Solutions to Case Studies and Exercises

Chapter 1 Solutions

Case Study 1: Chip Fabrication Cost

1.1 a. Yield $= \left(1 + \frac{0.30 \times 3.89}{4.0}\right)^{-4} = 0.36$

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. Dies per wafer $= \frac{\pi \times (30/2)^2}{1.5} - \frac{\pi \times 30}{\text{sqrt}(2 \times 1.5)} = 471 - 54.4 = 416$

$$\text{Yield} = \left(1 + \frac{0.30 \times 1.5}{4.0}\right)^{-4} = 0.65$$

$$\text{Profit} = 416 \times 0.65 \times \$20 = \$5408$$

b. Dies per wafer $= \frac{\pi \times (30/2)^2}{2.5} - \frac{\pi \times 30}{\text{sqrt}(2 \times 2.5)} = 283 - 42.1 = 240$

$$\text{Yield} = \left(1 + \frac{0.30 \times 2.5}{4.0}\right)^{-4} = 0.50$$

$$\text{Profit} = 240 \times 0.50 \times \$25 = \$3000$$

c. The Woods chip

d. Woods chips: 50,000/416 = 120.2 wafers needed
Markon chips: 25,000/240 = 104.2 wafers needed

Therefore, the most lucrative split is 120 Woods wafers, 30 Markon wafers.

1.3 a. Defect – Free single core $= \left(1 + \frac{0.75 \times 1.99/2}{4.0}\right)^{-4} = 0.28$

No defects = $0.28^2 = 0.08$
One defect = $0.28 \times 0.72 \times 2 = 0.40$
No more than one defect = $0.08 + 0.40 = 0.48$

b. $\$20 = \frac{\text{Wafer size}}{\text{old dpw} \times 0.28}$

$\$20 \times 0.28$ = Wafer size/old dpw

$$x = \frac{\text{Wafer size}}{1/2 \times \text{old dpw} \times 0.48} = \frac{\$20 \times 0.28}{1/2 \times 0.48} = \$23.33$$

Case Study 2: Power Consumption in Computer Systems

1.4 a. $.80x = 66 + 2 \times 2.3 + 7.9$; $x = 99$

b. $.6 \times 4\ \text{W} + .4 \times 7.9 = 5.56$

c. Solve the following four equations:

seek7200 = .75 × seek5400
seek7200 + idle7200 = 100
seek5400 + idle5400 = 100
seek7200 × 7.9 + idle7200 × 4 = seek5400 × 7 + idle5400 × 2.9

idle7200 = 29.8%

1.5 a. $$\frac{14\ \text{KW}}{(66\ \text{W} + 2.3\ \text{W} + 7.9\ \text{W})} = 183$$

b. $$\frac{14\ \text{KW}}{(66\ \text{W} + 2.3\ \text{W} + 2 \times 7.9\ \text{W})} = 166$$

c. 200 W × 11 = 2200 W
2200/(76.2) = 28 racks
Only 1 cooling door is required.

1.6 a. The IBM x346 could take less space, which would save money in real estate. The racks might be better laid out. It could also be much cheaper. In addition, if we were running applications that did not match the characteristics of these benchmarks, the IBM x346 might be faster. Finally, there are no reliability numbers shown. Although we do not know that the IBM x346 is better in any of these areas, we do not know it is worse, either.

1.7 a. $(1 - 8) + .8/2 = .2 + .4 = .6$

b. $$\frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.60)^2 \times (F \times 0.60)}{V^2 \times F} = 0.6^3 = 0.216$$

c. $$1 = \frac{.75}{(1 - x) + x/2}\ ; x = 50\%$$

d. $$\frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.75)^2 \times (F \times 0.60)}{V^2 \times F} = 0.75^2 \times 0.6 = 0.338$$

Exercises

1.8 a. $(1.35)^{10}$ = approximately 20

b. $3200 \times (1.4)^{12}$ = approximately 181,420

c. $3200 \times (1.01)^{12}$ = approximately 3605

d. Power density, which is the power consumed over the increasingly small area, has created too much heat for heat sinks to dissipate. This has limited the activity of the transistors on the chip. Instead of increasing the clock rate, manufacturers are placing multiple cores on the chip.

e. Anything in the 15–25% range would be a reasonable conclusion based on the decline in the rate over history. As the sudden stop in clock rate shows, though, even the declines do not always follow predictions.

