under any statement interleaving that does not break the semantics of each thread when executed individually, the values of A and B will remain 0.

a. Sequential consistency
For example, in the ordering:

```
P1:While (B == 0);
    P1: A=1;
P2: While (A == 0);
    P2: B=1;
```

B will remain 0, and neither A nor B will be changed. That will be the case for any SC ordering.

b. Total store order
TSO relaxes W → R ordering in a thread. Neither of these two threads has write followed by a read. So the answer will be like part (a).

5.16

P1:	P2:
A=1;	B=1;
A=2;	While (A <> 1);
While (B == 0);	B = 2;

Without an optimizing compiler the threads, SC will allow different orderings. Depending on the relative speeds of P1 and P2, "While (A <> 1);" may be legitimately executed

a. Zero times:
B=1; → A=1;→ While (A <> 1);→ B=2; → A=2; While (B == 0);
B will be set to 2

b. Infinite number of times:
B=1; → A=1;→ A=2; →While (A <> 1); ……
B will be set to 1

c. A few times (A is initially 0)
B=1; → While (A <> 1);→A=1;→ B=2;→ A=2; ….
B will be set to 2

An optimizing compiler might decide that the assignment "A=1;" is extraneous (because A is not read between the two assignments writing to it) and remove it. In that case, "while A .." will loop forever.

5.17 A data-prefetch unit brings data into the cache earlier than it is needed. SC allows multiple legitimate orderings (with different outcomes) and these orderings are often dependent on the speed of reading and writing variables to the memory. Therefore, a data-prefetch unit may boost the likelihood that some orderings will be favored over others.

However, data-prefetching will NOT induce wrong orderings on an otherwise correctly implemented SC system.

5.18
```
A = 1;
B = 2;
If (C == 3)
D = B;
```

Assuming that the programmer is content we the possible outcomes of the code on a PSO-compliant processor.

a. Adding TSO constraints:
```
A = 1;
acquire(S);
B = 2;
release(S);
If (C==3)
    acquire(S) //order B and D writes
    D = B;
    release(S)
```

b. Adding SC constraints
```
A = 1;
acquire(S);
B = 2;
release(S);
acquire(S)
if (C== 3)
    release(S)
    acquire(S) //to order B and D writes
    D = B;
    release(S)
```

5.19 **Assume implementation 1 of Figure 5.38**

a.	P0: write 110 <– 80	Miss, RC satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles).
	P0: read 108	Miss 100 stall cycles for both RC or SC
b.	P0: read 110	Miss 100 stall cycles for both RC or SC
	P0: write 110 <– 90	Hit: no stall cycles for both RC and SC
c.	P0: write 100 <– 80	Miss, RC satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles)
	P0: write 110 <– 90	Hit, but must wait for preceding operation: RC = 0, SC = 100

5.20 P0: write 110 <– 80 Miss, SC must wait until it receives the data (80 = 100 − 20 stall cycles).

P0: read 108	Miss SC must wait until it receives the data (80 = 100 − 20 stall cycles).

Prefetching is one of the techniques that can improve the performance of SC.

5.21 a. The general form for Amdahl's Law is

$$Speedup = \frac{Execution\ time_{old}}{Execution\ time_{new}} = T/t$$

To compute the formula for speedup we need to derive the new execution time.

The exercise states that for the portion of the original execution time that can use *i* processors is given by *F*(*i, p*). The time for running the application on *p* processors is given by summing the times required for each portion of the execution time that can be sped up using *i* processors, where *i* is between one and *p*. This yields

$$t = T * \sum_{i=1}^{p} \frac{f(i,p)}{i}$$

The new speedup formula is then $1/\sum_{i=1}^{p} \frac{f(i,p)}{i}$.

(a) New run time for 8 processors = T(0.2/1 + + 0.2/2 + 0.1/4 + 0.05/6 + 0.45/8)= 0.2 + 0.1 + 0.025 + 0.008 + 0.056 = 0.39 *T

(b) New run time for 32 processors = T(0.2/1 + + 0.2/2 + 0.1/4 + 0.05/6 + 0.15/8 + 0.3/16) = 0.37 *T

(c) New runtime for infinite processors = T(0.2/1 + + 0.2/2+ 0.1/4 + 0.05/6 + 0.15/8 + 0.2/16 + 0.1/128) = 0.36 *T

5.22 a. i. 64 processors arranged a as a ring: largest number of communication hops = 32 → communication cost = (100 + 10*32) ns = 420 ns.

ii. 64 processors arranged as 8x8 processor grid: largest number of communication hops = 14 → communication cost = (100 + 10*14) ns = 240 ns.

iii. 64 processors arranged as a hypercube: largest number of hops = 6 ($\log_2$ 64) → communication cost = (100 + 10*6) ns = 160 ns.

b. Base CPI = 0.75 *cpi*

i. 64 processors arranged a as a ring: Worst case CPI = 0.75 + 0.2/100*(420) = 1.34 *cycles/inst*

ii. 64 processors arranged as 8x8 processor grid: Worst case CPI = 0.75 + 0.2/100 *(240) = 0.98 *cycles/inst*

iii. 64 processors arranged as a hypercube: Worst case CPI = 0.75 + + 0.2/100 *(160) = 0.82 *cycles/inst*

The average CPI can be obtained by replacing the largest number of communications hops in the above calculation by $\hat{h}$,the average numbers of communications hops. That latter number depends on both the topology and the application.

5.23 To keep the figures from becoming cluttered, the coherence protocol is split into two parts. Figure S.3 presents the CPU portion of the coherence protocol, and Figure S.4 presents the bus portion of the protocol. In both of these figures, the

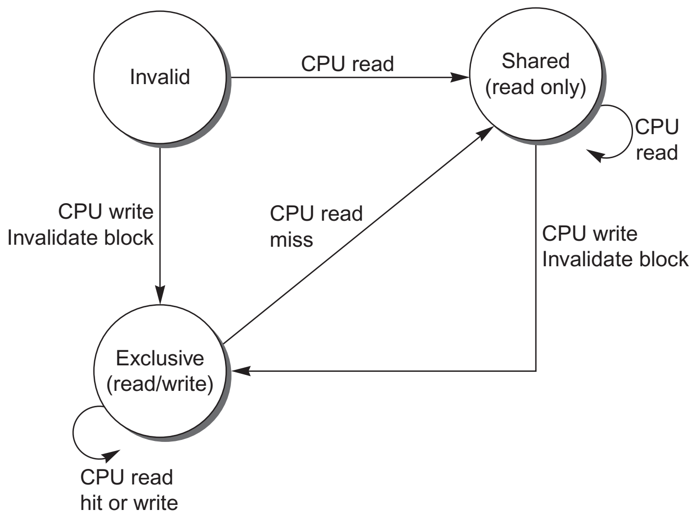


Figure S.3 CPU portion of the simple cache coherency protocol for write-through caches.

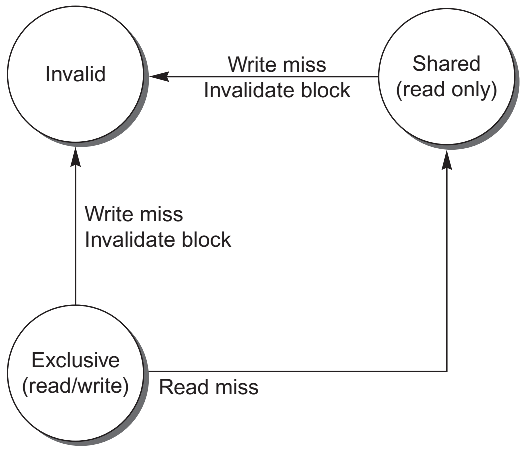


Figure S.4 Bus portion of the simple cache coherency protocol for write-through caches.

arcs indicate transitions and the text along each arc indicates the stimulus (in normal text) and bus action (in bold text) that occurs during the transition between states. Finally, like the text, we assume a write hit is handled as a write miss.

Figure S.3 presents the behavior of state transitions caused by the CPU itself. In this case, a write to a block in either the invalid or shared state causes us to broadcast a "write invalidate" to flush the block from any other caches that hold the block and move to the exclusive state. We can leave the exclusive state through either an invalidate from another processor (which occurs on the bus side of the coherence protocol state diagram), or a read miss generated by the CPU (which occurs when an exclusive block of data is displaced from the cache by a second block). In the shared state only a write by the CPU or an invalidate from another processor can move us out of this state. In the case of transitions caused by events external to the CPU, the state diagram is fairly simple, as shown in Figure S.4. When another processor writes a block that is resident in our cache, we unconditionally invalidate the corresponding block in our cache. This ensures that the next time we read the data we will load the updated value of the block from memory. In addition, whenever the bus sees a read miss, it must change the state of an exclusive block to "shared" as the block is no longer exclusive to a single cache.

The major change introduced in moving from a write-back to write-through cache is the elimination of the need to access dirty blocks in another processor's caches. As a result, in the write-through protocol it is no longer necessary to provide the hardware to force write back on read accesses or to abort pending memory accesses. As memory is updated during any write on a write-through cache, a processor that generates a read miss will always retrieve the correct information from memory. It is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.

5.24 (Refer to solutions in Case Study 1)

5.25 An obvious complication introduced by providing a valid bit per word is the need to match not only the tag of the block but also the offset within the block when snooping the bus. This is easy, involving just looking at a few more bits. In addition, however, the cache must be changed to support write-back of partial cache blocks. When writing back a block, only those words that are valid and modified should be written to memory because the contents of invalid words are not necessarily coherent with the system.

Finally, given that the state machine of Figure 5.6 is applied at each cache block, there must be a way to allow this diagram to apply when state can be different from word to word within a block. The easiest way to do this would be to provide the state information of the figure for each word in the block. Doing so would require much more than one valid bit per word, though. Without replication of state information, the only solution is to change the coherence protocol slightly.

5.26 a. The instruction execution component would significantly be improved because the out-of-order execution and multiple instruction issue allows the latency of this component to be overlapped. The cache access component would be similarly be sped up due to overlap with other instructions, but since cache accesses take longer than functional unit latencies, they would need more instructions to be issued in parallel to overlap their entire latency. Therefore, the speedup for this component would be lower.

The memory access time component would also improve, but the speedup here would be lower than the previous two cases. Because the memory comprises local and remote memory accesses and possibly other cache-to-cache transfers, the latencies of these operations are likely to be very high (100's of processor cycles). The 64-entry instruction window in this example is not likely to allow enough instructions to overlap with such long latencies.

There is, however, one case when large latencies can be overlapped: when they are masked by other long latency operations. This leads to a technique called miss-clustering that has been the subject of some compiler optimizations. The other-stall component would generally improve because they mainly consist of resource stalls, branch mispredictions, etc. The synchronization component—if any-will not be sped up much.

b. Memory stall time and instruction miss stall time dominate the execution for OLTP, more so than for the other benchmarks. Both of these components are not very well addressed by out-of-order execution. Hence, the OLTP workload has lower speedup compared to the other benchmarks with System B.

5.27 Because false sharing occurs when both the data object size is smaller than the granularity of cache block valid bit(s) coverage and more than one data object is stored in the same cache block frame in memory, there are two ways to prevent false sharing. Changing the cache block size or the amount of the cache block covered by a given valid bit are hardware changes and outside the scope of this exercise. However, the allocation of memory locations to data objects is a software issue.

The goal is to locate data objects so that only one truly shared object occurs per cache block frame in memory, and that NO non-shared objects are located in the same cache block frame as any shared object. If this is done, then even with just a single valid bit per cache block, false sharing is impossible. Note that shared, read-only-access objects could be combined in a single cache block and not contribute to the false sharing problem because such a cache block can be held by many caches and accessed as needed without an invalidations to cause unnecessary cache misses.

To the extent that shared data objects are explicitly identified in the program source code, then the compiler should, with knowledge of memory hierarchy details, be able to avoid placing more than one such object in a cache block frame in memory. If shared objects are not declared, then programmer directives can be added to the program. The remainder of the cache block frame should not contain data that would cause false sharing misses.

Alternatively, a block can be padded with non-referenced locations. Padding a cache block frame containing a shared data object with unused memory locations may lead to rather inefficient use of memory space. A cache block may contain a shared object plus objects that are read-only as a trade-off between memory use efficiency and incurring some false-sharing misses. This optimization almost certainly requires programmer analysis to determine if it would be worthwhile. Generally, careful attention to data distribution with respect to cache lines and partitioning the computation across processors is required.

5.28

```
for (int p= 0; p<=3; p++) // Each iteration of is executed on a separate processor.
        {
                sum [p] = 0;
                for (int i= 0; i<n/4; i++) // n is size of word_count and is
                                                          divisible by 4
                          sum[p] = sum[p] + word_count[p+4*i];
        }
total_sum = sum[0] +sum[1]+sum[2]+sum[3] //executed only on processor.
```

a. Array sum [] has four elements that are distributed on four processors (outer loop is distributed). They share the same cache line and each one of the them is modified in each inner-loop iteration. However, the four elements do not interact within the for loops. They—thus—suffer from false sharing, and will cause a large number of coherence write misses (one miss for every inner loop iteration).

   A cache line of array word-count will be shared by all four processors too. It is false sharing because the different processors read distinct elements. However, this false sharing is benign from the coherence perspective, as the word_count elements are read but not written.

   On the other hand, elements n*k through n*k + 7 of array word-count will be in the same memory block, and it will suffer four read misses: one miss per processor. Every processor will use two elements of a the block, and will not need after that.

b.

```
for (int p= 0; p<=3; p++) // Each iteration of is executed on a separate processor.
        {
                sum [p] = 0;
                for (int i= p*n/4; i<(p+1)*n/4; i++) // n is size of
                                          word_count and is divisible by 4
                          sum[p] = sum[p] + word_count[i];
        }
total_sum = sum[0] +sum[1]+sum[2]+sum[3] //executed only on processor.
```

   The original code forced the processors to access the array in an interleaved manner and reduced the reuse of every memory block fetched. Every memory block experiences four misses (one miss per processor).

   The updated code divides the array into four portions and every processor will work on one of these portions. A memory block will experience one miss is one the processors' caches.

c. To rid the code of false sharing, the array sum will be modified to an become sum[4] [8]. Only element 0 of every row will be used. The rest of the elements are there to ensure that each sum [0] [*] is in a different cache line.

   The summation statements will be modified to

```
Sum[p][0] = sum[p][0] + word_count[i];
.....
total_sum = sum[0][0] +sum[1][0]+sum[2][0]+sum[3][0];
```

5.29 The problem illustrates the complexity of cache coherence protocols. In this case, this could mean that the processor P1 evicted that cache block from its cache and immediately requested the block in subsequent instructions. Given that the write-back message is longer than the request message, with networks that allow out-of-order requests, the new request can arrive before the write back arrives at the directory. One solution to this problem would be to have the directory wait for the write back and then respond to the request. Alternatively, the directory can send out a negative acknowledgment (NACK).

Note that these solutions need to be thought out very carefully since they have potential to lead to deadlocks based on the particular implementation details of the system. Formal methods are often used to check for races and deadlocks.

5.30 If replacement hints were used, then the CPU replacing a block would send a hint to the home directory of the replaced block. Such hint would lead the home directory to remove the CPU from the sharing list for the block. That would save an invalidate message when the block is to be written by some other CPU. Note that while the replacement hint might reduce the total protocol latency incurred when writing a block, it does not reduce the protocol traffic (hints consume as much bandwidth as invalidates).

5.31 a. Considering first the storage requirements for nodes that are caches under the directory subtree (Figure S.5):

The directory at any level will have to allocate entries for all the cache blocks cached under that directory's subtree. In the worst case (all the CPU's under

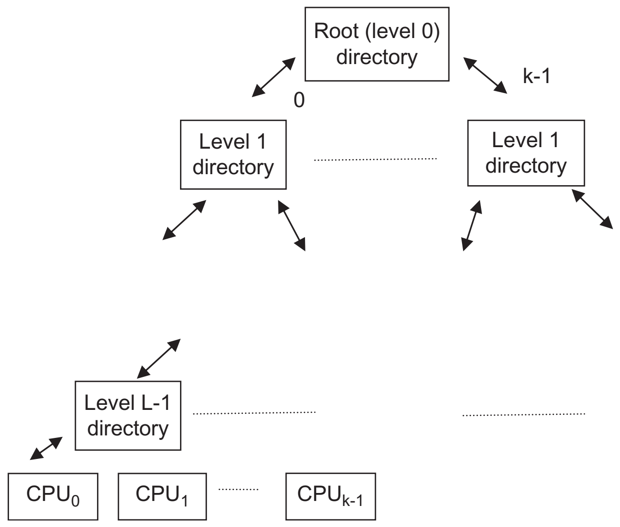


Figure S.5 Tree-based directory hierarchy (k-ary tree with l levels).

the subtree are not sharing any blocks), the directory will have to store as many entries as the number of blocks of all the caches covered in the subtree. That means that the root directory might have to allocate enough entries to reference all the blocks of all the caches. Every memory block cached in a directory will represented by an entry <block address, k-bit vector>, the k-bit vector will have a bit specifying all the subtrees that have a copy of the block. For example, for a binary tree an entry <m, 11> means that block m is cached under both branches of the tree. To be more precise, one bit per subtree would be adequate if only the valid/invalid states need to be recorded; however to record whether a block is modified or not, more bits would be needed. Note that no entry is needed if a block is not cached under the subtree.

If the cache block has m bits (tag + index) then and s state bits need to be stored per block, and the cache can hold b blocks, then the directories at level L-1 (lowest level just above CPU's) will have to hold k*b entries. Each entry will have (m + k*s) bits. Thus each directory at level L-1 will have ($mkb + k^2bs$) bits. At the next level of the hierarchy, the directories will be k times bigger. The number of directories at level i is k^i.

To consider memory blocks with a home in the subtree cached outside the subtree. The storage requirements per directory would have to be modified.

Calculation outline:
Note that for some directory (for example the ones at level l-1) the number of possible home nodes that can be cached outside the subtree is equal to ($b*(k^1 - x)$), where k^1 is the total number of CPU's, b is the number of blocks per cache and x is the number of CPU's under the directory's subtree. It should be noted that the extra storage diminishes for directories in higher levels of the tree (for example the directory at level 0 does not require any such storage since all the blocks have a home in that directory's subtree).

b. When a memory block is accessed (read/written) the home directory "cache" will be consulted for the address of the block.

If there is a hit then sharing bits will be modified as needed, the actions (invalidates, etc.) performed in a full-directory implementation will be taken.

If the access is a miss then the directory block descriptor will need to be brought in the directory cache before being updated with the pertinent information. Unless there is an available (i.e., invalid) entry in the directory cache, some entry will have to be evicted. To ensure coherence for the memory blocked managed by the evicted entry, invalidates will have to be sent to all the processors who own this block so that it is taken out of circulation.

c. Leftmost bit is flag, organization is **flag: bit[0:7]**
flag = 0 → node number mode, the other 8 bit describe a node number
flag = 1 → bit vector mode, the other 8 bits indicate which of eight groups of processors (8 processors each) has a copy of the block.

For example
0:0000-1111 → block is used only by processor 15 (maybe be reserved for blocks in modified or exclusive states)
1:0000-0000 → block is not used by any group of processors
1:0000-0011 → block is used by some processors in 8-processor groups 6 and 7 (positions with non-zero bits)

d. A directory with no status information about home memory blocks does not have any information about its home memory blocks, however it is needed to impose ordering on requests competing for its memory blocks. Since it does not which processors own a block and in what state, it will have to broadcast message to all processors and wait for ALL processors (whether they own a block or not) to reply.

5.32 Test and set code using load linked and store conditional.

```
mov x3, #1
lr  x2, x1
sc x3, x1
```

Typically, this code would be put in a loop that spins until a 1 is returned in x3.

5.33 Assume a cache line that has a synchronization variable and the data guarded by that synchronization variable in the same cache line. Assume a two-processor system with one processor performing multiple writes on the data and the other processor spinning on the synchronization variable.

With an invalidate protocol, false sharing will mean that every access to the cache line ends up being a miss resulting in significant performance penalties when the missing processor needs to access the rest of the data in the same cache line.

By padding the cache, line with extra unused space (see problem 5.28) this problem can be resolved.

5.34 The monitor has to be placed at a point through which all memory accesses pass. One suitable place will be in the memory controller at some point where accesses from the four cores converge (since the accesses are uncached anyways). The monitor will use some sort of a cache where the tag of each valid entry is the address accessed by some load-linked instruction. In the data field of the entry, the core number that produced the load-linked access—whose address is stored in the tag field—is stored.
This is how the monitor reacts to the different memory accesses.

- Read not originating from a load-linked instruction:
  - Bypasses the monitor progresses to read data from memory
- Read originating from a load-linked instruction:
  - Checks the cache, if there is any entry with whose address matches the read address even if there is a partial address match (for example, read [0:7] and read [4:11] overlap match in addresses [4:7]), the matching cache entry is invalidated and a new entry is created for the new read (recording the core number that it belongs to). If there is no matching entry in the cache, then a

new entry is created (if there is space in the cache). In either case the read progresses to memory and returns data to originating core.

- Write not originating from a store-conditional instruction:
  - Checks the cache , if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the matching cache entry is invalidated. The write progresses to memory and writes data to the intended address.
- Write originating from a store-conditional instruction:
  - Checks the cache, , if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the core number in the cache entry is compared to the core that originated the write. If the core numbers are the same, then the matching cache entry is invalidated, the write proceeds to memory and returns a success signal to the originating core. In that case, we expect the address match to be perfect—not partial—as we expect that the same core will not issue load-linked/store conditional instruction pairs that have overlapping address ranges. If the core numbers differ, then the matching cache entry is invalidated, the write is aborted and returns a failure signal to the originating core. This case signifies that synchronization variable was corrupted by another core or by some regular store operation.

5.35 Inclusion states that each higher level of cache contains all the values present in the lower cache levels, i.e., if a block is in L1 then it is also in L2. The problem states that L2 has equal or higher associativity than L1, both use LRU, and both have the same block size.

When a miss is serviced from memory, the block is placed into all the caches, i.e., it is placed in L1 and L2. Also, a hit in L1 is recorded in L2 in terms of updating LRU information. Another key property of LRU is the following. Let A and B both be sets whose elements are ordered by their latest use. If A is a subset of B such that they share their most recently used elements, then the LRU element of B must either be the LRU element of A or not be an element of A.

This simply states that the LRU ordering is the same regardless if there are 10 entries or 100. Let us assume that we have a block, D, that is in L1, but not in L2. Since D initially had to be resident in L2, it must have been evicted. At the time of eviction D must have been the least recently used block. Since an L2 eviction took place, the processor must have requested a block not resident in L1 and obviously not in L2. The new block from memory was placed in L2 (causing the eviction) and placed in L1 causing yet another eviction. L1 would have picked the least recently used block to evict.