1.9 a. 50%

b. Energy = ½ load × V^2. Changing the frequency does not affect energy–only power. So the new energy is ½ load × (½ V)^2, reducing it to about ¼ the old energy.

1.10 a. 60%

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

c. newPower/oldPower = ½ Capacitance × (Voltage × .8)^2 × (Frequency × .6)/½ Capacitance × Voltage × Frequency = 0.8^2 × 0.6 = 0.256 of the original power.

d. 0.4 + 0 .3 × 2 = 0.46, which reduce the energy to 46% of the original energy.

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

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

c. [need solution]

1.12 a. 35/10000 × 3333 = 11.67 days

b. There are several correct answers. One would be that, with the current system, one computer fails approximately every 5 minutes. 5 minutes is unlikely to be enough time to isolate the computer, swap it out, and get the computer back on line again. 10 minutes, 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/hr

1.13 a. Itanium, because it has a lower overall execution time.

b. Opteron: 0.6 × 0.92 + 0.2 × 1.03 + 0.2 × 0.65 = 0.888

c. 1/0.888 = 1.126

1.14 a. See Figure S.1.

b. 2 = 1/((1 – x) + x/10)
5/9 = x = 0.56 or 56%

c. 0.056/0.5 = 0.11 or 11%

d. Maximum speedup = 1/(1/10) = 10
5 = 1/((1 – x) + x/10)
8/9 = x = 0.89 or 89%

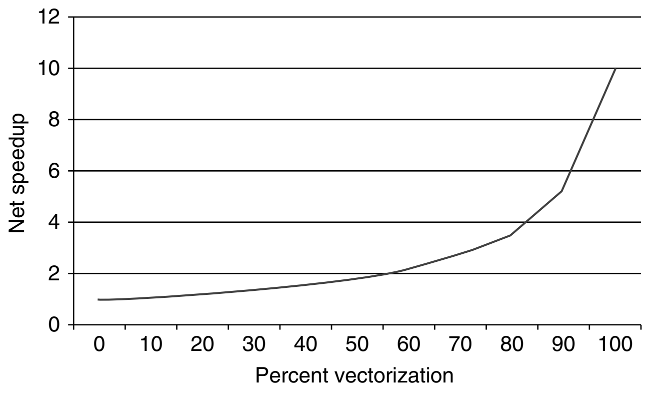


Figure S.1 Plot of the equation: y = 100/((100 – x) + x/10).

e. Current speedup: 1/(0.3 + 0.7/10) = 1/0.37 = 2.7
Speedup goal: 5.4 = 1/((1 – x) + x /10) = x = 0.91

This means the percentage of vectorization would need to be 91%

1.15 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

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

b. 1/(0.7 + 0.20/2 + 0.10 × 3/2) = 1.05

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

1.17 a. 1/(0.6 + 0.4/2) = 1.25

b. 1/(0.01 + 0.99/2) = 1.98

c. 1/(0.2 + 0.8 × 0.6 + 0.8 × 0.4/2) = 1/(.2 + .48 + .16) = 1.19

d. 1/(0.8 + 0.2 × .01 + 0.2 × 0.99/2) = 1/(0.8 + 0.002 + 0.099) = 1.11

1.18 a. 1/(.2 + .8/N)

b. 1/(.2 + 8 × 0.005 + 0.8/8) = 2.94

c. 1/(.2 + 3 × 0.005 + 0.8/8) = 3.17

d. 1/(.2 + logN × 0.005 + 0.8/N)

e. d/dN(1/((1 – P) + logN × 0.005 + P/N)) = 0

Chapter 2 Solutions

Case Study 1: Optimizing Cache Performance via Advanced Techniques

2.1 a. Each element is 8B. Since a 64B cacheline has 8 elements, and each column access will result in fetching a new line for the non-ideal matrix, we need a minimum of 8x8 (64 elements) for each matrix. Hence, the minimum cache size is $128 \times 8B = 1KB$.

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 64Bx256 of storage, or 16KB. 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 Since 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. Since 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 1MB and has a 128B block size.

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

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

d. The main memory is 512MB.

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

2.5 a. Hint: This is visible in the graph above as a slight increase in L2 miss service time for large data sets, and is 4KB 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 as a slight increase in L2 miss service time for large data sets, and is 15ns 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.

Exercises

2.8 a. The access time of the direct-mapped cache is 0.86ns, while the 2-way and 4-way are 1.12ns and 1.37ns respectively. This makes the relative access times 1.12/.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 16KB cache is 1.27ns, while the 32KB and 64KB are 1.35ns and 1.37ns respectively. This makes the relative access times 1.35/1.27 = 1.06 or 6% larger for the 32KB and 1.37/1.27 = 1.078 or 8% larger for the 64KB.

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), .09% (8-way).

Direct mapped access time = .86ns @ .5ns cycle time = 2 cycles
2-way set associative = 1.12ns @ .5ns cycle time = 3 cycles

4-way set associative = 1.37ns @ .83ns cycle time = 2 cycles
8-way set associative = 2.03ns @ .79ns cycle time = 3 cycles
Miss penalty = (10/.5) = 20 cycles for DM and 2-way; 10/.83 = 13 cycles for 4-way; 10/.79 = 13 cycles for 8-way.

Direct mapped – $(1 - .022) \times 2 + .022 \times (20) = 2.39\ 6$ cycles => $2.396 \times .5 = 1.2$ns
2-way – $(1 - .012) \times 3 + .012 \times (20) = 3.\ 2$ cycles => $3.2 \times .5 = 1.6$ns
4-way – $(1 - .0033) \times 2 + .0033 \times (13) = 2.036$ cycles => $2.06 \times .83 = 1.69$ns
8-way – $(1 - .0009) \times 3 + .0009 \times 13 = 3$ cycles => $3 \times .79 = 2.37$ns

Direct mapped cache is the best.

2.9 a. The average memory access time of the current (4-way 64KB) cache is 1.69ns.
64KB direct mapped cache access time = .86ns @ .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 = 1.13ns

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

c. With 1 cycle way misprediction penalty, AMAT is 1.13ns (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 2.3ns.

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

2.10 a. The access time is 1.12ns, while the cycle time is 0.51ns, which could be potentially pipelined as finely as 1.12/.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 0.16nJ. The banked cache has an area of 1.36 mm^2 and energy per access of 0.13nJ. 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.

2.11 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.

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

b. Assume merging write buffer entries are 16B wide. Since 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 non-merging write buffer would take 4 cycles to write the 8B result of each store. This means the merging write buffer would be 2 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 non-blocking caches, writes can be processed from the write buffer during misses, which may mean fewer entries are needed.

2.13 a. A 2GB DRAM with parity or ECC effectively has 9 bit bytes, and would require 18 1Gb 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 × 8 = 5336 MB/s.

The DDR-533 DIMM bandwidth is 533 × 8 = 4264 MB/s.

2.14 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 × 2 = 18 cycles of a 333MHz clock, or 18 × (1/333MHz) = 54.0ns.

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

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

It is 12 × (1/266MHz) = 45ns (80% of the time) and 8 × (1/266MHz) = 30ns (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 × 42 × .8 + 0.00333 × 27 × .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 × 45 × .8 + 0.00333 × 30 × .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 × 1060 = 1728 while the cost/performance of the DDR2-533 system is 1.64 × 1000 = 1640, so the DDR2-533 system is a better value.

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

2.17 a. The system built from 1Gb DRAMs will have twice as many banks as the system built from 2Gb DRAMs. Thus the 1Gb-based system should provide higher performance since 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 since there are fewer DRAMs being activated in the x8 design option, it would have lower power.

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

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

Access time when there is a miss

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

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.

i.e.,

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

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,

$$\frac{r}{100}(Tcas + Tddr)A + \frac{100 - r}{100}(Trp + Trcd + Tcas + Tddr)A$$
$$= 0.9 \times (Trcd + Tcas + Tddr)A + 0.1 \times (Trcd + Tcas + Trp + Tddr)$$
$$\Rightarrow r = 90 \times \left(\frac{Trp}{Trp + Trcd}\right)$$
$$r = 90 \times 15/30 = 45\%$$

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

2.19 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.

$$Time = \frac{8 \times 10^9 \times 2 \times 2.56 \times 10^{-6}}{64 \times 1.6}$$

$$= 400 \text{ seconds}$$

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

2.20 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.

2.21 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 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.

2.22 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.

2.23 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.