Since we know that D is in L1, it must be the LRU entry since it was the LRU entry in L2 by the argument made in the prior paragraph. This means that L1 would have had to pick D to evict. This results in D not being in L1 which results in a contradiction from what we assumed. If an element is in L1 it has to be in L2 (inclusion) given the problem's assumptions about the cache.

5.36 Analytical models can be used to derive high-level insight on the behavior of the system in a very short time. Typically, the biggest challenge is in determining the values of the parameters. In addition, while the results from an analytical model can give a good approximation of the relative trends to expect, there may be significant errors in the absolute predictions.

Trace-driven simulations typically have better accuracy than analytical models, but need greater time to produce results. The advantages are that this approach can be fairly accurate when focusing on specific components of the system (e.g., cache system, memory system, etc.). However, this method does not model the impact of aggressive processors (mispredicted path) and may not model the actual order of accesses with reordering. Traces can also be very large, often taking gigabytes of storage, and determining sufficient trace length for trustworthy results is important. It is also hard to generate representative traces from one class of machines that will be valid for all the classes of simulated machines. It is also harder to model synchronization on these systems without abstracting the synchronization in the traces to their high-level primitives.

Execution-driven simulation models all the system components in detail and is consequently the most accurate of the three approaches. However, its speed of simulation is much slower than that of the other models. In some cases, the extra detail may not be necessary for the particular design parameter of interest.

5.37 One way to devise a multiprocessor/cluster benchmark whose performance gets worse as processors are added:

Create the benchmark such that all processors update the same variable or small group of variables continually after very little computation.

For a multiprocessor, the miss rate and the continuous invalidates in between the accesses may contribute more to the execution time than the actual computation, and adding more CPU's could slow the overall execution time.

For a cluster organized as a ring, communication costs needed to update the common variables could lead to inverse linear speedup behavior as more processors are added.

Chapter 6 Solutions

Case Study 1: Total Cost of Ownership Influencing Warehouse-Scale Computer Design Decisions

6.1 a. The servers being 10% faster means that fewer servers are required to achieve the same overall performance. From the numbers in Figure 6.13, there are initially 45,978 servers. Assuming a normalized performance of 1 for the baseline servers, the total performance desired is thus $45{,}978 \times 1 = 45{,}978$. The new servers provide a normalized performance of 1.1, thus the total servers required, y, is $y \times 1.1 = 45{,}978$. Thus $y = 41{,}799$. The price of each server is 20% more than the baseline, and is thus $1.2 \times 1450 = \$1740$. Therefore the server CAPEX is $72,730,260.

b. The servers use 15% more power, but there are fewer servers as a result of their higher performance. The baseline servers use 165 W, and thus the new servers use $1.15 \times 165 = 190$ W. Given the number of servers, and therefore the network load with that number of servers (377,795 W), we can determine the critical load needed to support the higher powered servers. Critical load = number of servers $\times$ watts per server + network load. Critical load = $41{,}799 \times 190$ W + 377,795 W. Critical load = 8,319,605 W. With the same utilization rate and the same critical load usage, the monthly power OPEX is $493,153.

c. The original monthly costs were $3,530,920. The new monthly costs with the faster servers is $3,736,648, assuming the 20% higher price. Testing different prices for the faster servers, we find that with a 9% higher price, the monthly costs are $3,536,834, or roughly equal to the original costs.

6.2 a. The low-power servers will lower OPEX but raise CAPEX; the net benefit or loss depends on how these values comprise TCO. Assume server operational and cooling power is a fraction x of OPEX and server cost is a fraction y of CAPEX. Then, the new OPEX, CAPEX, and TCO, by using low-power servers, are given by:

$$\text{OPEX}' = \text{OPEX}[(1-0.15)x + (1-x)]$$
$$\text{CAPEX}' = \text{CAPEX}[(1+0.2)y + (1-y)]$$
$$\text{TCO}' = \text{OPEX}' + \text{CAPEX}' = \text{OPEX}(1-0.5x) + \text{CAPEX}(1+0.2y)$$

For example, for the data in Figure 6.14, we have

$$\text{OPEX}' = \$560{,}000(1-0.15\times 0.87) = \$486{,}920$$
$$\text{CAPEX}' = \$3{,}225{,}000(1+0.2\times 0.62) = \$3{,}624{,}900$$
$$\text{TCO}' = \text{OPEX}' + \text{CAPEX}' = \$4{,}111{,}820$$

In this case, TCO $>$ TCO′, so the lower power servers are not a good tradeoff from a cost perspective.

b. We want to solve for the fraction of server cost, S, when TCO = TCO′:

$$\text{TCO} = \text{TCO}' = \text{OPEX}(1 - 0.15x) + \text{CAPEX}(1 + Sy)$$
$$\text{TCO} - \text{OPEX}(1 - 0.15x) = \text{CAPEX}(1 + Sy)$$
$$S = \frac{\text{TCO} - \text{OPEX}(1 - 0.15x) - \text{CAPEX}}{\text{CAPEX}y}$$

We can solve for this using the values in Figure 6.14:

$$S = \frac{\$3,800,000 - \$560,000(1 - 0.15 \times 0.87) - \$3,225,000}{\$3,255,000(0.62)} \approx 0.044$$

So, the low-power servers need to be less than 4.4% more expensive than the baseline servers to match the TCO.

If the cost of electricity doubles, the value of x in the previous equations will increase as well. The amount by which it increases depends on the other costs involved. For the data in Figure 6.14, the cost of electricity doubling will make the monthly power use equal to \$475,000 × 2 = \$950,000. This increases total cost to be \$4,275,000 and gives a value of $x \approx 0.92$.

$$S = \frac{\$4,275,000 - \$1,035,000(1 - 0.15 \times 0.92) - \$3,225,000}{\$3,255,000(0.62)} \approx 0.079$$

At this value of x, $S \approx 7.9\%$, slightly more than before. Intuitively this makes sense, because increasing the cost of power makes the savings from switching to lower-power hardware greater, making the break-even point for purchasing low-power hardware less aggressive.

6.3 a. The baseline WSC used 45,978 servers. At medium performance, the performance is 75% of the original. Thus the required number of servers, x, is $0.75 \times x = 45{,}978$. $x = 61{,}304$ servers.

b. The server cost is remains the same baseline \$1450, but the per-server power is only 60% of the baseline, resulting in 0.6 × 165 W = 99 W per server. These numbers yield the following CAPEX numbers: Network: \$16,444,934.00; Server: \$88,890,800.00. Like in 6.1, we calculate the total critical load as Critical load = number of servers × watts per server + network load. Critical load = 61,304 × 99 W + 531,483 W. Critical load =6,600,579 W. This critical load, combined with the servers, yields the following monthly OPEX numbers: Power: \$391,256; Total: \$4,064,193.

c. At 20% cheaper, the server cost is 0.8 × \$1450 = \$1160. Assuming a slowdown of X%, the number of servers required is 45,978/(1 − X). The network components required can be calculated through the spreadsheet once the number of servers is known. The critical load required is Critical load = number of servers × watts per server + network load. Given a reduction of power Y% compared to the baseline server, the watts per server =(1 − Y) × 165 W. Using these numbers and plugging them into the spreadsheet, we can find a few points where the TCO is approximately equal.

6.4 For a constant workload that does not change, there would not be any performance advantage of using the normal servers at medium performance versus the slower but cheaper servers. The primary differences would be in the number of servers required for each option, and given the number of servers, the overall TCO of each option. In general, the server cost is one of the largest components, and therefore reducing the number of servers is likely to provide greater benefits than reducing the power used by the servers.

6.5 Running all of the servers at medium power (option 1) would mean the WSC could handle higher peak loads throughout the day (such as the evening in populated areas of the world when internet utilization increases), compared to the cheaper but slower servers (option 2).

This resiliency to spikes in load would come at the cost of increased CAPEX and OPEX cost, however. A detailed analysis of peak and average server load should be considered when making such a purchasing decision.

6.6 There are multiple abstractions made by the model. Some of the most significant abstractions are that the server power and datacenter power can be represented by a single, average number. In reality, they are likely to show significant variation throughout the day due to different usage levels. Similarly, all servers may not have the same cost, or identical configurations. They may be purchased over a time period, leading to multiple different kinds of servers or different purchase prices. Furthermore, they may be purchased with different usages in mind, such as a storage-oriented server with more hard drives or a memory-capacity oriented server with more RAM. There are other details that are left out, such as any licensing fees or administration costs for the servers. When dealing with the large number of servers in a WSC, it should be safe to make assumptions about average cost across the different systems, assuming there are no significant outliers. There are additional assumptions in the cost of power being constant (in some regions its pricing will vary throughout the day). If there are significant variations in power pricing and power usage throughout the day, then it can lead to significantly different TCO versus a single average number.

Case Study 2: Resource Allocation in WSCs and TCO

6.7 a. Consider the case study for a WSC presented in Figure 6.13: We must provision 8 MW for 45,978 servers. Assuming no oversubscription, the nameplate power for each server is:

$$P_{\text{nameplate}} = \frac{8\,\text{MW}}{45{,}978\,\text{servers}} \approx 174\frac{\text{W}}{\text{server}}$$

In addition, the cost per server is given in Figure 6.13 as \$1450.

Now, if we assume a new nameplate server power of 200 W, this represents an increase in nameplate server power of $200 - 174 = 26$ W/server, or approximately 15%. Since the average power utilization of servers is 80% in this WSC, however, this translates to a $15\% \times 80\% = 12\%$ increase in the cost of server power. To calculate the percentage increase in power and cooling, we consider the power usage effectiveness of 1.45. This means that an increase of server power of 12% translates to a power and cooling increase of 12% × 1.45 = 17.4%.

If we also assume, as in Figure 6.13, that the original cost per server is $1450, a new server would cost $3000 − $1450 = $1550 more, for an increase of about 109%.

We can estimate the effects on TCO by considering how power and cooling infrastructure and server cost factor into monthly amortized CAPEX and OPEX, as given in Figure 6.14. Increasing the cost of power and cooling infrastructure by 17.4% gives a new monthly cost of $475,000 × 1.174 = $557,650 (an increase of $82,650) and increasing server cost by 109% gives a new monthly cost of $2,000,000 × 2.09 = $4,180,000 (an increase of $2,180,000).

The new monthly OPEX increases by $82,650 + $2,180,000 = $2,262,650.

b. We can use a similar process as in Part a, to estimate the effects on TCO for this cheaper but more power-hungry server:

Nameplate server power increases by 300 − 174 = 126 W/server, or approximately 72%. This translates to an increase in power and cooling of 72% × 1.45 ≈ 104%.

In addition, a new server would cost $2000 − $1450 = $550 more, for an increase of about 38%.

We can once again estimate the effects on TCO by calculating the new monthly cost of power and cooling infrastructure as $475,000 × 1.72 = $817,000 (an increase of $342,000) and the new monthly server cost as $2,000,000 × 1.38 = $2,760,000 (an increase of $760,000).

The new monthly OPEX increases by $342,000 + $760,000 = $1,102,000. This option does not increase TCO by as much as that in Part a.

c. If average power usage of the servers is only 70% of the nameplate power (recall in the baseline WSC it was 80%), then the average server power would decrease by 12.5%. Given a power usage effectiveness of 1.45, this translates to a decrease in server power and cooling of 12.5% × 1.45 = 18.125%. This would result in a new monthly power and cooling infrastructure cost of $475,000 × (1 − 0.18125) = $388,906.25 (a decrease of $86,093.75).

6.8 a. Assume, from Figure 6.13, that our WSC initially contains 45,978 servers. If each of these servers has a nameplate power of 300 W, the critical load power for servers is given by 45,978 × 300 = 13,793,400 W.

If, in actuality, the servers had an average power consumption of 225 W, the average power load of the servers in the WSC would be 45,978 × 225 = 10,345,050 W.

This means that (13,793,400 − 10,345,050)/13,793,400 = 25% of the provisioned power capacity remains unused. The monthly cost of power for such servers, assuming \$0.07 per kWh, from Figure 6.13, and 720 hours per month, is $10,345.05\,\text{kW} \times 0.07\frac{\$}{\text{kWh}} \times 720\,\text{h} = \$521,390.52$.

b. If we instead assume the nameplate power of the server to be 500 W, the critical load power for servers would be given by 45,978 × 500 =22,989,000 W.

If, in actuality, the servers had an average power consumption of 300 W, the average power load of the servers in the WSC would be 45,978 × 300 = 13,793,400 W.

This means that (22,989,000 − 13,793,400)/22,989,000 = 40% of the provisioned power capacity remains unused. The monthly cost of power for such servers, assuming \$0.07 per kWh, from Figure 6.13, and 720 hours per month, is $13,793.4\,\text{kW} \times 0.07\frac{\$}{\text{kWh}} \times 720\,\text{h} = \$695,187,36$.

6.9 Assuming infrastructure can be scaled perfectly with the number of machines, the *per-server* TCO of a data-center whose capacity matches its utilization will not change.

In general, however, it depends on how a variety factors such as power delivery and cooling scale with the number of machines. For example, the power consumption of data-center cooling infrastructure may not scale perfectly with the number of machines and, all other things equal, could require a larger per-server TCO for adequately cooling higher data-center capacities. Conversely, server manufacturers often provide discounts for servers bought in bulk; all other things equal, the per-server TCO of a data-center with a higher capacity could be less than that of a lower capacity.

6.10 During off-peak hours the servers could potentially be used for other tasks, such as off-line batch processing jobs. Alternatively, the operators could try to sell the excess capacity by offering computing services that are priced cheaper during the off-peak hours versus peak hours. Other options to save cost include putting the servers into low-power modes, or consolidating multiple workloads onto less servers and switching off the now idle servers. If there are certain classes of servers that can more efficiently serve the smaller load (such as Atom-based servers versus Xeon-based servers), then they could be used instead. However, each of these options effectively reduce compute capacity available during the more idle periods. If the workload happens to have a spike in activity (e.g., a large news event leads to significant traffic during an off-peak period), then these options run the risk of not having enough compute capacity available for those spikes. Switching to low-power modes will allow the servers to more quickly respond to higher spikes in load rather than consolidation and shutting off of servers, but will provide less cost and power savings.

6.11 There are many different possible proposals, including: server consolidation, using low-power modes, using low-power processors, using embedded-class processors, using low-power devices such as solid-state disks, developing energy-proportional servers, and others. The challenges to many of these proposals is developing models that are detailed enough to capture all of the impacts of the different operational modes, as well as having accurate workload traces to drive the power models. Based on the proposals, some advantages include lower power usage or better response times. Potential disadvantages include inflexibility, lower performance, or higher CAPEX costs.

Exercises

6.12 a. One example of when it may be more beneficial to improve the instruction- or thread-level parallelism than request-level parallelism is if the latency of a workload is more important than the throughput. While request-level parallelism can enable more concurrent requests to be processed (increasing system throughput), without improving instruction- or thread-level parallelism, each of those requests will be processed at a slower speed.

b. The impact of increasing request-level parallelism on software design depends on the workload being parallelized. The impact can range from minimal—for applications which do not keep any state and do not communicate with multiple instances of themselves, such as web servers—to far-reaching—for applications which require fine-grained communicate or shared state, such as database software. This is because by increasing request-level parallelism, communication and state must be shared across racks in a datacenter or even datacenters across the world, introducing a range of issues such as data consistency and redundancy.

c. In addition to the software design overheads discussed in Part b., one example of the potential drawbacks of increasing request-level parallelism, is that more machines will be required to increase system throughput, which, for a fixed TCO, may require the purchase of more inexpensive, commodity machines. Such machines may fail more frequently than more expensive machines optimized for high instruction- and thread-level parallelism, and mechanisms will have to be put in place to counteract these machine failures to prevent the loss of work and data.

6.13 a. At a high level, one way to think about the effect of round-robin scheduling for compute-heavy workloads is that it more evenly spreads the amount of computation across a given number of machines compared to consolidated scheduling. There are a variety of trade-offs present by making such a scheduling decision, and we present several here for discussion.

In terms of power and cooling, round-robin scheduling may decrease the power density of a given group of servers, preventing hot spots from forming in the data center and possibly requiring less aggressive cooling solutions to be employed. On the other hand, modern server power supplies are not very efficient at low utilizations (this means more power is lost in conversion for lower utilizations than for higher utilizations), so under-utilizing many machines can lead to losses in power efficiency.

In terms of performance, round-robin scheduling will likely benefit compute-heavy workloads because there will be no resource sharing between processes, in this example, which would otherwise lead to performance degradation. However, if processes accessed the same data, they might benefit from being located on the same machine, as one process could fetch data, making it readily available by the time the other process needed to use it.

For reliability, round robin scheduling will decrease the number of jobs lost due to machine failure (assuming machine failures occur on machines running jobs). Placing jobs which must communicate with one another across many machines, however, may introduce new points of failure, such as networking equipment.

b. In general, the effect of round-robin scheduling I/O-heavy workloads will depend on how data is laid out across the servers and the access patterns of the workload itself. For example, if data are replicated across different servers, the scheduling decision will have less of an effect on performance than if the data are partitioned across racks. In the first case, wherever a process is scheduled, it will be able to access its data locally; in the second case, processes must be scheduled on the same rack that their data is located on in order to not pay the cost of traversing the array of racks to access their data.

c. Assuming the bandwidth available at the networking hardware used to connect the servers being used is limited, and the topology of the network used to connect the servers is tree-like (e.g., a centralized switch connecting the racks and localized switches connecting servers within racks), scheduling the jobs in a round-robin fashion at the largest scope (array of racks) can help balance the bandwidth utilization for requests across each of the racks. If the applications are network-intensive because they communicate with one another, however, round-robin scheduling will actually create additional traffic through the switches used to connect various servers across the racks.

6.14 a. Total dataset size is 300 GB, network bandwidth 1 Gb/s, map rate is 10 s/GB, reduce rate is 20 s/GB. 30% of data will be read from remote nodes, and each output file is written to two other nodes. According to Figure 6, disk bandwidth is 200 MB/s. For simplicity, we will assume the dataset is broken up into an equal number of files as there are nodes. With five nodes, each node will have to process $300\text{ GB}/5 = 60\text{ GB}$. Thus $60\text{ GB} \times 0.3 = 18\text{ GB}$ must be read remotely, and 42 GB must be read from disk. Using the disk bandwidths from Figure 6.6, we calculate the time for remote data access as $18\text{ GB}/100\text{ MB/s} = 180$ seconds, and the time for local data access as $42\text{ GB}/200\text{ MB/s} = 210$ seconds; the data must be accessed for both the map and the reduce. Given the map rate, the map will take $60\text{ GB} \times 10\text{ s/GB} = 600$ seconds; given the reduce rate, the reduce will take $60\text{ GB} \times 20\text{ s/GB} = 1200$ seconds. The total expected execution time is therefore $(180 + 210) \times 2 + 600 + 1200 = 2508$ seconds. The primary bottleneck is the reduce phase, while the total data transfer time is the secondary bottleneck. At 1000 nodes, we have $300\text{ GB}/1000 = 300\text{ MB}$ per node. Thus $300\text{ MB} \times 0.3 = 90\text{ MB}$ must be read remotely, and 210 MB must be read from local disk. These numbers give the following access times: network. = 90 MB/100 MB/s = 0.9 seconds, and disk = 210 MB/200 MB/s = 1.05 seconds. The map time is 300 MB × 10 s/GB = 3 seconds, and the reduce time is 300 MB × 20 s/GB = 6 seconds. The bottlenecks actually remain identical across the different node sizes as all communication remains local within a rack and the problem is divided up evenly.

b. With 40 nodes per rack, at 100 nodes there would be 3 racks. Assume an equal distribution of nodes (~33 nodes per rack). We have 300 GB/100 = 3 GB per node. Thus 900 MB must be read remotely, and there is a 2/3 chance it must go to another rack. Thus 600 MB must be read from another rack, which has disk bandwidth of 10 MB/s, yielding 60 seconds to read from a different rack. The 300 MB to read from within the rack will take 300 MB/100 MB/s = 3 seconds. Finally the time to do the map is 30 seconds, and the reduce is 60 seconds. The total runtime is therefore (60 + 3) × 2 + 30 + 60 = 216. Data transfer from machines outside of the rack wind up being the bottleneck.

c. Let us calculate with 80% of the remote accesses going to the same rack. Using the numbers from b., we get 900 MB that must be read remotely, of which 180 MB is from another rack, and 720 MB is within a node's rack. That yields 180 MB/10 MB/s = 18 seconds to read remote data, and 720 MB/100 MB/s = 7.2 seconds to read local data. The total runtime is therefore (18 + 7.2) × 2 + 30 + 60 = 140.4 seconds. Now the reduce phase is the bottleneck.

d. The MapReduce program works by initially producing a list of each word in the map phase, and summing each of the instances of those words in the reduce phase. One option to maximize locality is, prior to the reduce phase, to sum up any identical words and simply emit their count instead of a single word for each time they are encountered. If a larger amount of data must be transferred, such as several lines of text surrounding the word (such as for creating web search snippets), then during the map phase the system could calculate how many bytes must be transferred, and try to optimize the placement of the items to be reduced to minimize data transfer to be within racks.

6.15 a. Assume that replication happens within the 10-node cluster. One-way replication means that only one copy of the data exists, the local copy. Two-way replication means two copies exist, and three-way replication means three copies exist; also assume that each copy will be on a separate node. The numbers from Figure 6.1 show the outages for a cluster of 2400 servers. We need to scale the events that happen based on individual servers down to the 10 node cluster. Thus this gives approximately four events of hard-drive failures, slow disks, bad memories, misconfigured machines, and flaky machines per year. Similarly we get approximately 20 server crashes per year. Using the calculation in the Example in 6.1, we get Hours Outage = (4 + 1 + 1 + 1) × 1 hour + (1 + 20) × 5 minutes =8.75 hours. This yields 99.9% availability, assuming one-way replication. In two-way replication, two nodes must be down simultaneously for availability to be lost. The probability that two nodes go down is $(1 - 0.999) \times (1 - 0.999) = 1e-6$. Similarly the probability that three nodes go down is $1e-9$.

b. Repeating the previous problem but this time with 1000 nodes, we have the following outages: Hours Outage = $(4 + 104 + 104 + 104) \times 1$ hour + $(104 + 2083) \times 5$ minutes = $316 + 182.25 = 498.25$ hours. This yields 94.3% availability with one-way replication. In two-way replication, two nodes must be down simultaneously, which happens with probability $(1 - 0.943) \times (1 - 0.943) = 0.003$; in other terms, two-way replication provides 99.7% availability. For three-way replication, three nodes need to be down simultaneously, which happens with probability $(1 - 0.943) \times (1 - 0.943) \times (1 - 0.943) = 0.0002$; in other terms, three-way replication provides 99.9998% availability.

c. For a MapReduce job such as sort, there is no reduction in data from the map phase to the reduce phase as all of the data is being sorted. Therefore there would be 1PB of intermediate data written between the phases. With replication, this would double or triple the I/O and network traffic. Depending on the availability model, this traffic could either be fully contained within a rack (replication only within a rack) or could be across racks (replication across racks). Given the lower disk bandwidth when going across racks, such replication must be done in a way to avoid being on the critical path.

d. For the 1000 node cluster, we have 94.3% availability with no replication. Given that the job takes 1 hour to complete, we need to calculate the additional time required due to restarts. Assuming the job is likely to be restarted at any time due to a failure in availability, on average the job will therefore restart halfway through the task. Thus if there is one failure, the expected runtime = 1 hour + $0.067 \times$ (1 hour/2). However, for a restarted task there is still some probability that it will again have a failure and must restart. Thus the generalized calculation for expected runtime $= 1\,\text{hour} + \sum_{n-1}^{n-inf} 0.067^n \times \left(\frac{1\,\text{hour}}{2}\right)$ This equation approximates to 1.036 hours.

e. The disk takes 10,000 microseconds to access, and the write of 1 KB of data at 200 MB/s will take 5 microseconds. Therefore each log takes 10,005 microseconds to complete. With two and three-way replication, writing to a node in the rack will take 11,010 microseconds, and in the array will take 12,100 microseconds. This leads to 10% and 21% overhead, respectively. If the log was in-memory and only took 10 μs to complete, this leads to 110099900% and 120999900% overhead, assuming the replicas are written to disk. If instead they are written to remote memory, then the time to write the replicas are 100 + 10 = 110 μs and 300 + 100 = 400 μs for rack and array writes, respectively. These times translate to 1000% and 3900% overhead.

f. Assuming the two network trips are within the same rack, and the latency is primarily in accessing DRAM, then there is an additional 200 μs due to consistency. The replications cannot proceed until the transaction is committed, therefore the actions must be serialized. This requirement leads to an additional

310 μs per commit for consistency and replication in-memory on other nodes within the rack.

6.16 a. The "small" server only achieves approximately 3 queries per second before violating the <0.5 second response SLA. Thus to satisfy 30,000 queries per second, 10,000 "small" servers are needed. The baseline server on the other hand achieves 7 queries per second within the SLA time, and therefore needs approximately 4286 servers to satisfy the incoming queries. If each baseline server cost $2000, the "small" servers must be $857.20 (42% cheaper) or less to achieve a cost advantage.

b. Figure 6.1 gives us an idea of the availability of a cluster of nodes. With 4286 servers, we can calculate the availability of all nodes in the cluster. Looked at in another way, if we take 1-Availability, it is the probability that at least one node is down, giving us an estimate of how much overprovisioned the cluster must be to achieve the desired query rate at the stated availability. Extrapolating, we get the Hours Outage = (4 + 447 + 447 + 447) ×1 hour + (447 + 8929) × 5 minutes = 1345 + 781 = 2126 hours. This yields 75.7% availability, or 24.3% probability that a server is down. To calculate 99% cluster availability, or 1% probability that the cluster doesn't have enough capacity to service the requests, we calculate how many simultaneous failures we must support. Thus we need (0.243)^*n* = 0.001, where *n* is the number of simultaneous failures. Thus a standard cluster needs approximately 5 extra servers to maintain the 30,000 queries per second at 99.9% availability. Comparatively, the cluster with "small" servers will have an availability based on: Hours Outage = (4 + 1041 + 1041 +1041 × 1.3) ×1 hour + (1041 × 1.3 + 20,833) × 5 minutes = 3439 +1849 = 5288 hours. This yields 39.6% availability. Following the equations from above, we need (0.604)^*n* = 0.001. Thus we need approximately 14 extra servers to maintain the 30,000 queries per second at 99.9% availability. If each baseline server cost $2000, the "small" servers must be $857.00 or less to achieve a cost advantage.

c. In this problem, the small servers provide 30% of the performance of the baseline servers. With 2400 servers, we need 8000 small servers, assuming linear scaling with node size. At 85% scaling, we need 9412 servers, and at 60% scaling, we need 13,333 servers. Although computation throughput will improve as cluster sizes are scaled up, it may become difficult to partition the problem small enough to take advantage of all of the nodes. Additionally, the larger the cluster, there will potentially be more intra-array communication, which is significantly slower than intra-rack communication.

6.17 a. See Figure S.1. In terms of latency, the devices perform more favorably than disk and less favorably than DRAM across both the array and rack. In terms of bandwidth, these devices perform more poorly than both DRAM and disk across the array and rack.

b. Software might be changed to leverage this new storage by storing large amounts of temporary data to Flash or PCIe devices when operating at the local

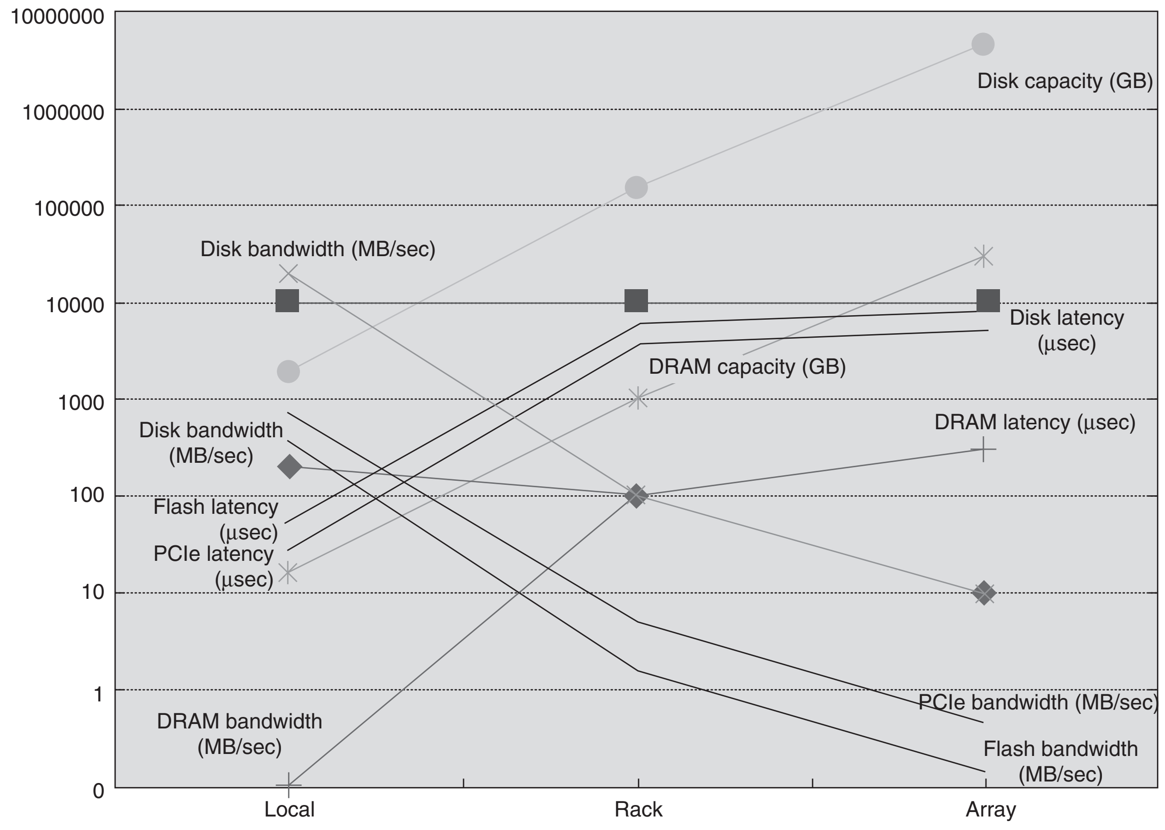


Figure S.1 Graph of latency, bandwidth, and capacity of the memory hierarchy of a WSC.

level to benefit from the reduced latency and increased bandwidth compared to disk, and storing permanent data in disk to benefit from the better bandwidth the disk has to offer over the rack and array levels—where data will ultimately be fetched from.

c. (This question refers to a different problem.)

d. Flash memory performs well for random accesses to data, while disks perform well when streaming through sequential data. A cloud service provider might look at the different types of workloads being run on his or her data center and offer as an option to his or her customers the option of using Flash memory to improve the performance of workloads which perform many random accesses to data.

6.18 a. For total storage capacity, let's assume the average size of a Netflix movie is 75 minutes (to take into account some 30-minute television shows and 2-hour movies). 75 minutes in seconds is $75\text{m} \times 60\frac{\text{s}}{\text{m}} = 4500\text{s}$. Taking into account each of the playback formats, per title, the storage requirements for each of the video formats is $\left(500\frac{\text{Kb}}{\text{s}} \times 4500\text{s} \times \frac{1\,\text{MB}}{8\times 1000\,\text{Kb}} = 281.25\text{MB}\right)$ + $\left(1000\frac{\text{Kb}}{\text{s}} \times 4500\text{s} \times \frac{1\,\text{MB}}{8\times 1000\,\text{Kb}} = 562.5\text{MB}\right)$ + $\left(1600\frac{\text{Kb}}{\text{s}} \times 4500\text{s} \times \frac{1\,\text{MB}}{8\times 1000\,\text{Kb}} = 900\text{MB}\right)$ + $\left(2200\frac{\text{Kb}}{\text{s}} \times 4500\text{s} \times \frac{1\,\text{MB}}{8\times 1000\,\text{Kb}} = 1237.5\text{MB}\right) = 2981.25\text{MB}$. Given that there were 12,000 titles, the total storage capacity would need to be at least 35,775,000 MB.

We know that there were 100,000 concurrent viewers on the site, but we do not know what playback format each of them was streaming their title in. We assume a uniform distribution of playback formats and get an average playback bitrate of (500 + 1000 + 1600 + 2200)/4 = 1325 kbps. This makes for an average I/O and network bandwidth of 100,000 × 1325 =132,500,000 kbps.

b. Per user, the access pattern is streaming, as it is unlikely a user will watch more than one movie concurrently. Temporal and spatial locality will be low, as titles will simply be streamed in from disk (unless the user chooses to replay a part of a title). The exact working set size will depend on the user's desired playback bitrate and the title, but in general, for the files that Netflix hosts, will be large.

Per movie, assuming more than one person is watching different parts of the movie, the access pattern is random, as multiple people will be reading different parts of the movie file at a time. Depending on how closely in time many users are to one another in the movie, the movie data may have good temporal and spatial locality. The working set size depends on the movie and title, but will be large.

Across all movies, assuming many people are watching many different movies, the access pattern is random. There will be little temporal or spatial locality across different movies. The working set size could be very large depending on the number of unique movies being watched.

c. In terms of both performance and TCO, DRAM is the greatest, then SSDs, then hard drives. Using DRAM entirely for storing movies will be extremely costly and may not deliver much improved performance because movie streaming is predominantly limited by network bandwidth which is much lower than DRAM bandwidth. Hard drives require a lower TCO than DRAM, but their bandwidth may be slower than the network bandwidth (especially if many random seeks are being performed as may be the case in the Netflix workload). SSDs would be more expensive, but could provide a better balance between storage and network bandwidth for a Netflix-like workload.

6.19 a. Let's assume that at any given time, the average user is browsing MB of content, and on any given day, the average user uploads MB of content.

b. Under the assumptions in Part a., the amount of DRAM needed to host the working set of data (the amount of data currently being browsed), assuming no overlap in the data being viewed by the users, is 100,000d MB = 100d GB. 96 GB of DRAM per server means that there must be (100d GB)/96 GB ≈ ϒ1.04d/ servers to store the working set of data. Assuming a uniform distribution of user data, every server's memory must be accessed to compose a user's requested page, requiring 1 local memory access and Υ1.04*d*/ remote accesses.

c. The Xeon processor may be overprovisioned in terms of CPU throughput for the memcached workload (requiring higher power during operation), but may also be provisioned to allow for large memory throughput, which would benefit the memcached workload. The Atom processor is optimized for low power consumption for workloads with low CPU utilization like memcached, however, the Atom may not be fit for high memory throughput, which would constrain the performance of memcached.

d. One alternative is to connect DRAM or other fast memories such as Flash through other device channels on the motherboard such as PCIe. This can effectively increase the capacity of the system without requiring additional CPU sockets. In terms of performance, such a setup would most likely not be able to offer the same bandwidth as traditionally-connected DRAM. Power consumption depends on the interface being used; some interfaces may be able to achieve similar power consumptions as traditionally-attached DRAM. In terms of cost, such a device may be less expensive as it only involves adding additional memory (which would be purchased in any event) without adding an additional CPU and socket to address that memory. Reliability really depends on the device being used and how it is connected.

e. In case (1), co-locating the memcached and storage servers would provide fast access when data needs to be brought in or removed from memcached, however, it would increase the power density of the server and require more aggressive cooling solutions and would be difficult to scale to larger capacities. Case (2) would require higher latencies than case (1) when bring data in or removing data from memcached, but would be more amenable to scaling the capacity of the system and would also require cooling less power per volume. Case (3) would require very long access latencies to bring data in and remove data from memcached and may be easier to maintain than case (2) due to the homogeneity of the contents of the racks (hard drive and memory failures will mainly occur in one portion of the data center).

6.20 a. We have 1 PB across 48,000 hard drives, or approximately 21 GB of data per disk. Each server has 12 hard disks. Assuming each disk can maintain its maximum throughput, and transfers data entirely in parallel, the initial reading of data at each local node takes 21 GB/200 MB/s = 105 seconds. Similarly writing data back will also take 105 seconds, assuming the data is perfectly balanced among nodes and can be provided at the total aggregate bandwidth of all of the disks per node.

b. With an oversubscription ratio of 4, each NIC only gets 1/4 Gb/s, or 250 Mb/s. Assuming all data must be moved, the data is balanced, and all of the movement can be done in parallel, each server must handle 1 PB/4000 = 250 GB of data, meaning it must both send and receive 250 GB of data during the shuffle phase $(2 \times 250\ \text{GB})/(2 \times 250\ \text{Mb/s}) = 8000$ seconds, or 2.22 hours.

c. If data movement over the network during the shuffle phase is the bottleneck, then it takes approximately 6 hours to transfer 2 × 250 GB of data per server. This translates to each 1 Gb NIC getting approximately 93 Mb/s of bandwidth, or approximately 10:1 oversubscription ratio. However, these numbers assume only the data is being shuffled, no replicas are being sent over the network, and no redundant jobs are being used. All of these other tasks add network traffic without directly improving execution time, which implies the actual oversubscription ratio may be lower.

d. At 10 Gb/s with no oversubscription, we again calculate the transfer time as sending and receiving 250 GB of data. Assuming the server still has two NICs, we calculate the transfer time as (2 × 250 GB)/(2 × 10 Gb/s) = 200 seconds, or 0.06 hours.

e. There are many possible points to mention here. The massively scale-out approach allows for the benefits of having many servers (increased total disk bandwidth, computational throughput, any failures affect a smaller percentage of the entire cluster) and using lower-cost components to achieve the total performance goal. However, providing enough network bandwidth to each of the servers becomes challenging, especially when focusing on low-cost commodity components. On the other hand, a small-scale system offers potential benefits in ease of administration, less network distance between all the nodes, and higher network bandwidth. On the other hand, it may cost significantly more to have a small-scale system that provides comparable performance to the scale-out approach.

f. Some example workloads that are not communication heavy include computation-centric workloads such as the SPEC CPU benchmark, workloads that don't transfer large amounts of data such as web search, and single-node analytic workloads such as MATLAB or R. For SPEC CPU, a High-CPU EC2 instance would be beneficial, and for the others a high-memory instance may prove the most effective.

6.21 a. Each access-layer switch has 48 ports, and there are 40 servers per rack. Thus there are 8 ports left over for uplinks, yielding a 40:8 or a 5:1 oversubscription ratio. Halving the oversubscription ratio leads to a monthly Network costs of \$560,188, compared to the baseline of \$294,943; total costs are \$3,796,170 compared to \$3,530,926. Doubling the oversubscription ratio leads to monthly Network costs of \$162,321, and total costs of \$3,398,303.

b. With 120 servers and 5 TB of data, each server handles approximately 42 GB of data. Based on Figure 6.2, there is approximately 6:1 read to intermediate data ratio, and 9.5:1 read to output data ratio. Assuming a balanced system, each system must read 42 GB of data, of which 21 GB is local, 21 GB is remote (16.8 GB in the rack, 4.2 GB in the array). For intermediate data, each system must handle 7 GB of data, of which 4.9 GB is local, 2.1 GB is remote (1.9 GB in the rack, 0.2 GB in the array). Finally for writing, each system must handle 4.4 GB of data, of which 3.1 GB is local, 1.3 GB is remote (1.2 GB in the rack, 0.1 GB in the array). We will make the simplifying assumption that all transfers happen in parallel, therefore the time is only defined by the slowest transfer. For reading, we obtain the following execution times: local = 21 GB/200 MB/s = 105 seconds, rack = 16.8 GB/100 MB/s = 168 seconds, array = 4.2 GB/10 MB/s = 420 seconds. For intermediate data: local = 4.9 GB/200 MB/s = 24.5 seconds, rack =1.9 GB/100 MB/s = 19 seconds, array = 0.2 GB/10 MB/s = 20 seconds. For writing: local = 3.1 GB/200 MB/s = 15.5 seconds, rack = 1.2 GB/100 MB/s = 12 seconds, array = 0.1 GB/10 MB/s = 10 seconds. The total performance is thus 420 + 24.5 + 15.5 = 460 seconds. If the oversubscription ratio is cut by half, and bandwidth to the rack and array double, the read data time will be cut in half from 420 seconds to 210 seconds, resulting in total execution time of 250 seconds. If the oversubscription is doubled, the execution time is 840 + 40 + 24 = 904 seconds. Network costs can be calculated by plugging the proper numbers of switches into the spreadsheet.

c. The increase in more cores per system will help reduce the need to massively scale out to many servers, reducing the overall pressure on the network. Meanwhile, optical communication can potentially substantially improve the network in both performance and energy efficiency, making it feasible to have larger clusters with high bandwidth. These trends will help allow future data centers to be more efficient in processing large amounts of data.

6.22 a. At the time of print, the "Standard Extra Large", "High-Memory Extra Large", "High-Memory Double Extra Large", and "High-Memory Quadruple Extra Large", and "Cluster Quadruple Extra Large" EC2 instances would match or exceed the current server configuration.

b. The most cost-efficient solution at the time of print is the "High-Memory Extra Large" EC2 instance at $0.50/hour. Assuming the number of hours per month is 30 × 24 = 720, the cost of hosting the web site on EC2 would be 720 hours × $0.50/hour = $360.

c. Data transfer IN to EC2 does not incur a cost, and we conservatively assume that all 100 GB/day of data is transferred OUT of EC2 at a rate of $0.120/GB. This comes to a monthly cost of 30 days/month × 100 GB × $0.120/GB = $48/month. The cost of an elastic IP address is $0.01/hour, or $7.20/month. The monthly cost of the site is now $360 + $48 + $7.20 =$415.20.

d. The answer to this question really depends on your department's web traffic. Modestly-sized web sites may be hosted in this manner for free.

e. Because new arrivals may happen intermittently, a dedicated EC2 instance may not be needed and the cheaper spot instances can be utilized for video encoding. In addition, the Elastic Block Store could allow for gracefully scaling the amount of data present in the system.

6.23 a. There are many possible options to reduce search query latency. One option is to cache results, attempting to keep the results for the most frequently searched for terms in the cache to minimize access times. This cache could further improve latency by being located either in main memory or similar fast memory technology (such as Flash). Query results can be improved by removing bottlenecks as much as possible. For example, disk is likely to be a bottleneck if the search must access anything from disk due to its high access latency. Therefore one option is to avoid accessing disk as much as possible by storing contents either in main memory or nonvolatile solid state storage. Depending on how many queries each search server is receiving, the network could potentially be a bottleneck. Although the data transferred per search query is likely small (only a page of text and a few images is returned to the user), the number of concurrent connections may be quite high. Therefore a separate core to help network processing or a TCP offload engine may also provide benefits in reducing query latency.

b. Monitoring statistics would need to be at multiple levels. To understand where time is spent in low-level code in a single server, a tool such as *Oprofile* would help provide a picture of what code is taking the most time. At a higher level, a tool such as *sar* or *perf* would help identify the time the system is spending in different states (user versus kernel) and on different devices (CPU, memory, disk, network utilization). All of this information is critical to identifying which resources are the bottlenecks. At an even higher level, a network monitoring tool at the rack switch level can help provide a picture of network traffic and how servers are sending and receiving packets. All of this information must be combined at a cluster level to understand the bottlenecks for each search. This high level tool would need to be implemented through a cluster-level profiler that can collect the data for each individual node and switch, and aggregate them into a single report that can be analyzed.

c. We want to achieve an SLA of 0.1 second for 95% of the queries. Using the normal distribution to create a cumulative density function, we find that 95% of queries will require 7 disk accesses. If disk is the only bottleneck, we must have 7 accesses in 0.1 seconds, or 0.014 seconds per access. Thus the disk latency must be 14 ms.

d. With in-memory results caching, and a hit rate of 50%, we must only worry about misses to achieve the SLA. Therefore 45% of the misses must satisfy the SLA of <0.1 seconds. Given that 45 out of the remaining 50% of accesses must satisfy the SLA, we look at the normal distribution CDF to find the 45%/50% = 90th percentile. Solving for that, we find the 90th percentile requires approximately 6 disk accesses, yielding a required disk latency of 0.016 seconds, or 16 ms.

e. Cached content can become stale or inconsistent upon change to the state of the content. In the case of web search, this inconsistency would occur when the web pages are re-crawled and re-indexed, leading to the cached results becoming stale with respect to the latest results. The frequency depends on the web service; if the web is re-crawled every evening, then the cache would become inconsistent every night. Such content can be detected by querying the caches upon writing new results; if the caches have the data in their cache, then it must be invalidated at that time. Alternatively, the servers storing the persistent results could have a reverse mapping to any caches that may contain the content. If the caches are expected to frequently have their contents changed, this scheme may not be efficient due to the frequent updates at the persistent storage nodes.

6.24 a. Let's assume that most CPUs on servers operate in the range of 10%–50% utilization as in Figure 6.3, and that average server utilization is the average of these values, 30%. If server power is perfectly proportional to CPU utilization, the average PSU efficiency will be 75%.

b. Now, the same power will be supplied by two PSUs, meaning that for a given load, half of the power will be supplied by one PSU and the other half will be supplied by the other PSU. Thus, in our example, the 30% power load from the system will be divided between the two PSU, 15% on each. This causes the efficiency of the PSUs to each be 65%, now.

c. Let's assume the same average server utilization of 30% from Part a. In this case, each PSU will handle the load of $\frac{16}{6} = 2\frac{2}{3}$ servers for a PSU utilization of $30\% \times 2\frac{2}{3} = 8\%$. At this load, the efficiency of each PSU is at least 80%.