2.24 Open hands-on exercise, no fixed solution.

2.25 a. These results are from experiments on a 3.3GHz Intel® Xeon® Processor X5680 with Nehalem architecture (westmere at 32nm). The number of misses per 1K instructions of L1 Dcache increases significantly by more than 300X when input data size goes from 8KB to 64 KB, and keeps relatively constant around 300/1K 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 8KB to 128 KB, from 128KB to 4MB, and from 4MB to 32MB, 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 256KB 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 8KB to 128KB, the number of L1 Dcache misses per 1K instructions increases by around 300, and the number of L2 cache misses per 1K 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 1K instructions due to L1 misses, which means an increase of 3.3 cycles per instruction on average. The measured CPI with the 8KB input data size is 1.37. Without any latency tolerance mechanisms we would expect the CPI of the 128KB case to be $1.37 + 3.3 = 4.67$. However, the measured CPI of the 128KB case is 3.44. This means that memory latency hiding techniques such as OOO execution, prefetching, and non-blocking 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 3.48, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 40. See Figure S.2. 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.

```
Loop:    LD       F2,0(Rx)      1 + 4
         DIVD     F8,F2,F0      1 + 12
         MULTD    F2,F6,F2      1 + 5
         LD       F4,0(Ry)      1 + 4
         ADDD     F4,F0,F4      1 + 1
         ADDD     F10,F8,F2     1 + 1
         ADDI     Rx,Rx,#8      1
         ADDI     Ry,Ry,#8      1
         SD       F4,0(Ry)      1 + 1
         SUB      R20,R4,Rx     1
         BNZ      R20,Loop      1 + 1
                                ____
         cycles per loop iter   40
```

Figure S.2 Baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48.

3.2 How many cycles would the loop body in the code sequence in Figure 3.48 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.3. 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 `LD` must stall the next instruction for three clock cycles. The `MULTD` produces a result for its successor, and therefore must stall 4 more clocks, and so on.

```
Loop:   LD          F2,0(Rx)                  1 + 4
        <stall>
        <stall>
        <stall>
        <stall>
        DIVD        F8,F2,F0                  1 + 12
        MULTD       F2,F6,F2                  1 + 5
        LD          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>
        ADDD        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>
        ADDD        F10,F8,F2                 1 + 1
        ADDI        Rx,Rx,#8                  1
        ADDI        Ry,Ry,#8                  1
        SD          F4,0(Ry)                  1 + 1
        SUB         R20,R4,Rx                 1
        BNZ         R20,Loop                  1 + 1
        <stall branch delay slot>
                                              ------
        cycles per loop iter                  25
```

Figure S.3 Number of cycles required by the loop body in the code sequence in Figure 3.48.

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, as shown in Figure S.4. The LD goes first, as before, and the DIVD must wait for it through 4 extra latency cycles. After the DIVD comes the MULTD, which can run in the second pipe along with the DIVD, 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 LD following the MULTD does not depend on the DIVD nor the MULTD, so had this been a superscalar-order-3 machine,

	Execution pipe 0		Execution pipe 1
Loop:	**LD F2,0(Rx)**	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	DIVD F8,F2,F0	;	**MULTD F2,F6,F2**
	LD F4,0(Ry)	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	ADD F4,F0,F4	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	ADDD F10,F8,F2	;	**ADDI Rx,Rx,#8**
	ADDI Ry,Ry,#8	;	**SD F4,0(Ry)**
	SUB R20,R4,Rx	;	**BNZ R20,Loop**
	<nop>	;	**<stall due to BNZ>**
	cycles per loop iter 22		

Figure S.4 Number of cycles required per loop.

that LD could conceivably have been executed concurrently with the DIVD and the MULTD. Since this problem posited a two-execution-pipe machine, the LD executes in the cycle following the DIVD/MULTD. 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.
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 `DIVD` instr will complete long after the `LD F4,0(Ry)`, for example.