6.25 a. In Figure 6.25, we can compute stranded power by examining the normalized power of the CDF curve for a particular group of machines when the CDF is equal to 1. This is the normalized power when all of the machines are considered. As we can see from Figure 6.25B, at the cluster level, almost 30% of the power is stranded; at the PDU level, slightly more than 80% of the power is stranded; and at the rack level, around 5% of the power is stranded. At larger groups of machines, larger oversubscription may be a more viable option because of the lack of synchronization in the peak utilization across a large number of machines.

b. Oversubscription could cause the differences in power stranding at different groups of machines. Larger groups of machines may provide more opportunities for oversubscription due to their potential lack of synchronization of peak utilizations. Put another way, for certain workloads, it is unlikely that all machines will simultaneously require their peak rated power consumption at the cluster level, but at the per-server (PDU) level, maximum utilization is a likely occurrence.

c. From Figure 6.14, referenced in the case study, the total monthly cost of a data-center provisioned for peak capacity (100% utilization) is \$3,800,000. If, however, we provision the data-center for actual use (78% utilization), we would only require $N' = 0.78N$ machines, where N is the original number of servers in the data-center and the new utilization of each machine is 100% (note that we could instead choose to use any number of machines between $0.78N$ and N, but in this case we choose the minimum number of machines because server cost, CAPEX, dominates total cost).

With fewer machines, most costs in Figure 6.14 are reduced, but power use increases due to the increased utilization of the system. The average power utilization of the data-center increases by $100\% - 80\% = 20\%$. To estimate the effect on TCO, we conservatively scale only the cost of server hardware and power and cooling infrastructure by 0.78 and the cost of power use by 1.20 to get a new monthly cost of operation of ($\$2{,}000{,}000 \times 0.78$) +\$290,000 + ($\$765{,}000 \times 0.78$) + \$170,000 + ($\$475{,}000 \times 1.20$) + \$85,000 = \$3,271,700, for a savings of \$528,300 per month.

d. Let's assume there are X servers per PDU, Y servers per rack, and Z servers per array. At the PDU level, we can afford to oversubscribe by around 30%, for a total of $\leq 1.3Xf - X$ additional servers. Similarly, at the rack and cluster level, we can afford to oversubscribe by around $\leq 1.2Yf - Y$ and $\leq 1.05Zf - Z$ servers, respectively. The total number of additional servers we can employ is given by $(\leq 1.3Xf - X) + (\leq 1.2Yf - Y) + (\leq 1.2Zf - Z)$.

e. In order to make the optimization in Part d. work, we would need some way of throttling the utilization of the system. This may require additional support from the software or hardware and could result in increased service times, or, in the worst case, job starvation whereby a user's job would not be serviced at all.

f. Preemptive policies may lead to inefficient utilization of system resources because all resources are throttled by default; such policies, however, may provide better worst-case guarantees by making sure power budgets are not violated. This technique may be most beneficial in settings where a controlled power and performance environment is required.

Conversely, reactive policies may provide more efficient utilization of system resources by not throttling resource utilization unless violations occur. This, however, can lead to problems when contention for resources causes power budgets to be exceeded, resulting in unexpected (from an application's perspective) throttling. If the resource utilization of a workload is well understood, and this technique can safely be applied, it could beneficially provide applications with more resources than a preemptive policy.

g. Stranded power will increase as systems become more energy proportional. This is because at a given utilization, system power will be less for an energy proportional system than a nonenergy proportional system, such as that in Figure 6.4. Lower system power means that the same number of system at a given utilization will require even less power than what is provisioned, increasing the amount of stranded power and allowing even more aggressive oversubscribing to be employed.

6.26 a. First let's calculate the efficiency prior to the UPS. We obtain $99.7\% \times 98\% \times 98\% \times 99\% = 94.79\%$ efficient. A facility-wide UPS is 92% efficient, yielding a total efficiency of $94.79\% \times 92\% = 87.21\%$ efficiency. A battery being 99.99% efficient yields $94.79\% \times 99.99\% = 94.78\%$ efficiency, or 7.57% more efficient.

b. Assume that the efficiency of the UPS is reflected in the PUE calculation. That is, the case study's base PUE of 1.45 is with a 92% efficient facility-wide UPS and 87.21% efficiency. The batteries are 7.57% more efficient. All power must go through the transformations, thus the PUE is directly correlated with this efficiency. Therefore the new PUE is 1.34. This PUE reduces the size of the total load to 10.72 MW; the same number of servers is needed, so the critical load remains the same. Thus the monthly power costs are reduced from \$474,208 to \$438,234. The original total monthly costs were \$3,530,926.

Let us assume the facility-wide UPS is 10% of the total cost of the facility that is power and cooling infrastructure. Thus the total facility cost is approximately \$66.1 M without the UPS. We now need to calculate the new total cost of the facility (assuming a battery has the same depreciation as the facility) by adding the cost of the per-server battery to the base cost, and amortizing the new facility costs. By doing so, we find the batteries can cost $\sim$14% of the per-server cost and break even with the facility-wide UPS. If we assume a depreciation time the same as the server (and add the battery directly into the per-server cost), we find the batteries can cost $\sim$5% of the per-server cost and break even with the facility-wide UPS.

c. The per-server UPS likely has lower costs, given the cost of batteries, and reduces the total power needed by the entire facility. Additionally it removes a single point of failure by instead distributing the UPS. However, if the batteries cannot provide the other functionality that a UPS provides (such as power conditioning), there will still be a need for additional facility equipment. The management model likely becomes more complex, as the administrators must manage as many UPS's as there are servers, as opposed to a single facility-wide UPS (or two if redundancy is provided). In particular, they need to ensure that each of those UPS properly work, and message the server that the power has been interrupted in case special action must be taken (such as checkpointing state and shutting down servers)

6.27 a. Total operational power = (1 + Cooling inefficiency multiplier) $\times$ IT equipment power. For this problem, there is an 8 MW datacenter, with 80% power usage, electricity costs of \$0.10 per kWh, and cooling-inefficiency of 0.8. Thus the baseline costs are Total operational power $\times$ electricity costs =(1 + 0.8) $\times$ (8 MW $\times$ 80%) $\times$ \$0.10 per kWh = \$1152 per hour. If we improve cooling efficiency by 20%, this yields a cooling-inefficiency of $0.8 \times (1 - 20\%) = 0.64$, and a cost of \$1049.60 per hour. If we improve energy efficiency by 20%, this yields a power usage of $80\% \times (1 - 20\%) = 0.64$, and a cost of \$921.60 per hour. Thus improving the energy efficiency is a better choice.

b. To solve for this question, we use the equation $(1 + 0.8) \times (8\text{ MW} \times 80\% \times (1 - x)) \times \$0.10$ per kWh = \$1049.60, where x is the percentage improvement in IT efficiency to break even 20% cooling efficiency improvement. Solving, we find $x = 8.88\%$.

c. Overall, improving server energy efficiency will more directly lower costs rather than cooling efficiency. However, both are important to the total cost of the facility. Additionally, the datacenter operator may more directly have the ability to improve cooling efficiency, as opposed to server energy efficiency, as they control the design of the building (e.g., can use air cooling, run at increased temperatures, etc.), but do not directly control the design of the server components (while the operator can choose low power parts, they are at the mercy of the companies providing the low power parts).

6.28 a. The COP is the ratio of the heat removed (Q) to the work needed to remove the heat (W). Air returns the CRAC unit at 20°C, and we remove 10 kW of heat with a COP of 1.9. COP $= Q/W$, thus $1.9 = 10$ kW/W. $W = 5.26$ kJ. In the second example, we are still removing 10 kW of heat (but have a higher return temperature by allowing the datacenter to run hotter). This situation yields a COP of 3.1, thus $3.1 = 10$ kW/W. $W = 3.23$ kJ. Thus we save ~2 kJ by allowing the air to be hotter.

b. The second scenario is 3.23 kJ/5.26 kJ = 61.4% more efficient, providing 39.6% savings.

c. When multiple workloads are consolidated, there would be less opportunity for that server to take advantage of ACPI states to reduce power, as the server requires higher total performance. Depending on the most optimal point to run all servers (e.g., if running all servers at 70% utilization is better than half of the servers at 90% and half at 50%), the combination of the two algorithms may result in suboptimal power savings. Furthermore, if changing ACPI states changes the server's reporting of utilization (e.g., running in a lower-power state results in 80% utilization rather than 60% utilization at a higher state) then opportunities to consolidate workloads and run at better efficiency points will be missed. Thus using the two algorithms together may result in information being obscured between the two algorithms, and therefore missed opportunities for better power saving. Other potential issues are with algorithms that may have conflicting goals. For example, one algorithm may try to complete jobs as quickly as possible to reduce energy consumption and put the server into a low-power sleep mode (this "race-to-sleep" algorithm is frequently used in cell phones). Another algorithm may try to use low-power ACPI modes to reduce energy consumption. These goals conflict with each other (one tries to use maximum throughput for as brief as possible, the other tries to use minimum acceptable throughput for as long as possible), and can lead to unknown results. One potential way to address this problem is to introduce a centralized control plane that manages power within servers and across the entire datacenter. This centralized control plane could use the inputs that are provided to the algorithms (such as server utilization, server state, server temperature, datacenter temperature) and decide the appropriate action to take, given the actions available (e.g., use ACPI modes, consolidate workloads, migrate workloads, etc.).

6.29 a. We can see that although reducing the idle power can help when the server is not utilized, there is still a strong nonlinearity between 0% and 100% utilization, meaning that operating at other utilization points is likely less efficient than it can potentially be.

b. Using the numbers from column 7, the performance from column 2, and the power from the previous problems, and assuming we run the workload constantly for 1 hour, we get the following table.

		Power				Energy (WH)			Performance
	Servers	**Performance**	**Baseline**	**Half idle**	**Zero idle**	**Baseline**	**Half idle**	**Zero idle**	**Baseline**
0%	109	0	181	90.5	0	19729	9864.5	0	0
10%	80	290762	308	308	308	24640	24640	24640	23260960
20%	153	581126	351	351	351	53703	53703	53703	88912278
30%	246	869077	382	382	382	93972	93972	93972	213792942
40%	191	1159760	416	416	416	79456	79456	79456	221514160
50%	115	1448810	451	451	451	51865	51865	51865	166613150
60%	51	1740980	490	490	490	24990	24990	24990	88789980
70%	21	2031260	533	533	533	11193	11193	11193	42656460
80%	15	2319900	576	576	576	8640	8640	8640	34798500
90%	12	2611130	617	617	617	7404	7404	7404	31333560
100%	8	2889020	662	662	662	5296	5296	5296	23112160
Total:	1001					380888	371023.5	361159	934784150

c. One potential such nonlinear curve is shown here:
This curve yields the following table:

		Power			Energy (WH)		Performance
	Servers	**Performance**	**Baseline**	**Sublinear**	**Baseline**	**Sublinear**	**Baseline**
0%	109	0	181	181	19729	19729	0
10%	80	290762	308	221	24640	17680	23260960
20%	153	581126	351	261	53703	39933	88912278
30%	246	869077	382	301	93972	74046	214000000
40%	191	1159760	416	341	79456	65131	222000000
50%	115	1448810	451	381	51865	43815	167000000
60%	51	1740980	490	610	24990	31110	88789980
70%	21	2031260	533	620	11193	13020	42656460
80%	15	2319900	576	630	8640	9450	34798500
90%	12	2611130	617	640	7404	7680	31333560
100%	8	2889020	662	662	5296	5296	23112160
Total:	1001				380888	326890	935000000

6.30 a. Using Figure 6.26, we get the following table:

		Power			Energy (WH)		Performance
	Servers	**Performance**	**Case A**	**Case B**	**Case A**	**Case B**	**Baseline**
0%	109	0	181	250	19729	27250	0
10%	80	290762	308	275	24640	22000	23260960
20%	153	581126	351	325	53703	49725	88912278
30%	246	869077	382	340	93972	83640	214000000
40%	191	1159760	416	395	79456	75445	222000000
50%	115	1448810	451	405	51865	46575	167000000
60%	51	1740980	490	415	24990	21165	88789980
70%	21	2031260	533	425	11193	8925	42656460
80%	15	2319900	576	440	8640	6600	34798500
90%	12	2611130	617	445	7404	5340	31333560
100%	8	2889020	662	450	5296	3600	23112160
Total:	1001				380888	350265	935000000

b. Using these assumptions yields the following table:

		Power				Energy (WH)			Performance
	Servers	**Performance**	**Case A**	**Case B**	**Linear**	**Case A**	**Case B**	**Linear**	**Baseline**
0%	504	0	181	250	0	91224	126000	0	0
10%	6	290762	308	275	66.2	1848	1650	397.2	1744572
20%	8	581126	351	325	132.4	2808	2600	1059.2	4649008
30%	11	869077	382	340	198.6	4202	3740	2184.6	9559847
40%	26	1159760	416	395	264.8	10816	10270	6884.8	30153760
50%	57	1448810	451	405	331	25707	23085	18867	82582170
60%	95	1740980	490	415	397.2	46550	39425	37734	165393100
70%	123	2031260	533	425	463.4	65559	52275	56998.2	249844980
80%	76	2319900	576	440	529.6	43776	33440	40249.6	176312400
90%	40	2611130	617	445	595.8	24680	17800	23832	104445200
100%	54	2889020	662	450	662	35748	24300	35748	156007080
Total:	1000					352918	334585	223954.6	980692117

We can see that this consolidated workload is able to reduce the total energy consumption for case A, and provide slightly higher performance. We can see that the linear energy-proportional model provides the best results, yielding the least amount of energy. Having 0 idle power and linear energy-proportionality provides significant savings.

c. The data is shown in the table in b. We can see that the consolidation also provides savings for Case B, but slightly less savings than for Case A. This is due to the lower overall power of the Case B server. Again the linear energy-proportional model provides the best results, which is notable given its higher peak power usage. In this comparison, the 0 idle power saves a significant amount of energy versus the Case B servers.

6.31 a. With the breakdowns provided in the problem and the dynamic ranges, we get the following effective per-component dynamic ranges for the two cases:

	Effective dynamic range	
CPU	1.5	0.99
Memory	0.46	0.6
Disks	0.143	0.13
Networking/other	0.192	0.324
Total	2.295	2.044

b. From the table we can see the dynamic ranges are $2.295\times$ and $2.044\times$ for server 1 and server 2, respectively.

c. From a., we can see that the choice of components will affect the overall dynamic range significantly. Because memory and networking/other makes up a much larger portion of the second system its dynamic range is $\sim 10\%$ less than the first server, indicating worse proportionality. To effectively address server energy proportionality we must improve the dynamic range of multiple components; given memory's large contribution to power, it is one of the top candidates for improving dynamic range (potentially through use of different power modes).

6.32 *Note*: This problem cannot be done with the information given.

6.33 We can see from Figure 6.12 that users will tolerate some small amount of latency (up to $\sim 4\times$ more than the baseline 50 ms response time) without a loss in revenue. They will tolerate some additional amount of latency ($10\times$ slower than the baseline 50 ms response time) with a small hit to revenue (1.2%). These results indicate that in some cases there may be flexibility to trade off response time for actions that may save money (for example, putting servers into low-power low-performance modes). However, overall it is extremely important to keep a high level of service, and although user satisfaction may only show small changes, lower service performance may be detrimental enough to deter users to other providers. Avoiding downtime is even more important, as a datacenter going down may result in significant performance hits on other datacenters (assuming georeplicated sites) or even service unavailability. Thus avoiding downtime is likely to outweigh the costs for maintaining uptime.

6.34 a. The following table shows the SPUE given different fan speeds. SPUE is the ratio of total power drawn by the system to power used for useful work. Thus a baseline SPUE of 1.0 is when the server is drawing 350 W (its baseline amount).

Fan speed	Fan power	SPUE	TPUE, baseline PUE	TPUE, improved PUE
0	0	1	1.7	1.581
10000	25.54869684	1.072996	1.824094	1.696407
12500	58.9420439	1.168406	1.98629	1.84725
18000	209	1.597143	2.715143	2.525083

b. We see an increase from 1.073 to 1.168 for increasing from 10,000 to 12,500 rpm, or approximately a 8.9% increase. Going from 10,000 to 18,000 rpm results in a 48.8% increase in SPUE. We can see that the TPUE of the improved case at 18,000 rpm is significantly higher than the TPUE of the baseline case with the fans at 10,000 or 12,500 rpm. Even if the fans used 12,500 rpm for the higher temperature case, the TPUE would still be higher than the baseline with the fans at 10,000 rpm.

c. The question should be, "Can you identify another design where changes to TPUE are lower than the changes to PUE?" For example, in Exercise 6.26, one level of inefficiency (losses at the UPS) is moved to the IT equipment by instead using batteries at the server level. Looking at PUE alone, this design appears to significantly approve PUE by increasing the amount of power used at the IT equipment (thereby decreasing the ratio of power used at the facility to power used at the IT equipment). In actuality, this design does not enable any more work to be accomplished, yet PUE does not reflect this fact. TPUE, on the other hand, will indicate the decreased efficiency at the IT level, and thus TPUE will change less than PUE. Similarly, designs that change power used at the facility level to power used at the IT level will result in greater changes to PUE than TPUE. One other example is a design that utilizes very high power fans at the server level, and thereby reduces the speed of the fans in the facility.

6.35 a. The two benchmarks are similar in that they try to measure the efficiency of systems. However, their focuses differ. SPEC power is focused on the performance of server side Java (driven by CPU and memory), and the power used by systems to achieve that performance. JouleSort on the other hand is focused on balanced total system performance (including CPU, memory, and I/O) and the energy required to sort a fixed input size. Optimizing for SPEC power would focus on providing the peak performance at the lowest power, and optimizing only for the CPU and memory. On the other hand, optimizing for JouleSort would require making sure the system is energy efficient at all phases, and needs I/O performance that is balanced with the CPU and memory performance.

To improve WSC energy efficiency, these benchmarks need to factor in the larger scale of WSCs. They should factor in whole cluster level performance of up to several thousand machines, including networking between multiple systems potentially across racks or arrays. Energy should also be measured in a well defined manner to capture the entire energy used by the datacenter (in a similar manner to PUE). Additionally, the benchmarks should also take into account the often interactive nature of WSCs, and factor in client-driven requests which lead to server activity.

b. JouleSort shows that high performing systems often come with a nonlinear increase in power. Although they provide the highest raw throughput, the total energy is higher for the same amount of work done versus lower power systems such as a laptop or embedded class computer. It also shows that efficiencies across the total system are important to reduce energy; lower power DRAM and solid state drives instead of hard disks, both reduce total system energy for the benchmark.

c. I would try to optimize the use of the memory hierarchy by the benchmark to improve the energy efficiency. More effective use of the processor's cache will help avoid lengthy stalls from accessing DRAM. Similarly, more effective use of DRAM will reduce stalls from going to disk, as well as disk access power. I would also consider having the application take more advantage of low power modes and aggressively putting the system into the optimal power state given expected delays (such as switching the CPU to low power modes when disk is being accessed).

6.36 a. First calculate the hours of outage due to the various events listed in Figure 6.1. We obtain: Hours outage = (4 + 250 + 250 + 250) × 1 hour +(250) × 5 minutes = 754 hours + 20.8 hours = 774.8 hours. With 8760 hours per year, we get an availability of (8760 − 774.8)/8760 = 91.2% availability. For 95% availability, we need the cluster to be up 8322 hours, or 438 hours of downtime. Focusing on the failures, this means reducing from 750 hours of hardware-related failures to 438 hours, or 58.4% as many failures. Thus we need to reduce from 250 failures for hard drives, memories, and flaky machines (each) to 146 failures.

b. If we add the server failures back in, we have a total of 754 hours + 438 hours of unavailability. If none of the failures are handled through redundancy, it will essentially be impossible to reach 95% availability (eliminating each of the hardware-related events completely would result in 94.95% availability). If 20% of the failures are handled, we have a total of 754 + (250 + 5000 × (1 − 20%)) × minutes = 754 + 70.8 hours of unavailability. The other hardware events need to be reduced to 121 events each to achieve 95% availability. With 50% of failures handled the other events need to be reduced to 129 events each.

c. At the scale of warehouse-scale computers, software redundancy is essential to provide the needed level of availability. As seen above, achieving even 95% availability with (impossibly) ultra-reliable hardware is essentially impossible due failures not related to hardware. Furthermore, achieving that level of reliability in hardware (no crashes ever due to failed components) results in extremely expensive systems, suitable for single use cases and not large scale use cases (for example, see HP's NonStop servers). Thus software must be used to provide the additional level of redundancy and handle failures.

However, having redundancy at the software level does not entirely remove the need for reliable systems. Having slightly less reliable servers results in more frequent restarts, causing greater pressure on the software to maintain a reliable entire system. If crashes are expected to be frequent, certain software techniques may be required, such as frequent checkpointing, many redundant computations, or replicated clusters. Each technique has its own overheads associated with it, costing the entire system performance and thereby increasing operating costs. Thus a balance must be struck between the price to achieve certain levels of reliability and the software infrastructure needed to accommodate the hardware.

6.37 a. There is a significant price difference in unregistered (non-ECC) and registered (ECC supporting) DDR3 memories. At the sweet spot of DDR3 (approximately 4 GB per DIMM), the cost difference is approximately 2 × higher for registered memory.

Based on the data listed in Section 6.8, a third of the servers experience DRAM errors per year, with an average of 22,000 correctable errors and 1 uncorrectable error. For ease of calculation, we will assume that the DRAM costs are for chipkill supporting memory, and errors only cause 5 minutes of downtime due to a reboot (in reality, some DRAM errors may indicate bad DIMMs that need to be replaced). With ECC, and no other failures, 1/3 of the servers are unavailable for a total of 5 minutes per year. In a cluster of 2400 servers, this results in a total of 66.67 hours of unavailability due to memory errors. The entire cluster has a total of 21,024,000 hours of runtime per year, yielding 99.9997% availability due to memory errors. With no error correction, 1/3 of the servers are unavailable for 22,000 × 5 minutes per year. In a cluster of 2400 servers, this results in a total of 1,466,666 hours of downtime per year. This yields 93.02% availability due to memory errors. In terms of uptime per dollar, assuming a baseline DDR3

DIMM costs $50, and each server has 6 DIMMs, we obtain an uptime per dollar of 14.60 hours/$ for the ECC DRAM, and 27.16 hours/$ for the non-ECC DRAM. Although the non-ECC DRAM has definite advantages in terms of uptime per dollar, as described in Section 6.8 it is very desirable to have ECC RAM for detecting errors.