3.5 Figure S.5 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 3.48. The number of cycles that this reordered code takes is 20.

```
       Execution pipe 0                     Execution pipe 1
Loop:  LD      F2,0(Rx)                 ;   LD      F4,0(Ry)
       <stall for LD latency>           ;   <stall for LD latency>
       <stall for LD latency>           ;   <stall for LD latency>
       <stall for LD latency>           ;   <stall for LD latency>
       <stall for LD latency>           ;   <stall for LD latency>
       DIVD    F8,F2,F0                 ;   ADDD    F4,F0,F4
       MULTD   F2,F6,F2                 ;   <stall due to ADDD latency>
       <stall due to DIVD latency>      ;   SD      F4,0(Ry)
       <stall due to DIVD latency>      ;   <nop>                        #ops:   11
       <stall due to DIVD latency>      ;   <nop>                        #nops:  (20 × 2) – 11 = 29
       <stall due to DIVD latency>      ;   ADDI    Rx,Rx,#8
       <stall due to DIVD latency>      ;   ADDI    Ry,Ry,#8
       <stall due to DIVD latency>      ;   <nop>
       <stall due to DIVD latency>      ;   <nop>
       <stall due to DIVD latency>      ;   <nop>
       <stall due to DIVD latency>      ;   <nop>
       <stall due to DIVD latency>      ;   <nop>
       <stall due to DIVD latency>      ;   SUB     R20,R4,Rx
       ADDD    F10,F8,F2                ;   BNZ     R20,Loop
       <nop>                            ;   <stall due to BNZ>

       cycles per loop iter 20
```

Figure S.5 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.5:

11 ops out of 2x20 opportunities.

$1 - 11/40 = 1 - 0.275$
$= 0.725$

b. Results of hand-unrolling two iterations of the loop from code shown in Figure S.6:

c. $\text{Speedup} = \dfrac{\text{exec time w/o enhancement}}{\text{exec time with enhancement}}$

$\text{Speedup} = 20 / (22/2)$
$= 1.82$

	Execution pipe 0		Execution pipe 1
Loop:	**LD F2,0(Rx)**	;	**LD F4,0(Ry)**
	LD F2,0(Rx)	;	LD F4,0(Ry)
	<stall for LD latency>	;	<stall for LD latency>
	<stall for LD latency>	;	<stall for LD latency>
	<stall for LD latency>	;	<stall for LD latency>
	DIVD F8,F2,F0	;	**ADDD F4,F0,F4**
	DIVD F8,F2,F0	;	ADDD F4,F0,F4
	MULTD F2,F0,F2	;	**SD F4,0(Ry)**
	MULTD F2,F6,F2	;	SD F4,0(Ry)
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	**ADDI Rx,Rx,#16**
	<stall due to DIVD latency>	;	**ADDI Ry,Ry,#16**
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	ADDD F10,F8,F2	;	**SUB R20,R4,Rx**
	ADDD F10,F8,F2	;	BNZ R20,Loop
	<nop>	;	**<stall due to BNZ>**
	cycles per loop iter 22		

Figure S.6 Hand-unrolling two iterations of the loop from code shown in Figure S.5.

3.7 Consider the code sequence in Figure 3.49. 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 3.50.)

Loop:	LD	T9,0(Rx)
I0:	MULTD	T10,F0,T2
I1:	DIVD	T11,T9,T10
I2:	LD	T12,0(Ry)
I3:	ADDD	T13,F0,T12
I4:	SUBD	T14,T11,T13
I5:	SD	T14,0(Ry)

Figure S.7 Register renaming.

3.8 See Figure S.8. 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 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.