6.38 a. Given the assumed numbers ($2000 per server, 5% failure rate, 1 hour of service time, replacement parts cost 10% of the server, $100/hour technician time), we obtain a $300 cost for fixing each server. With a 5% failure rate, we expect a $15 annual maintenance cost per server.

b. In the case of WSC, server repairs can be batched together. Repairing in such a manor can help reduce the technician visit costs. With software redundancy to overcome server failures, nodes can be kept offline temporarily until the best opportunity is available for repairing them with little impact to the overall service (minor performance loss). On the other hand, for traditional enterprise datacenter, each of those applications may have a strong dependency on individual servers being available for the service to run. In those cases, it may be unacceptable to wait to do repairs, and emergency calls to repair technicians may incur extra costs.

Additionally, the WSC model (where thousands of computers are running the same image) allows the administrator to more effectively manage the homogenous cluster, often enabling one administrator to handle up to 1000 machines. In traditional enterprise datacenters, only a handful of servers may share the same image, and even in those cases, the applications must often be managed on a per-server basis. This model results in a high burden on the administrator, and limits the number of servers they can manage to well under 100 per admin.

Chapter 7 Solutions

Case Study: Google's Tensor Processing Unit and Acceleration s9015 of Deep Neural Networks

7.1 a. Let n = max(M,N,K). Then the time complexity is O(n^3), or O(MNK). Arguments are O(n^2), or O(max(MK,KN,MN)). In other words, cubic compute for quadratic I/O. This means that at least for cubic matrix multiplication (where the inputs are roughly square), arithmetic intensity grows proportionally to n.

b. In C memory ordering, the rightmost index (last to be dereferenced) is adjacent in memory. So matrix A is $M \times K == 3 \times 5$, and the order of accesses to it looks like: [0,0],[0,1],[0,2],[0,3],[0,4], [1,0],[1,1],[1,2],[1,3],[1,4], [2,0],[2,1],[2,2],[2,3],[2,4]. In terms of addresses, that's the sequence of offsets 0, 1, 2,…,14 from the start of A, so A is accessed sequentially. C is 3×4, but is similarly accessed sequentially. B is 5×4, and is accessed in rows, so the access pattern looks like: [0,0],[1,0],[2,0],[3,0],[4,0], [0,1],[1,1],[2,1],[3,1],[4,1], [0,2],[1,2],[2,2],[3,2],[4,2], [0,3],[1,3],[2,3],[3,3],[4,3], [0,4],[1,4],[2,4],[3,4],[4,4]. This turns into a sequence of strided accesses to memory, at offsets: 0,5,10,15,20, 1,6,11,16,21, 2,7,12,17,22, 3,8,13,18,23, 4,9,14,19,24.

c. If we transpose matrix B, then its accesses become sequential in memory.

d. For this problem, we just need to modify the innermost loop to use the hardware_dot method:

```
for (int i =0; i < M; ++i) {
      for (int j = 0; j < N; ++j) {
            int k;
            for (k =0; k < K - 8; k += 8) {
                  hardware_dot(&c[i][j], &a[i][k], &b[i][k]);
            }
            for (; k < K; ++k) {
                  c[i][j] += a[i][k] * b[i][k];
            }
      }
}
```

The problem with hardware_dot itself, is that it needs to add up the results of eight different multiplications. This requires seven additions.

e. The easiest thing is to use scalar accesses to the B matrixes as part of the solution. To do this, we need to transpose our loops to make the indices of B fixed for our call to saxpy:

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

Then rewrite the i-based loop to call saxpy, which now takes b[k][j] as a scalar argument:

```
for (int j = 0; j < N; ++j) {
      for (int k = 0; k < K; ++k) {
            int i;
            for (i = 0; i < M - 8; i += 8) {
                  saxpy(&c[i][j], b[k][j], &a[i][k]);
            }
            for (; i < M; ++i) {
                  c[i][j] += a[i][k] * b[k][j];
            }
      }
}
```

7.2 a. For a single batch example, there are 2K = 2048 input values. So a batch size of 128 corresponds to 128 × 2048 = 256 KiB of space, so the corresponding sizes for larger batches are 512 KiB, 1024 KiB = 1 MiB, 2 MiB, and 4 MiB. Transfer time at a batch of 128 is (256 KiB × 8 bits/byte)/(100 Gibit/s) = (2 Mibit)/(100 Gibit/s) = roughly 20 microseconds. The larger batch sizes will take correspondingly 40, 80, 160, and 320 microseconds to transfer.

b. 20 MiB/30 GiB/s ⇒ roughly ⅔ of a millisecond, or 670 microseconds. Transferring one 256 × 256 tile is moving 64 KiB, which takes roughly 2 microseconds.

c. There are 256 × 256 = 65,536 ALUs, each of which performs two operations per cycle (one multiply and one add). So 65,536 × 2 × 700 MHz ⇒ 9.175e13, or about 92T operations/second.

d. It takes roughly 2 microseconds to load a weight tile. At 700 MHz, that's roughly 1400 cycles, or a batch size of 1400, to break even with the time to load a weight tile.

e. The Haswell x86 server computes at roughly 1 T FLOPS. The 20 M weights will require 40 M floating point operations, times batch = 128 gives 512 M operations total. If the Haswell server could hit its peak of 1 T FLOPS, it ought to take about 500 microseconds to perform this calculation. The K80 has a peak of roughly 3 T FLOPS, so it should take ⅓ the time if it were compute-dominated, or 167 microseconds. The TPU takes 670 microseconds just to load weights, as we computed in part 2b.

f. At batch size 128, we have 40 microseconds of transfer time and 670 microseconds of compute time (dominated by loading weights). 40/(670 + 40) gives about 6% PCIe utilization. For batch sizes 256, 512, 1024, and 2048, these become 11%, 19%, 32%, 48%, respectively.

g. At batch = 128, it takes 20 microseconds to transfer to and from the host CPU, and it also takes 20 microseconds to move values among the five cooperating TPUs. Compute latency is unchanged at 670 microseconds, but now we have $6 \times 20 = 120$ microseconds of PCIe latency, for a total of 790 microseconds of latency. This pipeline is five stages deep, so the throughput is roughly one batch of 128 every $790/5 = 160$ microseconds. The single TPU has a latency of 710 microseconds, which is slightly better, but throughput is also a batch of 128 every 710 microseconds. So throughput and latency would both be better if you operated each of the five TPUs to process a whole batch of 128 examples to completion, and replicated the weights across all five TPUs.

h. For the batch = 128, single-TPU configuration, the CPU needs to produce a batch of 128 examples every 710 microseconds. That takes $128 \times 50 = 6.4$ core-milliseconds of work every 710 milliseconds, or about $6400/710 = 9$ cores operating full time.

7.3 a. Everything gets shorter, because we're counting vectors or weight tiles:

```
read_host u#0,2048
read_weights w#0, 1
matmul a#0, u#0,2048 // weights are implicitly read from the FIFO.
activate u#2048, a#0, 2048
write_host, u#2048, 2048
```

The program reads 2048 256-element vectors. It uses 2048 256-element accumulators, and it writes back 2048 256-element vectors to the host.

b. We need four matmul instructions, each one adding into the same set of accumulators.

```
read_host u#0, 8192 // 4x as many things as input.
read_weights w#0, 4 // 4 weight tiles, each 256x256, for 1024x256 total.
matmul.= a#0, u#0, 2048 // weights are implicitly read from the FIFO.
matmul.+= a#0, u#2048, 2048 //weights are implicitly read from the FIFO.
matmul.+= a#0, u#4096, 2048 //weights are implicitly read from the FIFO.
matmul.+= a#0, u#6144, 2048 //weights are implicitly read from the FIFO.
activate u#8192, a#0, 2048
write_host, u#8192, 2048
```

The input activations come in packed in a particular order, so that there are 4 stripes of 2048 256-element vectors. The weight tiles need to be processed in corresponding order. Put differently, we can think of the input activations as a matrix that has shape (2048,1024), while the weights have shape (1024,256). We actually store the input activations in slices that hold the ranges (0:2048, 0:256), (0:2048, 256:512), (0:2048, 512:768), and (0:2048, 768:1024). If we store them in that order, then we'll need the corresponding weight slices (0:256,0:256), (256:512,0:256), (512:768,0:256), and (768:1024,0:256) of the original 1024×256 weight matrix.

c. For 256 × 512 matrix multiplication, we need to perform two passes over the input data, one for each slice of the 512 outputs per batch example. Here's the version that requires 4096, 256-entry accumulators:

```
read_host u#0, 2048 // 256-element inputs, so 2048 256-element vectors suffices.
read_weights w#0, 2 // 2 weight tiles, each 256x256, for 256x512 total.
matmul.= a#0, u#0, 2048
matmul.= a#2048, u#0, 2048 // Note that we reuse the input u#0 values.
activate u#2048, a#0, 4096
write_host, u#2048, 4096
```

Here's the version that requires just 2048, 256-entry accumulators:

```
read_host u#0, 2048 // 256-element inputs, so 2048 256-element vectors suffices.
read_weights w#0, 2 // 2 weight tiles, each 256x256, for 256x512 total.
matmul.= a#0, u#0, 2048
activate u#2048, a#0, 2048 // Activate batch of 2048, first 256/512 of output
write_host, u#2048, 2048
// In the actual TPU, we would need to use synchronization instructions here,
// to keep the second matmul from overwriting the accumulators until after the
// first activate completed.
matmul.= a#0, u#0, 2048 // Note that we reuse the input u#0 values.
activate u#4096, a#0, 2048
write_host, u#4096, 2048
```

Both versions of the program use just two 256 × 256 weight tiles. They should be stored in weight DRAM so that the left one comes first and the right one comes second.

d.
```
read_host u#0, 8192 // batch=2048, input depth=1024, /256
for vector length
read_weights w#0, 12 // 1024x768 / 256x256 ⇒ 4x3 = 12 weight tiles.

// First column of weights
matmul.= a#0, u#0, 2048 // Tile [0,0]
matmul.+= a#0, u#2048, 2048 // Tile [1,0]
matmul.+= a#0, u#4096, 2048 // Tile [2,0]
matmul.+= a#0, u#6144, 2048 // Tile [3,0]
activate u#8192, a#0, 2048 // Subtotal for the column is in a#[0:2048]

// Second column of weights
matmul.= a#0, u#0, 2048
matmul.+= a#0, u#2048, 2048
matmul.+= a#0, u#4096, 2048
matmul.+= a#0, u#6144, 2048
activate u#10240, a#0, 2048
```

```
// Third column of weights
matmul.= a#0, u#0, 2048
matmul.+= a#0, u#2048, 2048
matmul.+= a#0, u#4096, 2048
matmul.+= a#0, u#6144, 2048
activate u#12288, a#0, 2048

write_host, u#8192, 6144
```

In this example, we want to order the weight tiles so that the match the 4×3 order in which we perform the matrix multiplications. Each input activation gets read three times (once by each of three matmul instructions), and used 768 times, once for each output.

e. To read each of the input activations just once, we would need an architecture that had enough capacity to store all of the accumulators for all the columns of the weight matrix. So we would need batch_size × max_number_columns 32-bit accumulators. This is potentially much larger than the batch_size × 2 32-bit accumulators that the TPU's double-buffering scheme uses.

7.4 a. $(3/256) \times (48/256) = 144/65536 = 0.22\%$.

b. We need to find the smallest $x^2 > 1400$. $37 \times 37 = 1369$; $38 \times 38 = 1444$. So somewhere around a 37×37 or 38×38 image is the breakeven point.

c. Using this transformation, each weight matrix is now 48×48, so we get $(48/256) \times (48/256) = 2304/65536 = 3.5\%$.

7.5 a. We have $0.0 = 0 \times s + b$ and $6.0 = 255 \times s + b$. Solving, we get $b = 0$ and $s = 6.0/255 = 0.0235$.

b. There are only 256 representable values in this 8-bit representation. The spacing is s, or about 0.0235.

c. The quantization error looks like a sawtooth waveform, with zeros at $0.0 = 0 \times s, 1 \times s, 2 \times s, \ldots, 255 \times s = 6.0$. The maximum quantization error occurs exactly in between the zeros (which are the exactly representable values), or at $0.5 \times s, 1.5 \times s, 2.5 \times s, \ldots, 254.5 \times s$.

d. 1.0 turns out to be a point of maximum error, exactly spaced between the points represented by the integers 42 and 43 (0.988 and 1.012, respectively). So this is a trick question, You might validly choose either 42 or 43, depending on your preference for rounding mode. If you were choosing to round to the nearest even, you might pick 42. In that case, adding 1.0 to 1.0 on the TPU would give the representations of 42 and 42 as integers. Adding them gives the integer value 84. Converting back using *s*, we get 1.976.

e. We would expect the integers 0 and 255 to show up with ½ the usual frequency, because the interval of real numbers that map to these two integers are half the width of the intervals covered by the values 1, …, 254. So integers in the range

1, …, 254 will show up 1/255 = 0.00392 of the time, while 0 and 255 will show up 1/(2 × 255) = 1/510 = 0.00196 of the time.

f. For an 8-bit unsigned integer representation, we have $-1.0 = 0 \times s + b$ and $1.0 = 255 \times s + b$. So $b = -1.0$. And that gives us $s = 2.0/255 = 0.00784$. In this case, integer 0 represents −1.0 on the real number line. There is no exact representation for 0.0; 127 and 128 are equally spaced on either side of 0.0.

For an 8-bit twos-complement representation, we have $-1.0 = -128 \times s + b$ and $1.0 = 127 \times s + b$. Solving, we get $s = 2.0/255 = 0.00784$ and $b = 128 \times 2/255 - 1 = 0.00392$. In this case, there is again no exact integer representation for 0.0; –1 and 0 are equally spaced on either side of 0.0.

Not having an exact representation for 0.0 can be handled with reasonable expectations for how quantization errors propagate through a calculation. But it makes for strange behaviors of things that look like algebraic identities, such as multiplication by zero.

7.6 a. Just type the equations into the graphing calculator.

b. The equation is $\text{sigmoid}(x) = \frac{\tanh(x/2)}{2} + \frac{1}{2}$.

Substituting in the definition of tanh(x), we get $= \frac{1-\exp(-x)}{2\big(1+\exp(-x)+\frac{1}{2}}$.

$= \frac{1-\exp(-x)+(1+\exp(-x))}{2(1+\exp(-x))} = \frac{2}{2(1+\exp(-x))} = \frac{1}{1+\exp(-x)}$, which is the definition of sigmoid(x).

c. Thanks to the identity, we can scale and bias the results of one approximation to do the other.

d. Because tanh is odd, we can use half as many table entries in our approximation.

e. With a constant approximation, the best thing we can do is average the left and right endpoints of each interval, so that the worst-case error over the interval is half the difference between the left and right endpoints. For this constant approximation, the slope of tanh is steepest close to zero (at zero the slope is 1.0), so our worst-case constant approximation is in the interval [0.0,0.1), and has a value 0.049834.

f. One simple strategy is to just draw a line segment from the value of tanh at the left end of the interval to the value of tanh at the right end of the interval. This strategy is monotonic, but it's not the most accurate because tanh will take the form of a curve that is above this approximation line within a sub-interval. So all of the approximation errors will be positive. We could make the average approximation error be zero within each interval by increasing *b*. But increasing *b* in this way will mean that our approximation is no longer monotonic at the boundaries between intervals. There is no guarantee that the individual line segments meet up. Depending on the application, the monotonicity, or the average absolute value of the approximation error might be the more important feature to preserve.

i. In general, it's not possible to build the quadratic quantized approximation to be monotonic within the interval.

Exercises

7.7 a. sqrt(3600) = 60, so we can easily build a 60×60 multiplication array.

b. A non-square matrix multiplication array would mean that the input and output vector lengths for the multiplication unit would not be equal. This would mean that the architecture would need to support two different vector lengths. It's not clear how to handle the addressing implications of differently sized vector lengths, although it might be OK if one were a multiple of the other.

c. 3600 ALUs × 2 operations/MAC × 500 MHz ⇒ 3.6 T operations/second. That's in roughly the same range as a K80 GPU with turbo disabled.

d. 4096 × 2 × 350 ⇒ 2.9 T operations/second. That's a bad deal.

7.8 a. The ratio of core counts is 36 to 20, so scaling the $17,962 by 36/20 give us $32,332. Subtracting this CPU cost from the $184,780 gives us $152,448 for the non-CPU parts of the p2 system. Dividing this by 16 gives us a $9,528 three-year cost for a single K80 GPU.

b. For the p2 instance to be more cost effective, the ratio of performance (T/1) must exceed the ratio of costs ($184,780/$17,962 = 10.3). So T must be greater than or equal to 10.3.

The 20 cores of the c4 instance perform an aggregate of 600G FLOPS, or 0.6T FLOPS. A single K80 GPU, according to table 7.42, performs at about 2.8T FLOPS. So that's 0.6/2.8 = 21% of a single K*).

c. For the f1, 8 chips × 6840 slices × 2 operations per MAC × 500MHz ⇒ 55T operations/second. Cost/performance is $165,758/55 ⇒ $3013/3 TFLOP-years

For the p2, 16 K80s × 2.8 T FLOPS ⇒ 44.8T FLOPS. Cost/performance is $184,780/44.8 ⇒ $4124/3 TFLOP-years.

So given the assumption of substitutable arithmetic, the f1 solution is more cost effective, by a ratio of about 4124/3013 = 1.37.

7.9 a. We start with original data that's 64×32, so let's call that two-dimensional range image[0:64][0:32]. Then the eight cores want to import:

[0:18][0:18], with two pixels of zeroed halo on the left and bottom.

[14:34][0:18], with two pixels of zeroed halo on the bottom.

[30:50][0:18], with two pixels of zeroed halo on the bottom.

[46:64][0:18], with two pixels of zeroed halo on the right and bottom.

[0:18][14:32], with two pixels of zeroed halo on the left and top.

[14:34][14:32], with two pixels of zeroed halo on the top.

[30:50][14:32], with two pixels of zeroed halo on the top.

[46:64][14:32]], with two pixels of zeroed halo on the right and top.

b. If we change to a 3 × 3 stencil, then we only need one pixel depth of halo. So instead we have:

[0:17][0:17], with one pixel of zeroed halo on the left and bottom.

[15:33][0:17], with one pixel of zeroed halo on the bottom.

[31:49][0:17], with one pixel of zeroed halo on the bottom.

[47:64][0:17], with one pixel of zeroed halo on the right and bottom.

[0:17][15:32], with one pixel of zeroed halo on the left and top.

[15:33][15:32], with one pixel of zeroed halo on the top.

[31:49][15:32], with one pixel of zeroed halo on the top.

[47:64][15:32]], with one pixel of zeroed halo on the right and top.

c. In this case, each PE needs three pixels of halo, so the outermost ring of full PEs get used as simplified PEs, and we end up with a 14 × 14 region of full PEs. We then tile this over the 64 × 32 image, which means that we end up with ceiling (64/14) = 5 horizontal tiles and ceiling(32/14) = 3 vertical tiles, for a total of 15 tiles to process.

For utilization:
8 of the tiles use all 14 × 14 PEs
2 tiles on the right side of the image end up using just 8 × 14 of their PEs.
4 tiles on the top of the image use 14 × 4 of their PEs.
1 tile on the upper right of the image uses just 4 × 4 of its PEs.

To process the 15 tiles on 8 PEs, we need to perform two passes, where we process 8 tiles on the first pass and 7 tiles on the second pass.

So out of two passes over 8 tiles of 16 × 16 PEs, we have 2 × 8 × 16 × 16 = 4096 slots where we might use a full PE on a tile. We actually use 8 × 14 × 14 + 2 × 8 × 14 + 4 × 14 × 4 + 1 × 4 × 4 = 2032. That gives us 2032/4096 = 49.6% utilization.

7.10 a. In this case, each core can access its local memory at the full bandwidth B. The links from core to switch and switch to SRAM will each run at bandwidth B, while the inter-switch units won't be used.

b. In this case, all links will use bandwidth B, so each core can still use the full bandwidth B in this off-by-one case. To support this case, the switch must support full pairwise bandwidth, connecting its left switch neighbor to the SRAM at full bandwidth B while simultaneously connecting its core to its right switch neighbor at the full bandwidth B. If the switch can only process B bandwidth, then it would become the bottleneck, and the links would operate at B/2.

c. For the off-by-two pattern, the switch-to-switch links are oversubscribed by a factor of 2 (because each communication path requires two switch hops between core and memory). So each link gives B/2 bandwidth to each of the two cores that it serves, for a total of B bandwidth. The core-to-switch and switch-to-SRAM links run at B/2 bandwidth.

d. For the uniform random memory access pattern, suppose the processor ends up using P bandwidth to memory in total. The cases to consider are:

From the processor to the local SRAM, P/8 of the requested bandwidth uses no switch-to-switch links.

For the two cases where the processor accesses an off-by-one SRAM, we need one switch-to-switch link for each direction.

For the two cases where the processor accesses an off-by-two SRAM, we need two switch-to-switch links for each direction.

For the two cases where the processor accesses an off-by-three SRAM, we need three switch-to-switch links for each direction.

There's just one case where the processor accesses an off-by-four SRAM, but there are two paths, each with three switch-to-switch links that can serve the traffic.

So just focusing on one direction, P/2 of the bandwidth will need to use 1 + 2 + 3 + 4/2 = 8 link-hops worth of bandwidth. The processor can actually use P = B bandwidth with this memory access pattern.

7.11 a. One angstrom is 1×10^{-10} meters, so 64 angstroms is 6.4×10^{-9} meters. So the computer is about 150 million times larger than the simulated physical system.

b. 2.5 femtoseconds is 2.5×10^{-15} second, while 10 microseconds is $10e-6$ second. So the computer is about 4 billion times slower than the simulated physical system.

c. The speed of light constrains both the original physical system and the computer that simulates it. So if we made the computer larger than it was slower, the physical system would be able to exchange information across its volume of space more quickly than the simulating computer could exchange information across its volume of space. You can't do that with an accurate simulation of the physics. But if the computer is slower than it is larger, there's enough time to propagate information across the simulation at least as quickly as it propagates across the original physical system.

d. A warehouse might be thought of as being 100 meters on a side. So it might not quite work to have the computer be 15 billion times larger than the physical system while still trying to be only 4 billion times slower. A world-spanning Cloud server is much bigger (Earth's diameter is 12.7 Megameters), which would seem to make such a tightly coupled simulation impractical.

7.12 a. The diameter of the communication network is 12, with a $4 \times 4 \times 4$ path being the shortest distance between node (0,0,0) and node (4,4,4). Each node in the machine has a single "antipodal" node that is 12 hops away. The shortest broadcast latency would be 600 ns.

b. Reduction to a single node also requires 600 ns.

c. Interestingly, the all-reduce pattern also takes 600 ns of time. In such a pattern, each node broadcasts its values to each of its neighbors in the X dimension (i.e., that share the same Y and Z coordinates). Reaching the farthest such node is 4 hops, for 200 ns. Each node in the machine adds up the 7 values it has received from its X-dimension neighbors with its local value to get the total for all 8 nodes. So now we have 64 different 8-node sums, where groups of 8 nodes hold the same value. If we repeat the per-dimension broadcast-and-reduce in Y, we end up with 8 different 64-node sums, where all nodes in an XY plane of the 3D torus hold the same value. Finally, we repeat the per-dimension broadcast-and-reduce in Z, and each node ends up with a local copy of the 512-node sum.

The bandwidth for the reduce-to-one or broadcast-from-one pattern is 511 hops.

The bandwidth for the all-reduce pattern is $3 \times (4 + 2 \times 3 + 2 \times 2 + 2 \times 1) = 48$ hops for each of the 512 nodes in the machine, or 24,576 hops.

Appendix A Solutions

A.1 The exercise statement gives CPI information in terms of five major instruction categories, with two subcategories for conditional branches. The five main categories are ALU, loads, stores, conditional branches, and jumps. ALU instructions are any that take operands from the set of registers and return a result to that set of registers. Load and store instructions access memory. Conditional branches instructions must be able to set the program counter to a new value based on a condition, while jump instructions always set the program counter to a new value.

Instruction	**Clock cycles**
All ALU operations	1.0
Loads	5.0
Stores	3.0
Conditional branches	
Taken	5.0
Not taken	3.0
Jumps	3.0

For astar and gcc, the average instruction frequencies are shown in Figure S.1.

The effective CPI for programs in Figure A.29 is computed by combining instruction category frequencies with their corresponding average CPI measurements.

$$\begin{aligned}\text{Effective CPI} &= \sum \text{Instruction category frequency} \times \text{Clock cycles for category}\\ &= (0.41)(1.0) + (0.225)(5.0) + (0.145)(3.0)\\ &\quad + (0.19)[(0.6)(5.0) + (1-0.6)(3.0)] + (0.03)(3.0)\\ &= 2.86\end{aligned}$$

Instruction category	**astar**	**gcc**	**Average of astar and gcc**
ALU operations	46%	36%	41%
Loads	28%	17%	22.5%
Stores	6%	23%	14.5%
Branches	18%	20%	19%
Jumps	2%	4%	3%

Figure S.1 RISC-V dynamic instruction mix average for astar and gcc programs.

Instruction category	bzip	hmmer	Average of bzip and hmmer
ALU operations	54%	46%	50%
Loads	20%	28%	24%
Stores	7%	9%	8%
Branches	11%	17%	14%
Jumps	1%	0%	0.5%
Other	7%	0%	3.5%

Figure S.2 RISC-V dynamic instruction mix average for bzip and hmmer programs.

A.2 For bzip and hmmer, the average instruction frequencies are shown in Figure S.2. The effective CPI for programs in Figure A.29 is computed by combining instruction category frequencies with their corresponding average CPI measurements. Note that the total instruction frequency of bzip adds up to 93%, so an additional category is considered to account for the remaining 7% of instructions, each requiring 3.0 clock cycles.

$$\begin{aligned}\text{Effective CPI} &= \sum \text{Instruction category frequency} \times \text{Clock cycles for category}\\ &= (0.5)(1.0) + (0.24)(5.0) + (0.08)(3.0) + (0.14)[(0.6)(5.0) + (1-0.6)(3.0)]\\ &\quad + (0.005)(3.0) + (0.035)(3.0)\\ &= 2.65\end{aligned}$$

A.3 For gobmk and mcf, the average instruction frequencies are shown in Figure S.3. The effective CPI for programs in Figure A.29 is computed by combining instruction category frequencies with their corresponding average CPI measurements. Note that the total instruction frequency of gobmk adds up to 99%, so an additional category is considered to account for the remaining 1% of instructions, each requiring 3.0 clock cycles.

Instruction category	gobmk	mcf	Average of gobmk and mcf
ALU operations	50%	29%	39.5%
Loads	21%	35%	28%
Stores	12%	11%	16.5%
Branches	14%	24%	19%
Jumps	2%	1%	1.5%
Other	1%	0%	0.5%

Figure S.3 RISC-V dynamic instruction mix average for gobmk and mcf programs.

$$\begin{aligned}\text{Effective CPI} &= \sum \text{Instruction category frequency} \times \text{Clock cycles for category}\\ &= (0.395)(1.0) + (0.28)(3.5) + (0.165)(2.8)\\ &\quad + (0.19)[(0.6)(4.0) + (1-0.6)(2.0)] + (0.015)(2.4) + (0.005)(3.0)\\ &= 2.5\end{aligned}$$

A.4 For perlbench and sjeng, the average instruction frequencies are shown in Figure S.4.
The effective CPI for programs in Figure A.29 is computed by combining instruction category frequencies with their corresponding average CPI measurements.

$$\begin{aligned}\text{Effective CPI} &= \sum \text{Instruction category frequency} \times \text{Clock cycles for category}\\ &= (0.475)(1.0) + (0.22)(3.5)\\ &\quad + (0.105)(2.8) + (0.15)[(0.6)(4.0) + (1-0.6)(2.0)] + (0.05)(2.4)\\ &= 2.14\end{aligned}$$

A.5 Take the code sequence one statement at a time.

A = B + C;	The operands here are given, not computed by the code, so copy propagation will not transform this statement.

B = A + C;
= B + C + C;
	A is a computed operand, so transform the code by substituting A = B + C.
	The work has increased by one addition operation.

D = A – B;
 	
= (B + C) – (B + C + C);
 	
= –C;
	Both operands are computed, so transform the code by substituting for A and B.
	The work has increased by three addition operations, but expression can be simplified algebraically.
	The work has been reduced from a subtraction to a negation.

The copy propagation technique presents several challenges for compiler optimizations. Compilers need to consider how many levels of propagations are sufficient in order to gain benefit from this technique. Note that copy propagation might increase the computational work of some statements. Selecting the group of statements for which to apply copy propagation is another challenge. The above

Instruction category	perlbench	sjeng	Average of perlbench and sjeng
ALU operations	39%	56%	47.5%
Loads	25%	19%	22%
Stores	14%	7%	10.5%
Branches	15%	15%	15%
Jumps	7%	3%	5%

Figure S.4 RISC-V dynamic instruction mix average for perlbench and sjeng programs.

	Instruction count (%)				
Optimization level	**Branches/calls**	**Loads/stores**	**Integer ALU ops**	**FP ALU ops**	**Size (bytes)**
0	13.33	55.58	30.15	0.95	56,576
1	21.58	36.59	37.10	4.73	60,704
2	21.94	38.28	35.29	4.49	60,704
3	21.65	37.79	37.63	2.93	72,992

Figure S.5 Instruction count and program size for mcf at different optimization levels. The optimization level refers to the value used for the compiler optimization flag, i.e., -O0, -O1, -O2, and -O3. The instruction mix is divided into four categories: branches and calls, loads and stores, integer ALU operations, and floating-point ALU operations. The size column corresponds to the size of the executable file after compilation. The experiments were performed for a 64-bit RISC-V processor using GNU C compiler 7.1.1.

suggests that writing optimizing compilers means incorporating sophisticated trade-off analysis capability to control any optimizing steps, if the best results are to be achieved.

A.6 The mcf program is designed for scheduling single-depot vehicle in public mass transportation and consists almost exclusively of integer operations. The mcf benchmark was selected from the SPEC CPU2017 suite to compare instruction mixes and size of program for a 64-bit RISC-V processor as GNU C compiler optimization levels where varied, see Figure S.5. Results show about a 20% reduction of loads/stores instructions when optimizations are enabled, most likely due to local optimizations and local register allocations. For the other categories the instruction count increases when optimizations are enabled. Also, the code size increases as the compiler applies more optimizations, such as, code scheduling, loop unrolling, and procedure integration.

Figure A.21 shows the instruction count for mcf from SPEC CPU2000 suite using an Alpha compiler. For no optimizations, the Alpha compiler generated more branch/call instructions but less load/store and integer ALU instructions than the GNU C RISC-V compiler. Overall Figure S.5 shows a higher percentage of optimized code when optimizations are enabled compared to Figure A.21.

A.7 a. This exercise serves to highlight the value of compiler optimizations. For this exercise registers are not used to hold updated values; values are stored to memory when updated and subsequently reloaded. Immediate instructions can be used to construct addresses because all addresses referenced can be represented with 16 bits or less. Figure S.6 shows one possible translation of the given fragment of C code.

The number of instructions executed dynamically is the number of initialization instructions plus the number of instructions executed by the loop:

$$\text{Instructions executed} = 2 + (9 \times 101) = 911$$

```
ex_A7_a:    ADD     A1,A0,A0        ;A0 = 0, initialize i = 0
            SD      A1,7000(A0)     ;store i
loop:       LD      A1,7000(A0)     ;get value of i
            LD      A2,3000(A1)     ;load B[i]
            LD      A3,5000(A0)     ;load C
            ADD     A4,A2,A3        ;B[i] + C
            SD      A4,1000(A1)     ;store A[i] ← B[i] + C
            ADD     A1,A1,8     ;increment i
            SD      A1,7000(A0)     ;store i
            ADD     A5,A0,808       ;is i at 101?
            BNE     A1,A5,loop      ;if not at 101, repeat loop
```

Figure S.6 RISC-V code to implement the C loop without holding register values across iterations.

The number of memory-data references is a count of the load and store instructions executed:

$$\text{Memory-data references executed} = 1 + (5 \times 101) = 506$$

Since RISC-V instructions are 4 bytes in size, the code size is the number of instructions times 4:

$$\text{Instruction bytes} = 11 \times 4 = 44\text{ bytes}$$

b. This problem is similar to part (a) but implemented in x86-64 instructions instead, see Figure S.7.

$$\text{Instructions executed} = 3 + (9 \times 101) = 912$$

$$\text{Memory-data references executed} = 3 + (5 \times 101) = 508$$

Intel x86-64 instructions vary in size from 1 up to 14 bytes. Assume each instruction requires 7 bytes (MOVQ actually requires 7 bytes).

$$\text{Instruction bytes} = 12 \text{ x } 7 = 84\text{ bytes}$$

```
ex_A7_b:    MOVQ  $0x0,%rax             ;rax = 0, initialize i = 0
            MOVQ  $0x0,%rbp             ;base pointer = 0
            MOVQ  %rax,0x1b58(%rbp)     ;store i
loop:       MOVQ  0x1b58(%rbp),%rax     ;get value of i
            MOVQ  0x0bb8(%rax),%rcx     ;load B[i]
            MOVQ  0x1388(%rbp),%rdx     ;load C
            ADDQ        %rcx,%rdx       ;B[i] + C
            MOVQ  %rdx,0x03e8(%rax)     ;store A[i] ← B[i] + C
            ADDQ        $8,%rax         ;increment i
            MOVQ  %rax,0x1b58(%rbp)     ;store i
            CMPQ        $0x0328,%rax            ;is i at 101?
            JNE         loop                    ;if not at 101, repeat loop
```

Figure S.7 Intel x86-64 code to implement the C loop without holding register values across iterations.

c. No solution provided.

d. No solution provided.

e. No solution provided.

A.8 a. The C code for color representation conversion of pixels from RGB is implemented using RISC-V instruction set, see Figure S.8. Assume the variable p is kept in memory address 7000.

The number of instructions executed dynamically is the number of initialization instructions plus the number of instructions executed by the loop:

$$\text{Instructions executed} = 2 + (40 \times 8) \ = 322$$

The number of memory-data references is a count of the load and store instructions executed:

$$\text{Memory-data references executed} = 1 + (8 \times 8) = 65$$

Since RISC-V instructions are 4 bytes in size, the code size is the number of instructions times 4:

$$\text{Instruction bytes} = 42 \times 4 = 168\ \text{bytes}$$

b. This problem is similar to part (a) but implemented in x86-64 instructions instead, see Figure S.9.

$$\text{Instructions executed} = 3 + (31 \times 8) \ = 251$$

$$\text{Memory-data references executed} = 3 + (8 \times 8) = 67$$

$$\text{Instruction bytes} = 34 \times 7 = 238\ \text{bytes (assume each instruction requires 7 bytes)}$$

The number of instructions executed in Figure S.9 is over 12 times as many as the MMX implementation in A.8. The ratio of memory-data references are similar.

A.9 This exercise focuses on the challenges related to instruction set encoding. The length of an instruction is 14 bits. There are 64 general-purpose registers, thus the size of the address fields is 6 bits for each register operand. Since there are instructions with two-addresses, then the 2 most significant bits are reserved for opcodes. The notation addr[13:0] is used to represent the 14 bits of an instruction.

a. First, we need to support 3 two-address instructions. These can be encoded as follows:

	addr[13:12]	**addr[11:6]**	**addr[5:0]**
3 two-address instructions	‘00’, ‘01’, ‘10’	‘000000’ to ‘111111’	‘000000’ to ‘111111’
other encodings	‘11’	‘000000’ to ‘111111’	‘000000’ to ‘111111’

```
ex_A8_a:    ADD     A1,A0,A0        ;A0 = 0, initialize p = 0
            SD      A1,7000(A0)     ;store p
loop:       LD      A1,7000(A0)     ;get value of p
            LD      A2,1000(A1)     ;load R[p]
            LD      A3,2000(A1)     ;load G[p]
            LD      A4,3000(A1)     ;load B[p]
            LI      A10,9798        ;load immediate
            LI      A11,19235       ;load immediate
            LI      A12,3736        ;load immediate
            MUL     A5,A10,A2       ;9798 * R[p]
            MUL     A6,A11,A3       ;19235 * G[p]
            MUL     A7,A12,A4       ;3736 * B[p]
            ADD     A8,A5,A6        ;first addition of Y[p]
            ADD     A9,A8,A7        ;second addition of Y[p]
            SRL     A5,A9,15        ;divide by 32768
            SD      A5,4000(A1)     ;store Y[p]
            LI      A10,-4784       ;load immediate
            LI      A11,9437        ;load immediate
            LI      A12,4221        ;load immediate
            MUL     A5,A10,A2       ;-4784 * R[p]
            MUL     A6,A11,A3       ;9437 * G[p]
            MUL     A7,A12,A4       ;4221 * B[p]
            SUB     A8,A5,A6        ;addition of U[p]
            ADD     A9,A8,A7        ;subtraction of U[p]
            SRL     A5,A9,15        ;divide by 32768
            ADD     A5,A5,128       ;add 128
            SD      A5,5000(A1)     ;store U[p]
            LI      A10,20218       ;load immediate
            LI      A11,16941       ;load immediate
            LI      A12,3277        ;load immediate
            MUL     A5,A10,A2       ;20218 * R[p]
            MUL     A6,A11,A3       ;16941 * G[p]
            MUL     A7,A12,A4       ;3277 * B[p]
            SUB     A8,A5,A6        ;first subtraction of V[p]
            SUB     A9,A8,A7        ;second subtraction of V[p]
            SRL     A5,A9,15        ;divide by 32768
            ADD     A5,A5,128       ;add 128
            SD      A5,6000(A1)     ;store V[p]
            ADD     A1,A1,8     ;increment p
            SD      A1,7000(A0)     ;store p
            ADD     A5,A0,64        ;is p at 8?
            BNE     A1,A5,loop      ;if not at 8, repeat loop
```

Figure S.8 RISC-V code for color representation conversion of pixels from RGB (red, green, blue) to YUV (luminosity, chrominance) with each pixel represented by 3 bytes.

```
ex_A8_b:      MOVQ    $0x0,%rax              ;rax = 0, initialize p = 0
              MOVQ    $0x0,%rbp              ;base pointer = 0
              MOVQ    %rax,0x1b58(%rbp)      ;store p
loop:         MOVQ    0x1b58(%rbp),%rax      ;get value of p
              MOVQ    0x03e8(%rax),%rbx      ;load R[p]
              MOVQ    0x07d0(%rax),%rcx      ;load G[p]
              MOVQ    0x0bb8(%rax),%rdx      ;load B[p]
              IMULQ   9798,%rbx,%r8          ;9798 * R[p]
              IMULQ   19235,%rcx,%r9         ;19235 * G[p]
              IMULQ   3736,%rdx,%r10         ;3736 * B[p]
              ADDQ          %r8,%r9                        ;first addition of Y[p]
              ADDQ          %r9,%r10                       ;second addition of Y[p]
              SARQ          15,%r10                ;divide by 32768
              MOVQ    %r10,0x0fa0(%rax)            ;store Y[p]
              IMULQ   -4784,%rbx,%r8         ;-4784 * R[p]
              IMULQ   9437,%rcx,%r9          ; 9437 * G[p]
              IMULQ   4221,%rdx,%r10         ; 4221 * B[p]
              SUBQ          %r8,%r9                ;subtraction of U[p]
              ADDQ          %r9,%r10               ;addition of U[p]
              SARQ          15,%r10                ;divide by 32768
              ADDQ          $128,%r10              ;add 128
              MOVQ    %r10,0x1388(%rax)      ;store U[p]
              IMULQ   20218,%rbx,%r8         ; 20218 * R[p]
              IMULQ   16941,%rcx,%r9         ; 16941 * G[p]
              IMULQ   3277,%rdx,%r10         ; 3277 * B[p]
              SUBQ          %r8,%r9                        ;first subtraction of V[p]
              SUBQ          %r9,%r10                       ;second subtraction of V[p]
              SARQ          15,%r10                ;divide by 32768
              ADDQ          $128,%r10                      ;add 128
              MOVQ    %r10,0x1770(%rax)            ;store V[p]
              ADDQ          $8,%rax                ;increment i
              MOVQ    %rax,0x1b58(%rbp)            ;store i
              CMPQ          $0x0040,%rax                   ;is i at 8?
              JNE           loop                           ;if not at 8, repeat loop
```

Figure S.9 Intel x86-64 code for color representation conversion of pixels from RGB (red, green, blue) to YUV (luminosity, chrominance) with each pixel represented by 3 bytes.

Hence, the one-address and zero-address instructions must be encoded using addr [13:12] = ‘11’. For the one-address instructions, the opcode is extended by using 63 of the possible 64 combinations from the bits in addr[11:6], that is, from ‘000000’ to ‘111110’. Consequently, the opcode of zero-address instructions is extended with the remaining combination of ‘111111’ from addr[11:6]. There are 45 zero-address instructions, so using ‘000000’ to ‘101100’ from addr[5:0] suffices for the encoding.

	addr[13:12]	addr[11:6]	addr[5:0]
3 two-address instructions	‘00’, ‘01’, ‘10’	‘000000’ to ‘111111’	‘000000’ to ‘111111’
63 one-address instructions	‘11’	‘000000’ to ‘111110’	‘000000’ to ‘111111’
45 zero-address instructions	‘11’	‘111111’	‘000000’ to ‘101100’
19 unused encodings	‘11’	‘111111’	‘101101’ to ‘111111’

b. The 3 two-address instructions can be encoded similar to part (a). The one-address and zero-address instructions must be encoded using addr[13:12] = ‘11’. In order to encode 65 one-address instructions, at least 7 bits are required, so addr[11:5] can be used for this encoding. Then addr[4:0] is left for extending the opcode of zero-address instructions. The maximum number of zero-address instructions that can be encoded for this processor is 2ˆ5 = 32, thus encoding 35 zero-address instructions is not possible.

c. Similar to part (a), the 3 two-address instructions can be encoded as follows:

	addr[13:12]	addr[11:6]	addr[5:0]
3 two-address instructions	‘00’, ‘01’, ‘10’	‘000000’ to ‘111111’	‘000000’ to ‘111111’

The one-address and zero-address instructions must be encoded using addr[13:12] = ‘11’. One-address instructions can be encoded using all but one of the possible combinations from addr[11:6], that is, from ‘000000’ to ‘111110’. The ‘111111’ combination is used to extend the opcode for zero-address instructions. The maximum number of one-address instructions that can be encoded for this processor is 2ˆ6-1 = 63.

	addr[13:12]	addr[11:6]	addr[5:0]
3 two-address instructions	‘00’, ‘01’, ‘10’	‘000000’ to ‘111111’	‘000000’ to ‘111111’
63 one-address instructions	‘11’	‘000000’ to ‘111110’	‘000000’ to ‘111111’
24 zero-address instructions	‘11’	‘111111’	‘000000’ to ‘010111’
40 unused encodings	‘11’	‘111111’	‘010111’ to ‘111111’

d. Similar to part (a), the 3 two-address instructions can be encoded as follows:

	addr[13:12]	addr[11:6]	addr[5:0]
3 two-address instructions	‘00’, ‘01’, ‘10’	‘000000’ to ‘111111’	‘000000’ to ‘111111’

The one-address and zero-address instructions must be encoded using addr[13:12] = '11'. In order to encode 65 zero-address instructions, at least 7 bits are required, so addr[6:0] can be used for this encoding. This means that two combinations from addr[11:6] need to be used for the zero-address instructions. The maximum number of one-address instructions that can be encoded for this processor is 2ˆ6-2 = 62.

	addr[13:12]	**addr[11:6]**	**addr[5:0]**
3 two-address instructions	'00', '01', '10'	'000000' to '111111'	'000000' to '111111'
62 one-address instructions	'11'	'000000' to '111101'	'000000' to '111111'
65 zero-address instructions	'11'	'111110' to '111111'	'000000' to '111111'
63 unused encodings	'11'	'111111'	'000001' to '111111'

A.10 a. 1. Stack

Code	**Register addresses**	**Memory addresses**	**Code size (bytes)**
PUSH A		1	9
PUSH B		1	9
ADD			1
POP C		1	9
Total	0	3	28

2. Accumulator

Code	**Register addresses**	**Memory addresses**	**Code size (bytes)**
LOAD A		1	9
ADD B		1	9
STORE C		1	9
Total	0	3	27

3. Register-memory

Code	**Register addresses**	**Memory addresses**	**Code size (bytes)**
LOAD R1, A	1	1	9.75
ADD R3, R1, B	2	1	10.5
STORE R3, C	1	1	9.75
Total	4	3	30

4. Register-register

Code	Register addresses	Memory addresses	Code size (bytes)
LOAD R1, A	1	1	9.75
LOAD R2, B	1	1	9.75
ADD R3, R1, R2	3		3.25
STORE R3, C	1	1	9.75
Total	5	3	32.5

b. 1. Stack

Code	Destroyed data	Overhead data (bytes)	Code size (bytes)	Moved memory data (bytes)
PUSH A			9	8
PUSH B			9	8
ADD	A and B		1	
POP C	C		9	8
PUSH E			9	8
PUSH A		8	9	8
SUB	A and E		1	
POP D	D		9	8
PUSH C		8	9	8
PUSH D		8	9	8
ADD	C and D		1	
POP F	F		9	8
Total		24	84	72

2. Accumulator

Code	Destroyed data	Overhead data (bytes)	Code size (bytes)	Moved memory data (bytes)
LOAD A			9	8
ADD B	A		9	8
STORE C			9	8
LOAD A	C	8	9	8
SUB E	A		9	8
STORE D			9	8
ADD C	D		9	8
STORE F			9	8
Total		8	72	64

3. Register-memory

Code	Destroyed data	Overhead data (bytes)	Code size (bytes)	Moved memory data (bytes)
LOAD R1, A			9.75	8
ADD R3, R1, B			10.5	8
STORE R3, C			9.75	8
SUB R5, R1, E			10.5	8
STORE R5, D			9.75	8
ADD R6, R3, D			10.5	8
STORE R6, F			9.75	8
Total		0	70.5	56

4. Register-register

Code	Destroyed data	Overhead data (bytes)	Code size (bytes)	Moved memory data (bytes)
LOAD R1, A			9.75	8
LOAD R2, B			9.75	8
ADD R3, R1, R2			3.25	
STORE R3, C			9.75	8
LOAD R4, E			9.75	8
SUB R5, R1, R4			3.25	
STORE R5, D			9.75	8
ADD R6, R3, R5			3.25	
STORE R6, F			9.75	8
Total		0	68.25	48

A.11 Advantages of increasing the number of registers:

1. Greater freedom to employ compilation techniques that consume registers, such as loop unrolling, common subexpression elimination, and avoiding name dependences
2. More locations that can hold values to pass to subroutines
3. Reduce need to store and re-load values

Disadvantages of increasing the number of RISC-V registers:

1. More bits needed to represent a register, thus increasing the overall size of an instruction or reducing the size of other fields in the instruction
2. More CPU state to save in the event of an exception or context switch
3. Requires wider paths and additional control logic in the CPU pipeline, which could slow down the clock speed
4. Increases chip area and power consumption
5. Additional instructions may be required to allow use of the new registers

A.12 This program illustrates how alignment impacts data structures in C/C++. Note that the struct definition is for C++ because it includes a bool type.

Data type	**Data size on 32-bit machine (bytes)**	**Data size on 64-bit machine (bytes)**
char	1	1
bool	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

Alignment policies are defined by architecture, operating system, compiler, and/or language specification. For example, Visual C/C++ on 32/64-bit machines aligns double data types to 8 bytes. GNU C/C++ on 32-bit machines aligns double types to 4 bytes (unless -malign-double flag is set) while on 64-bit machines aligns them to 8 bytes. For 32-bit machines let us consider that double types require a 4 byte alignment since they are managed by two 4 byte addresses. In general, a struct has the alignment of its widest scalar member. To comply with the alignment, members are packed and/or padded. In the given struct the widest member is the double type with 8 bytes, thus the struct has an alignment requirement of 4 bytes for 32-bit processors and 8 bytes for 64-bit processors.

On a 32-bit processor, the struct consists of $1+1+4+8+2+4+8+4+4+4=40$ bytes in total. Due to alignment requirements, the char and bool will be packed and padded with two bytes to complete 4 bytes. The first int and the double types already satisfy the alignment, so no packing or padding is required. The short will be padded with two bytes. The remaining members satisfy the alignment boundaries, so the actual size of the struct is 40 + 2 + 2 = 44 bytes. To find the minimum size required for a struct, the data members need to be arranged from widest to narrowest, see below. The rearranged struct will first contain the pointer and double types, and end with the bool and char types. This ordering will not require padding bytes, so the actual size of the struct is 40 bytes.

On a 64-bit processor, the struct consists of 1 + 1 + 4 + 8 + 2 + 4 + 8 + 8 + 8 + 4 = 48 bytes in total.

Due to alignment requirements, the char and bool will be packed and padded with two bytes and also packed with the first int. The double and pointer types do not need to be packed nor padded. The short will be padded with two bytes and packed with the float. The last int will require four padding bytes to fulfill an 8 byte boundary. Then, the actual size of the struct is 48 + 2 + 2 + 4 = 56 bytes. The minimum size of the struct after rearrangement is 48 bytes since no padding bytes are needed.

```
struct foo {
    float *fptr;
    char *cptr;
    double d;
    double g;
    float f;
    int x;
    int c;
    short e;
    bool b;
    char a;
};
```

A.13 No solution provided.

A.14 No solution provided.

A.15 No solution provided.

A.16 No solution provided.

A.17 No solution provided.

A.18 No solution provided.

A.19 1. Accumulator

Instruction	Comments	Instruction fetched (bytes)	Moved memory data (bytes)
LOAD B	;Acc ← Mem[B]	3	2
ADD C	;Acc ← Acc + Mem[C]	3	2
STORE A	;Mem[A] ← Acc	3	2
ADD C	;Acc ← Acc + Mem[C] ;A is passed as operand, storage within processor	3	2
STORE B	;Mem[B] ← Acc	3	2
NEGATE	;Acc ← –Acc ;B is passed as operand, storage within processor	3	
ADD A	;Acc ← Acc + Mem[A] ;A is loaded again from memory ;B is passed as operand, storage within processor	3	2
STORE D	;Mem[D] ← Acc	3	2
Total		24	14

2. Memory-memory

Instruction	Comments	Instruction fetched (bytes)	Moved memory data (bytes)
ADD A, B, C	;Mem[A] ← Mem[B] + Mem[C]	7	6
ADD B, A, C	;Mem[B] ← Mem[A] + Mem[C] ;C is loaded again from memory ;A is passed as operand, storage in memory	7	6
SUB D, A, B	;Mem[D] ← Mem[A] – Mem[B] ;A, B are loaded again from memory ;A, B are passed as operands, storage in memory	7	6
Total		21	18

3. Stack (TOS is the top of stack, BOS is the bottom of stack, and [] is an empty stack entry)

Instruction	Comments	Instruction fetched (bytes)	Moved memory data (bytes)
PUSH B	;TOS ← Mem[B], BOS ← []	3	2
PUSH C	;BOS ← TOS, TOS ← Mem[C]	3	2
ADD	;TOS ← TOS + BOS, BOS ← []	1	
POP A	;Mem[A] ← TOS, TOS ← []	3	2
PUSH A	;TOS ← Mem[A]	3	2

PUSH C	;BOS ← TOS, TOS ← Mem[C] ;C is loaded again from memory	3	2
ADD	;TOS ← TOS + BOS, BOS ← [] ;A is passed as operand, storage in memory	1	
POP B	;Mem[B] ← TOS, TOS ← []	3	2
PUSH B	;TOS ← Mem[B] ;B is loaded again from memory	3	2
PUSH A	;BOS ← TOS, TOS ← Mem[A] ;A is loaded again from memory	3	2
SUB	;TOS ← TOS – BOS, BOS ← [] ;A, B are passed as operands, storage in memory	1	
POP D	;Mem[D] ← TOS, TOS ← []	3	2
Total		30	18

4. Load-store

Instruction	Comments	Instruction fetched (bytes)	Moved memory data (bytes)
LOAD R1, B	;R1 ← Mem[B]	3.5	2
LOAD R2, C	;R2 ← Mem[C]	3.5	2
ADD R3, R1, R2	;R3 ← R1 + R2	2.5	
STORE R3, A	;Mem[A] ← R3	3.5	2
ADD R1, R3, R2	;R1 ← R3 + R2 ;A is passed as operand, storage within processor	2.5	
STORE R1, B	;Mem[B] ← R1	3.5	2
SUB R4, R3, R1	;R4 ← R3 – R1 ;A, B are passed as operands, storage within processor	2.5	
STORE R4, D	;Mem[D] ← R4	3.5	2
Total		25	10

Based on total memory traffic, the load-store architecture is the most efficient with 35 bytes for code and data movement. The next efficient architecture is the accumulator with 38 bytes. Note that both make use of registers in contrast with the other two architectures.

Considering 64-bit addresses and data operands, the load-store architecture remains the most efficient since the penalty for accessing registers (assuming 64 registers) is less than increasing all memory addresses and operands with a 6 bytes overhead.

A.20 In this problem one has to consider how to implement a looping structure for the different architecture styles presented in A.19. Let us assume that A, B, C, D, and i are held in memory using addresses 1000, 3000, 5000, 6000, and 7000 as in problem A.7.

1. Stack

   For the stack machine case, we will add an indirect addressing mode where an address can placed on the stack. If a PUSHIND (push with indirect addressing) is used, it will take the value from the top of the stack as the address for the push. For POPIND (pop with indirect addressing), the top of the stack is the address for the target and the next entry on the stack is the value to be saved. We also assume a JLT instruction exists that takes the top three elements from the stack and jumps to the target (top of stack) if the second element is greater or equal than the third element from the top of the stack.

```
ex_A20: PUSH 0 ;load 0 on top of stack
               POP i          ;store i
loop:          PUSH i   ;load i (offset for B)
               PUSH 3000      ;load address of B
               ADD            ;compute address of B[i]
               PUSHIND        ;load B[i]
               PUSH C  ;load C
               MUL            ;B[i] * C
               PUSH D  ;load D
               ADD            ;add D
               PUSH i   ;load i (offset for A)
               PUSH 1000      ;load address of A
               ADD            ;compute address of A[i]
               POPIND         ;A[i] ← B[i] * C + D
               PUSH i   ;load i
               PUSH 8   ;load 8
               ADD            ;increment i
               POP i          ;store i
               PUSH i   ;load i
               PUSH 808       ;load termination value
               PUSH loop      ;load loop target addresses
               JLT            ;is i at 101? if not at 101, repeat loop
```

2. Accumulator

   For the accumulator machine, we can use the accumulator to compute an address for indirect addressing when we read a value from memory. Unfortunately, when we want to compute an address for saving a value to memory we will lose the value to be saved while computing the address, so an indirect store becomes hard to express. One could either add another register to support indirect addressing, which makes the machine no longer fit into the accumulator category. Alternately, one can employ self-modifying code to compute the A[i] address and then update the immediate field for direct addressing. To do this, we can assume that each instruction includes two words, one for the opcode and the other for the immediate value. We can then use LOADIND and STOREIND instruction in which the value in the accumulator is used as the address for the load or store operation.

```
ex_A20: LOAD 0 ;load 0 into accumulator
               STORE i        ;store i
loop:          LOAD i   ;load i (offset for B)
               ADD 3000       ;compute address of B[i]
               LOADIND        ;load B[i]
               MUL C    ;B[i] * C
               ADD D    ;add D
               LOAD i   ;load i (offset for A)
               ADD 1000       ;compute address of A[i]
               STOREIND       ;A[i] ← B[i] * C + D
               LOAD i   ;load i
               ADD 8          ;increment i
               STORE i        ;store i
               CMP 808        ;is i at 101?
               JLT            ;if not at 101, repeat loop
```

3. Register-memory
   For the register-memory case, one could consider a variant of the x86-64 code from A.7b.
4. Register-register (RISC-V)

```
ex_A20:  ADD      A1,A0,A0         ;A0 = 0, initialize i = 0
           SD       A1,7000(A0)         ;store i
loop:      LD       A1,7000(A0)         ;get value of i
           LD       A2,3000(A1)         ;load B[i]
           LD       A3,5000(A0)         ;load C
           LD       A4,6000(A0)         ;load D
           MUL      A5,A2,A3            ;B[i] * C
           ADD      A6,A5,A4            ;add D
           SD       A6,1000(A1)         ;store A[i] ← B[i] * C + D
           ADD      A1,A1,8     ;increment i
           SD       A1,7000(A0)         ;store i
           ADD      A5,A0,808           ;is i at 101?
           BNE      A1,A5,loop          ;if not at 101, repeat loop
```

A.21 No solution provided.

A.22 a. The ASCII interpretation of the 64-bit word using Big Endian byte order is as follows:

52	49	53	43	56	43	50	55
R	I	S	C	V	C	P	U

b. The ASCII interpretation of the 64-bit word using Little Endian byte order is as follows:

50	55	56	43	53	43	52	49
P	U	V	C	S	C	R	I

c. Misaligned 2-byte words start at odd addresses. Storing the 64-bit double word in Big Endian byte order produces the following misaligned 2-byte words when read: 4953, 4356, and 4350.

d. Misaligned 4-byte words start at addresses that are multiples of 4. Storing the 64-bit double word in Big Endian byte order produces the following misaligned 4-byte words when read: 4953, 5343, 4356, 4350, and 5055.

e. Storing the 64-bit double word in Little Endian byte order produces the following misaligned 2-byte words when read: 5556, 4353, and 4352.

f. Storing the 64-bit double word in Little Endian byte order produces the following misaligned 4-byte words when read: 5556, 5643, 4353, 4352, and 5249.

A.23 No solution provided.

A.24 Instruction set architectures can be optimized for specific application domains and markets since the required features and capabilities vary amongst them. The following is a list of how an ISA is impacted depending on the platform used for corresponding typical applications.

1. For desktop computing, the speed of both the integer and floating-point units is important, but power consumption is not as relevant. The ISA has to consider the compatibility of machine code and the mix of software running on top of it (e.g., x86 code). The ISA design must be capable of handling a wide variety of general-purpose applications, while maintaining commodity pricing and placing constraints on the investment of resources for specialized functions.

2. For servers, database applications are typical, so the ISA should be efficient with integer computations and memory operations. Support for high throughput, multiple users, and virtualization is necessary. Dynamic voltage and frequency scaling may be added for reduced energy consumption.

3. In the cloud computing paradigm, an ISA should effectively support virtualization as well as integer and memory operations. Power savings and features for easy system monitoring are important factors for large data centers. Security and encryption of data are also important aspects the ISA should provide support for.

4. For embedded computing, power consumption and price are important factors. The ISA should provide capabilities for sensing and controlling devices using special instructions and/or registers, analog-to-digital and digital-to-analog converters, and interfaces to specialized, application-specific functions. The ISA may omit floating-point operations, virtual memory, caching, or other performance optimizations to save energy and cost.

Appendix B Solutions

B.1 a. Average access time = (1 − miss rate) * hit time + miss rate * miss time = 0.97 * 1 + 0.03 * 110 = 4.27 cycles.

b. Because of the randomness of the accesses, the probability an access will be a hit is equal to the size of the cache divided by the size of the array. Hit rate = 64 Kbytes/1 Gbyte = $2^{16}/2^{30} = 2^{-14} \approx 6 \times 10^{-5}$ hits/access.

Therefore

$$\text{average access time} = 6 \times 10^{-5} \times 1 + \left(1 - 6 \times 10^{-5}\right) \times 110 = 109.99\,\text{cycles}.$$

c. The access time when the cache is disabled is 105 cycles, which is less than the average access time when the cache is enabled and almost all the accesses are misses. If there is no locality at all in the data stream, then the cache memory will not only be useless—but it will also be a liability.

d. Assuming the memory access time with no cache to be T_{off}, with cache T_{on}, and the miss rate is m, the average access time (with cache on).

$$T_{\text{on}} = (1 - m)(T_{\text{off}} - G) + m(T_{\text{off}} + L).$$

The cache becomes useless when the miss rate is high enough to make T_{off} less than or equal to T_{on}.

At this point, we have

$$T_{\text{off}} \le (1 - m)(T_{\text{off}} - G) + m(T_{\text{off}} + L)$$

After some algebraic manipulation, the inequality reduces to $m \ge \left(\frac{G}{G+L}\right)$. For part (a), $G = 109$ and $L = 5$, a miss rate greater than or equal to 109/114 (~0.96) would render the cache useless.

B.2 a.

Cache block	Set	Way	Possible memory blocks
0	0	0	M0, M1, M2, …, M31
1	1	0	M0, M1, M2, …, M31
2	2	0	M0, M1, M2, …, M31
3	3	0	…
4	4	0	…
5	5	0	…
6	6	0	…
7	7	0	M0, M1, M2, ……, M31

Direct-mapped cache organization.

b.

Cache block	Set	Way	Possible memory blocks
0	0	0	M0, M2, …, M30
1	0	1	M0, M2, …, M30
2	0	2	M0, M2, …, M30
3	0	3	M0, M2, …, M30
4	1	0	M1, M3, …, M31
5	1	1	M1, M3, …, M31
6	1	2	M1, M3, …, M31
7	1	3	M1, M3, …, M31

Four-way set-associative cache organization.

B.3

	Power consumption weight (per way accessed)
Data array	20 units
Tag array	5 units
Miscellaneous array	1 unit
Memory access	200 units

Estimate the power usage in power units for the following configurations. We assume the cache is 4-way associative.

a. Read hit:
   - LRU: 4 accesses for arrays of data/tag/miscellaneous components → 4 * (20+5+1) = 104 power units.
   - FIFO and Random: 4 accesses for arrays data/tag components → 4 * (20+5) = 100 power units.

b. Read miss:
   - LRU: 4 accesses for arrays of data/tag/miscellaneous components → 4 * (20+5+1) = 104 power units.
   - FIFO: 4 accesses for arrays data/tag components + one access to FIFO pointer → 4 * (20+5) + 1 = 101 power units.
   - Random: 4 accesses for arrays data/tag components → 4 * (20+5) = 100 power units.

c. Read hit (split access):
   - LRU: 4 accesses for arrays of tag/miscellaneous components plus one access to hit data array → 4 * (5+1) + 20 = 44 power units.
   - FIFO, Random: 4 accesses for arrays of tag components plus one access to hit data array → 4 * (5) + 20 = 40 power units.

d. Read miss, split access (cost of line-fill ignored):
   LRU: 4 accesses for arrays of tag/miscellaneous components $\rightarrow 4 * (5+1) = 24$ power units.
   FIFO, Random: 4 accesses for arrays of tag components $\rightarrow 4 * (5) = 20$ power units.

e. Read hit, split access with way prediction hit:
   LRU: one access to arrays of tag/miscellaneous components plus one access to data array $\rightarrow (5+1)+20 = 26$ power units.
   FIFO, Random: one access to arrays of tag components plus one access to data array $\rightarrow 5+20 = 25$ power units.

f. Read hit, split access with way prediction miss:
   LRU: one access to arrays of tag/miscellaneous components, plus 4 accesses of tag/miscellaneous components, plus one access to data array $\rightarrow (5+1)+4 * (5+1)+20 = 50$ power units.
   FIFO, Random: access to arrays of tag components, plus 4 accesses of tag components, plus one access to data array $\rightarrow 5+4 * (5)+20 = 45$ power units.

g. Read miss, split access with way prediction miss (cost of line-fill ignored):
   LRU: one access to arrays of tag/miscellaneous components, plus 4 accesses of tag/miscellaneous components $\rightarrow (5+1)+4 * (5+1) = 30$ power units.
   FIFO: one access to arrays of tag components, plus 4 accesses to arrays of tag component, and one access to miscellaneous component $\rightarrow 5+4 * (5) + 1 = 26$ power units.
   Random: one access to arrays of tag components, plus 4 accesses to arrays of tag component, $\rightarrow 5+4 * (5) = 25$ power units.

h. For every access:
   P (way hit, cache hit) = 0.95
   P (way miss, cache hit) = 0.02
   P (way miss, cache miss) = 0.03
   LRU = (0.95 * 26 + 0.02 * 50 + 0.03 * 30) power units
   FIFO = (0.95 * 25 + 0.02 * 45 + 0.03 * 26) power units
   Random = (0.95 * 25 + 0.02 * 45 + 0.03 * 25) power units

B.4 a. Loop has only one iteration in which one word (4 bytes) would be written. According to the formula for the number of CPU cycles needed for writing B bytes on the bus $(10+5\left(\left\lceil\frac{B}{8}\right\rceil-1\right))$, the answer is 10 cycles.

b. For a write-back cache, 32 bytes will be written. According to the above formula the answer is $10 + 5(4 - 1) = 25$ cycles.

c. When PORTION is eight, eight separate word writes will need to be performed by the write-through cache. Each (see part a) costs 10 cycles. So the total writes in the loop will cost 80 CPU cycles.

d. X array updates on the same cache line cost 10X cycles for the write-through cache and 25 cycles for the write-back cache (when the line is replaced). Therefore, if the number of updates is equal to or larger than 3, the write-back cache will require fewer cycles.

e. Answers vary, but one scenario where the write-through cache will be superior is when the line (all of whose words are updated) is replaced out and back to the cache several times (due to cache contention) in between the different updates. In the worst case, if the line is brought in and out eight times and a single word is written each time, then the write-through cache will require 8 * 10 = 80 CPU cycles for writes, while the write-back cache will use 8 * 25 = 200 CPU cycles.

B.5 A useful tool for solving this type of problem is to extract all of the available information from the problem description. It is possible that not all of the information will be necessary to solve the problem, but having it in summary form makes it easier to think about. Here is a summary:

- CPU: 1.1 GHz (0.909 ns equivalent), CPI of 1.35 (excludes memory accesses)
- Instruction mix: 75% nonmemory-access instructions, 20% loads, 10% stores
- Caches: Split L1 with no hit penalty (i.e., the access time is the time it takes to execute the load/store instruction
  - ○ L1 I-cache: 2% miss rate, 32-byte blocks (requires 2 bus cycles to fill, miss penalty is 15 ns + 2 cycles
  - ○ L1 D-cache: 5% miss rate, write-through (no write-allocate), 95% of all writes do not stall because of a write buffer, 16-byte blocks (requires 1 bus cycle to fill), miss penalty is 15 ns + 1 cycle
- L1/L2 bus: 128-bit, 266 MHz bus between the L1 and L2 caches
- L2 (unified) cache, 512 KB, write-back (write-allocate), 80% hit rate, 50% of replaced blocks are dirty (must go to main memory), 64-byte blocks (requires 4 bus cycles to fill), miss penalty is 60 ns + 7.52 ns = 67.52 ns
- Memory, 128 bits (16 bytes) wide, first access takes 60 ns, subsequent accesses take 1 cycle on 133 MHz, 128-bit bus

a. The average memory access time for instruction accesses:
   - L1 (inst) miss time in L2: 15 ns access time plus two L2 cycles (two = 32 bytes in inst. cache line/16 bytes width of L2 bus) = 15 + 2 * 3.75 = 22.5 ns (3.75 is equivalent to one 266 MHz L2 cache cycle)
   - L2 miss time in memory: 60 ns + plus four memory cycles (four = 64 bytes in L2 cache/16 bytes width of memory bus) = 60 + 4 * 7.5 = 90 ns (7.5 is equivalent to one 133 MHz memory bus cycle)
   - Avg. memory access time for inst = avg. access time in L2 cache + avg. access time in memory + avg. access time for L2 write-back. =0.02 * 22.5 + 0.02 * (1 − 0.8) * 90 + 0.02 * (1 − 0.8) * 0.5 * 90 = 0.99 ns (1.09 CPU cycles).

b. The average memory access time for data reads:
   Similar to the above formula with one difference: the data cache width is 16 bytes which takes one L2 bus cycles transfer (versus two for the inst. cache), so
   - L1 (read) miss time in L2: 15 + 3.75 = 18.75 ns
   - L2 miss time in memory: 90 ns
   - Avg. memory access time for read = 0.02 * 18.75 + 0.02 * (1 − 0.8) * 90 + 0.02 * (1 − 0.8) * 0.5 * 90 = 0.92 ns (1.01 CPU cycles)

c. The average memory access time for data writes:
Assume that writes misses are not allocated in L1, hence, all writes use the write buffer. Also, assume the write buffer is as wide as the L1 data cache.
- L1 (write) time to L2: 15 + 3.75 = 18.75 ns
- L2 miss time in memory: 90 ns
- Avg. memory access time for data writes = 0.05 * 18.75 + 0.05 * (1 − 0.8) * 90 + 0.05 * (1 − 0.8) * 0.5 * 90 = 2.29 ns (2.52 CPU cycles)

d. What is the overall CPI, including memory accesses:
- Components: base CPI, Inst fetch CPI, read CPI or write CPI, inst fetch time is added to data read or write time (for load/store instructions).

$$\text{CPI} = 1.35 + 1.09 + 0.2 * 1.01 + 0.10 * 2.52 = 2.84\,\text{cycles/inst.}$$

B.6 a. "Accesses per instruction" represents an average rate commonly measured over an entire benchmark or set of benchmarks. However, memory access and instruction commit counts can be taken from collected data for any segment of any program's execution. Essentially, average accesses per instruction may be the only measure available, but because it is an average, it may not correspond well with the portion of a benchmark, or a certain benchmark in a suite of benchmarks that is of interest. In this case, getting exact counts may result in a significant increase in the accuracy of calculated values.

b. misses/instructions committed = miss rate * (memory accesses/instructions committed) = miss rate * (memory accesses/instructions fetched) * (instructions fetched/instructions committed).

c. The measurement "memory accesses per instruction fetched" is an average over all fetched instructions. It would be more accurate, as mentioned in the answer to part (a), to measure the exact access and fetch counts. Suppose we are interested in the portion alpha of a benchmark.

$$\text{Misses/instructions committed} = \text{miss rate} * \left(\text{memory accesses}_{\text{alpha}}/\text{instruction committed}_{\text{alpha}}\right)$$

B.7 The merging write buffer links the CPU to the write-back L2 cache. Two CPU writes cannot merge if they are to different sets in L2. Therefore, for each new entry into the buffer, a quick check on only those address bits that determine the L2 set number need be performed at first. If there is no match in this "screening" test, then the new entry is not merged. If there is a set number match, then all address bits can be checked for a definitive result.

As the associativity of L2 increases, the rate of false positive matches from the simplified check will increase, reducing performance.

B.8 a. Assume the number of cycles to execute the loop with all hits is c. Assuming the misses are not overlapped in memory, then their effects will accumulate. So, the iteration would take

$$t = c + 4 \times 100\,\text{cycles}$$

b. If the cache line size, then every fourth iteration will miss elements of *a*, *b*, *c*, and *d*. The rest of iterations will find the data in the cache. So, on the average, an iteration will cost

$$t = (c + 4 \times 100 + c + c + c)/4 = c + 100\,\text{cycles}$$

c. Similar to the answer in part (b), every 16th iteration will miss elements of *a*, *b*, *c*, and *d*.

$$t = (c + 4 \times 100 + 15 \times c)/16 = c + 25\,\text{cycles}$$

d. If the cache is direct-mapped and is of same size as the arrays *a*, *b*, *c*, and *d*, then the layout of the arrays will cause every array access to be a miss! That is because a_i, b_i, c_i, and d_i will map to the same cache line. Hence, every iteration will have 4 misses (3 read misses and a write miss). In addition, there is cost of a write-back for d_i, which will take place in iterations 1 through 511.
Therefore, the average number of cycles is

$$t = c + 400 + 511/512 \times 100$$

B.9 Construct a trace of the form addr1, addr2, addr3, addr1, addr2, addr3, addr1, addr2, addr3, ……, such that all the three addresses map to the same set in the two-way associative cache. Because of the LRU policy, every access will evict a block and the miss rate will be 100%.

If the addresses are set such that in the direct mapped cache addr1 maps to one block while add2 and addr3 map to another block, then all addr1 accesses will be hits, while all addr2/addr3 accesses will be all misses, yielding a 66% miss rate.

Example for 32-word cache: consider the trace 0, 16, 48, 0, 16, 48, …

When the cache is direct mapped address 0 will hit in set 0, while addresses 16 and 48 will keep bumping each other off set 16.

On the other hand, if the 32 word cache is organized as 16 set of two ways each. All three addresses (0, 16, 48) will map to set 0. Because of LRU, that stream will produce a 100% miss rate!

That behavior can happen in real code except that the miss rates would not be that high because of all the other hits to the other blocks of the cache.

B.10 a. L1 cache miss behavior when the caches are organized in an inclusive hierarchy and the two caches have identical block size:

- Access L2 cache.
- If L2 cache hits, supply block to L1 from L2, evicted L1 block can be stored in L2 if it is not already there.
- If L2 cache misses, supply block to both L1 and L2 from memory, evicted L1 block can be stored in L2 if it is not already there.
- In both cases (hit, miss), if storing an L1 evicted block in L2 causes a block to be evicted from L2, then L1 has to be checked and if L2 block that was evicted is found in L1, it has to be invalidated.

b. L1 cache miss behavior when the caches are organized in an exclusive hierarchy and the two caches have identical block size:

- Access L2 cache
- If L2 cache hits, supply block to L1 from L2, invalidate block in L2, write evicted block from L1 to L2 (it must have not been there)
- If L2 cache misses, supply block to L1 from memory, write evicted block from L1 to L2 (it must have not been there)

c. When L1 evicted block is dirty, it must be written back to L2 even if an earlier copy was there (inclusive L2). No change for exclusive case.

B.11 a. Allocating space in the cache for an instruction that is used infrequently or just once means that the time taken to bring its block into the cache is invested to little benefit or no benefit, respectively. If the replaced block is heavily used, then in addition to allocation cost there will be a miss penalty that would not have been incurred if there were a no-allocate decision.

Code example

```
Begin Loop1
        C1
        Begin Loop2
                C2
        End Loop2
End Loop1
```

In the simple example above, it is better not to have code C1 enter the cache as it might eventually conflict with C2 which is executed more often.

b. A software technique to enforce exclusion of certain code blocks from the instruction cache is to place the code blocks in memory areas with noncacheable attributes. That might force some reordering of the code. In the example above that can be achieved by placing C1 in a noncacheable area and jumping to it. A less common approach would be to have instructions turning caching on and off surrounding a piece of code.

B.12 a. t_s = average access time for smaller cache = 0.22 + m1 × 100 ns
t_l = average access time for larger cache = 0.52 + m2 × 100 ns
$t_s < t_l \rightarrow 0.22 + \text{m1} \times 100 < 0.52 + \text{m2} \times 100 \rightarrow (\text{m1} - \text{m2}) \times 100 < 0.32$

b. $t_s < t_l \rightarrow (\text{m1} - \text{m2}) \times 10 < 0.32$, and $t_s < t_l \rightarrow (\text{m1} - \text{m2}) \times 1000 < 0.32$

The inequalities show that a smaller cache might be more advantageous when the ratio of hit time advantage divided by miss penalty is smaller.

B.13

VP#	PP#	Entry valid
5	30	1
7	1	0
10	10	1
15	25	1

Virtual page index	Physical page #	Present
0	3	Y
1	7	N
2	6	N
3	5	Y
4	14	Y
5	30	Y
6	26	Y
7	11	Y
8	13	N
9	18	N
10	10	Y
11	56	Y
12	110	Y
13	33	Y
14	12	N
15	25	Y

Virtual page accessed	TLB (hit or miss)	Page table (hit or fault)
1	miss	fault
5	hit	hit
9	miss	fault
14	miss	fault
10	hit	hit
6	miss	hit
15	hit	hit
12	miss	hit
7	hit	hit
2	miss	fault

B.14 a. We can expect software to be slower due to the overhead of a context switch to the handler code, but the sophistication of the replacement algorithm can be higher for software and a wider variety of virtual memory organizations can be readily accommodated. Hardware should be faster, but less flexible.

b. Factors other than whether miss handling is done in software or hardware can quickly dominate handling time. Is the page table itself paged? Can software implement a more efficient page-table search algorithm than hardware? What about hardware TLB entry prefetching?

c. Page table structures that change dynamically would be difficult to handle in hardware but possible in software.

d. Floating-point programs often traverse large data structures and thus more often reference a large number of pages. It is thus more likely that the TLB will experience a higher rate of capacity misses.

B.15 Solutions vary

Appendix C Solutions

C.1 a.

x1	ld	daddi
x1	daddi	sd
x2	ld	daddi
x2	sd	daddi
x2	dsub	daddi
x4	bnez	dsub

b. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`ld    x1, 0(x2)`	F	D	X	M	W													
`daddi x1, x1, 1`		F	s	s	D	X	M	W										
`sd    x1, 0(x2)`					F	s	s	D	X	M	W							
`daddi x2, x2, 4`								F	D	X	M	W						
`dsub  x4, x3, x2`									F	s	s	D	X	M	W			
`bnez  x4, Loop`												F	s	s	D	X	M	W
`LD R1, 0(R2)`																	F	D

Since the initial value of x3 is x2 + 396 and equal instances of the loop adds 4 to x2, the total number of iterations is 99. Notice that there are eight cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is 98 × 16 + 18 = 1584. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

c. Now we are allowed normal bypassing and forwarding circuitry. Branch outcomes and targets are known now at the end of decode.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`ld    x1, 0(x2)`	F	D	X	M	W													
`daddi x1, x1, #1`		F	D	s	X	M	W											
`sd    x1, 0(x2)`			F	s	D	X	M	W										
`daddi x2, x2, #4`					F	D	X	M	W									
`dsub  x4, x3, x2`						F	D	X	M	W								
`bnez  x4, Loop`							F	s	D	X	M	W						
`(incorrect instruction)`									F	s	s	s	s					
`ld    x1, 0(x2)`										F	D	X	M	W				

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is 9 × 98 + 12 = 894. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

d. See the table below.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`ld`	`x1, 0(x2)`	F	D	X	M	W													
`daddi`	`x1, x1, #1`		F	D	s	X	M	W											
`sd`	`x1, 0(x2)`			F	s	D	X	M	W										
`daddi`	`x2, x2, #4`					F	D	X	M	W									
`dsub`	`x4, x3, x2`						F	D	X	M	W								
`bnez`	`x4, Loop`							F	s	D	X	M	W						
`ld`	`x1, 0(x2)`									F	D	X	M	W					

Again we have 99 iterations. We still experience two RAW stalls, but since we correctly predict the branch, we do not need to flush after the branch. Thus, we have only 8 × 98 + 12 = 796.

e. See the table below.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ld`	`x1, 0(x2)`	F1	F2	D1	D2	X1	X2	M1	M2	W1	W2										
`daddi`	`x1, x1, #1`		F1	F2	D1	D2	s	s	s	X1	X2	M1	M2	W1	W2						
`sd`	`x1, 0(x2)`			F1	F2	D1	s	s	s	D2	X1	X2	M1	M2	W1	W2					
`daddi`	`x2, x2, #4`				F1	F2	s	s	s	D1	D2	X1	X2	M1	M2	W1	W2				
`dsub`	`x4, x3, x2`					F1	s	s	s	F2	D1	D2	s	X1	X2	M1	M2	W1	W2		
`bnez`	`x4, Loop`									F1	F2	D1	s	D2	X1	X2	M1	M2	W1	W2	
`ld`	`x1, 0(x2)`										F1	F2	s	D1	D2	X1	X2	M1	M2	W1	W2

We again have 99 iterations. There are three RAW stalls between the ld and addi, and one RAW stall between the daddi and dsub. Because of the branch prediction, 98 of those iterations overlap significantly. The total number of cycles is 10 × 98 + 19 = 999.

f. 0.9 ns for the 5-stage pipeline and 0.5 ns for the 10-stage pipeline.

g. CPI of 5-stage pipeline: 796/(99 × 6) = 1.34.

CPI of 10-stage pipeline: 999/(99 × 6) = 1.68.
Avg Inst Exe Time 5-stage: 1.34 × 0.9 = 1.21.
Avg Inst Exe Time 10-stage: 1.68 × 0.5 = 0.84

C.2. a. This exercise asks, "How much faster would the machine be … ," which should make you immediately think speedup. In this case, we are interested in how the presence or absence of control hazards changes the pipeline speedup. Recall one of the expressions for the speedup from pipelining presented on page C-13

$$\text{Pipeline speedup} = \frac{1}{1 + \text{Pipeline stalls}} \times \text{Pipeline depth} \qquad \text{(S.1)}$$

where the only contributions to Pipeline stalls arise from control hazards because the exercise is only focused on such hazards. To solve this exercise, we will compute the speedup due to pipelining both with and without control hazards and then compare these two numbers.

For the "ideal" case where there are no control hazards, and thus stalls, Equation (S.1) yields

$$\text{Pipeline speedup}_{\text{ideal}} = \frac{1}{1+0}(4) = 4 \qquad \text{(S.2)}$$

where, from the exercise statement the pipeline depth is 4 and the number of stalls is 0 as there are no control hazards.

For the "real" case where there are control hazards, the pipeline depth is still 4, but the number of stalls is no longer 0. To determine the value of Pipeline stalls, which includes the effects of control hazards, we need three pieces of information. First, we must establish the "types" of control flow instructions we can encounter in a program. From the exercise statement, there are three types of control flow instructions: taken conditional branches, not-taken conditional branches, and jumps and calls. Second, we must evaluate the number of stall cycles caused by each type of control flow instruction. And third, we must find the frequency at which each type of control flow instruction occurs in code. Such values are given in the exercise statement.

To determine the second piece of information, the number of stall cycles created by each of the three types of control flow instructions, we examine how the pipeline behaves under the appropriate conditions. For the purposes of discussion, we will assume the four stages of the pipeline are Instruction Fetch, Instruction Decode, Execute, and Write Back (abbreviated IF, ID, EX, and WB, respectively). A specific structure is not necessary to solve the exercise; this structure was chosen simply to ease the following discussion.

First, let us consider how the pipeline handles a jump or call. Figure S.43 illustrates the behavior of the pipeline during the execution of a jump or call. Because the first pipe stage can always be done independently of whether the control flow instruction goes or not, in cycle 2 the pipeline fetches the instruction following the jump or call (note that this is all we can do—IF must update the PC, and the next sequential address is the only address known at this point; however, this behavior will prove to be beneficial for conditional branches as we will see shortly). By the end of cycle 2, the jump or call resolves

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Jump or call	IF	ID	EX	WB		
i + 1		IF	IF	ID	EX	. . .
i + 2			*stall*	IF	ID	. . .
i + 3				*stall*	IF	. . .

Figure S.43 Effects of a jump of call instruction on the pipeline.

(recall that the exercise specifies that calls and jumps resolve at the end of the second stage), and the pipeline realizes that the fetch it issued in cycle 2 was to the wrong address (remember, the fetch in cycle 2 retrieves the instruction immediately following the control flow instruction rather than the target instruction), so the pipeline reissues the fetch of instruction *i* + 1 in cycle 3. This causes a one-cycle stall in the pipeline since the fetches of instructions after *i* + 1 occur one cycle later than they ideally could have. Figure S.44 illustrates how the pipeline stalls for two cycles when it encounters a taken conditional branch. As was the case for unconditional branches, the fetch issued in cycle 2 fetches the instruction after the branch rather than the instruction at the target of the branch. Therefore, when the branch finally resolves in cycle 3 (recall that the exercise specifies that conditional branches resolve at the end of the third stage), the pipeline realizes it must reissue the fetch for instruction *i* + 1 in cycle 4, which creates the two-cycle penalty.

Figure S.45 illustrates how the pipeline stalls for a single cycle when it encounters a not-taken conditional branch. For not-taken conditional branches, the fetch of instruction *i* + 1 issued in cycle 2 actually obtains the correct instruction. This occurs because the pipeline fetches the next sequential instruction from the program by default—which happens to be the instruction that follows a not-taken branch. Once the conditional branch resolves in cycle 3, the pipeline determines it does not need to reissue the fetch of instruction *i* + 1 and therefore can resume executing the instruction it fetched in cycle 2. Instruction *i* + 1 cannot leave the IF stage until *after* the branch resolves because the exercise specifies the pipeline is only capable of using the IF stage while a branch is being resolved.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Taken branch	IF	ID	EX	WB		
i + 1		IF	*stall*	IF	ID	. . .
i + 2			*stall*	*stall*	IF	. . .
i + 3				*stall*	*stall*	. . .

Figure S.44 Effects of a taken conditional branch on the pipeline.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Not-taken branch	IF	ID	EX	WB		
i + 1		IF	*stall*	ID	EX	. . .
i + 2			*stall*	IF	ID	. . .
i + 3				*stall*	IF	. . .

Figure S.45 Effects of a not-taken conditional branch on the pipeline.

Control flow type	**Frequency (per instruction)**	**Stalls (cycles)**
Jumps and calls	1%	1
Conditional (taken)	15% × 60% = 9%	2
Conditional (not taken)	15% × 40% = 6%	1

Figure S.46 A summary of the behavior of control flow instructions.

Combining all of our information on control flow instruction type, stall cycles, and frequency leads us to Figure S.46. Note that this figure accounts for the taken/not-taken nature of conditional branches. With this information we can compute the stall cycles caused by control flow instructions:

$$\text{Pipeline stalls}_{\text{real}} = (1 \times 1\%) + (2 \times 9\%) + (1 \times 6\%) = 0.24$$

where each term is the product of a frequency and a penalty. We can now plug the appropriate value for Pipeline stalls$_{\text{real}}$ into Equation (S.1) to arrive at the pipeline speedup in the "real" case:

$$\text{Pipeline speedup}_{\text{ideal}} = \frac{1}{1+0.24} = (4.0) = 3.23 \qquad \text{(S.3)}$$

Finding the speedup of the ideal over the real pipelining speedups from Equations (S.2), (S.3) leads us to the final answer:

$$\text{Pipeline speedup}_{\text{without controlhazards}} = \frac{4}{3.23} = 1.24$$

Thus, the presence of control hazards in the pipeline loses approximately 24% of the speedup you achieve without such hazards.

b. We need to perform similar analysis as in Exercise C.2 (a), but now with larger penalties. For a jump or call, the jump or call resolves at the end of cycle 5, leading to 4 wasted fetches, or 4 stalls. For a conditional branch that is taken, the branch is not resolved until the end of cycle 10, leading to 1 wasted fetch and 8 stalls, or 9 stalls. For a conditional branch that is not taken, the branch is still not resolved until the end of cycle 10, but there were only the 8 stalls, not a wasted fetch.

Pipeline $\text{stalls}_{\text{real}} = (4 \times 1\%) + (9 \times 9\%) + (8 \times 6\%) = 0.04 + 0.81 + 0.48 = 1.33$
Pipeline $\text{speedup}_{\text{real}} = (1/(1 + 1.33)) \times (4) = 4/2.33 = 1.72$
Pipeline $\text{speedup}_{\text{without control hazards}} = 4/1.72 = 2.33$
If you compare the answer to (b) with the answer to (a), you can see just how important branch prediction is to the performance of modern high-performance deeply-pipelined processors.

C.3 a. 2 ns + 0.1 ns = 2.1 ns

b. 5 cycles/4 instructions = 1.25

c. Execution Time = $I \times$ CPI $\times$ Cycle Time

Speedup = $(I \times 1 \times 7)/(I \times 1.25 \times 2.1) = 2.67$

d. Ignoring extra stall cycles, it would be: $I \times 1 \times 7/I \times 1 \times 0.1 = 70$

C.4 Calculating the branch in the ID stage does not help if the branch is the one receiving data from the previous instruction. For example, loops which exit depending on a memory value have this property, especially if the memory is being accessed through a linked list rather than an array. There are many correct answers to this.

```
LOOP:  LW   x1, 4(x2)       # x1 = x2->value
       ADD  x3, x3, x1      # sum = sum + x2->value
       LW   x2, 0(x2)       # x2 = x2->next
       BNE  x2, x0, LOOP    # while (x2 != null) keep looping
```

The second LW and ADD could be reordered to reduce stalls, but there would still be a stall between the LW and BNE.

C.5 (No solution provided)

C.6 (No solution provided)

C.7 a. Execution Time = I × CPI × Cycle Time
Speedup = (I × 6/5 × 1)/(I × 11/8 × 0.6) = 1.45

b. $\text{CPI}_{\text{5-stage}} = 6/5 + 0.20 \times 0.05 \times 2 = 1.22$,
$\text{CPI}_{\text{12-stage}} = 11/8 + 0.20 \times 0.05 \times 5 = 1.425$
Speedup = $(1 \times 1.22 \times 1)/(1 \times 1.425 \times 0.6) = 1.17$

C.8–C.11 (No solution provided)

C.12 a. 21 − 5 = 16 cycles/iteration

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`L.D F6, 0(R1)`	IF	ID	EX	MM	WB																
`MUL.D F4, F2, F0`		IF	ID	s	M1	M2	M3	M4	M5	M6	M7	MM	WB								
`LD F6, 0(R2)`			IF	s	ID	EX	MM	WB													
`ADD.D F6, F4, F6`					IF	ID	s	s	s	s	s	A1	A2	A3	A4	MM	WB				
`S.D 0(R2), F6`						IF	s	s	s	s	s	ID	s	s	s	EX	MM	WB			
`DADDIU R1, R1, #`												IF	ID	s	s	s	EX	MM	WB		
`DSGTUI`													IF	s	s	s	ID	EX	MM	WB	
`BEQZ`																	IF	ID	EX	MM	WB

b. (No solution provided)

c. (No solution provided)

C.13 (No solution provided)

C.14 a. There are several correct answers. The key is to have two instructions using different execution units that would want to hit the WB stage at the same time.

b. There are several correct answers. The key is that there are two instructions writing to the same register with an intervening instruction that reads from the register. The first instruction is delayed because it is dependent on a long-latency instruction (i.e., division). The second instruction has no such delays, but it cannot complete because it is not allowed to overwrite that value before the intervening instruction reads from it.

Computer Architecture: A Quantitative Approach 6e

John L. Hennessy and David A. Patterson

Errata list as of 11/28/2017

Chapter		Page #		Description	Correction
2		158		Exercises that follow the case studies are mis-numbered after problem 2.25.	Problems from 2.25 onward should be in sequence, without a gap from problem 2.25 to 2.30 (ie, there are not four problems missing). Note that the solutions file has the correct sequence (ie, problem 2.30 in the text corresponds to solution 2.26 in the solutions file, etc.)
3		269		In Exercise 3.8, Figure 3.48, the instruction on line I5 reads "sd"	Instruction should be "fsd" instead of "sd"
3		271		In Exercises 3.9, code is shown as: addi x1, x1, x1 addi x1, x1, x1 addi x1, x1, x1	Code should be: add x1, x1, x1 add x1, x1, x1 add x1, x1, x1