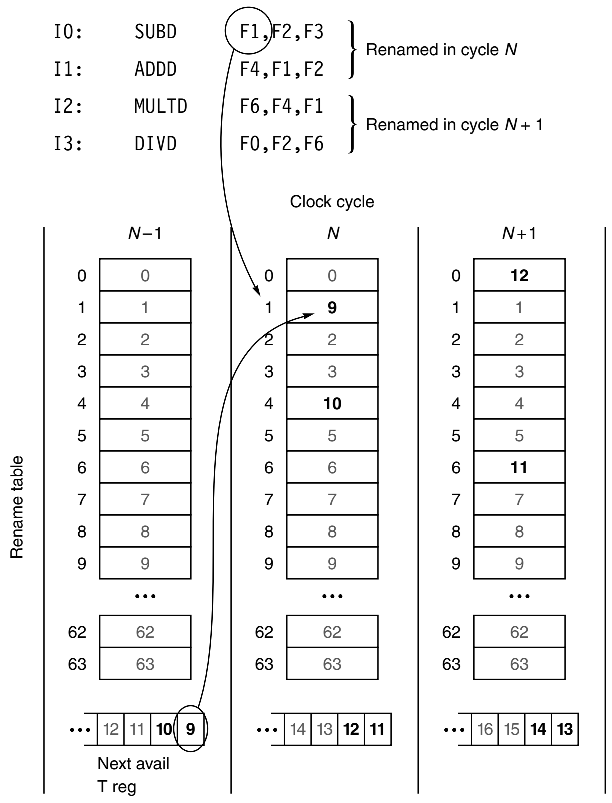


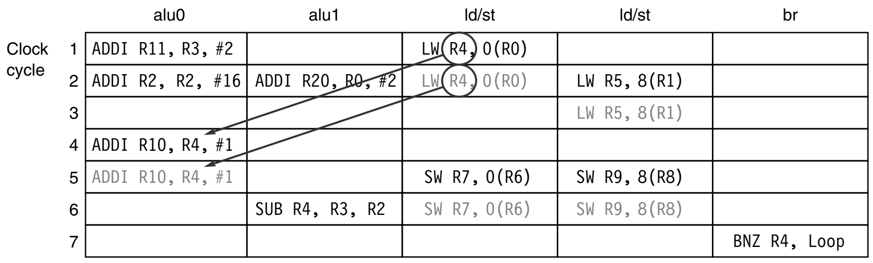
Figure S.8 Cycle-by-cycle state of the rename table for every instruction of the code in Figure 3.51.

3.9 See Figure S.9.

ADD	R1, R1, R1;	5 + 5 –> 10
ADD	R1, R1, R1;	10 + 10 –> 20
ADD	R1, R1, R1;	20 + 20 –> 40

Figure S.9 Value of R1 when the sequence has been executed.

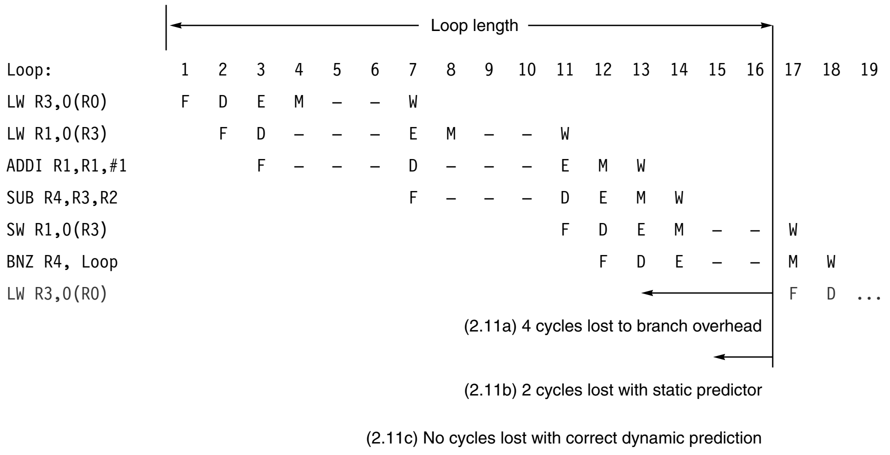
3.10 An example of an event that, in the presence of self-draining pipelines, could disrupt the pipelining and yield wrong results is shown in Figure S.10.



Clock cycle	alu0	alu1	ld/st	ld/st	br
1	ADDI R11, R3, #2		LW R4, 0(R0)		
2	ADDI R2, R2, #16	ADDI R20, R0, #2	LW R4, 0(R0)	LW R5, 8(R1)	
3				LW R5, 8(R1)	
4	ADDI R10, R4, #1				
5	ADDI R10, R4, #1		SW R7, 0(R6)	SW R9, 8(R8)	
6		SUB R4, R3, R2	SW R7, 0(R6)	SW R9, 8(R8)	
7					BNZ R4, Loop

Figure S.10 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.

3.11 See Figure S.11. The convention is that an instruction does not enter the execution phase until all of its operands are ready. So the first instruction, LW R3,0(R0), 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 R1, R1, #1, which depends on the 2nd LW) cannot enter the E stage, and must therefore stall at the D stage.



```
                   |<---------------- Loop length ---------------->|
Loop:              1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 | 17 18 19
LW R3,0(R0)        F  D  E  M  –  –  W
LW R1,0(R3)           F  D  –  –  –  E  M  –  –  W
ADDI R1,R1,#1            F  –  –  –  D  –  –  –  E  M  W
SUB R4,R3,R2                         F  –  –  –  D  E  M  W
SW R1,0(R3)                                      F  D  E  M  –  –  W
BNZ R4, Loop                                        F  D  E  –  –  M  W
LW R3,0(R0)                                          <-----------  F  D ...
                          (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
```

Figure S.11 Phases of each instruction per clock cycle for one iteration of the loop.

a. 4 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 LW R1 can't begin until the branch has completed.

b. 2 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.

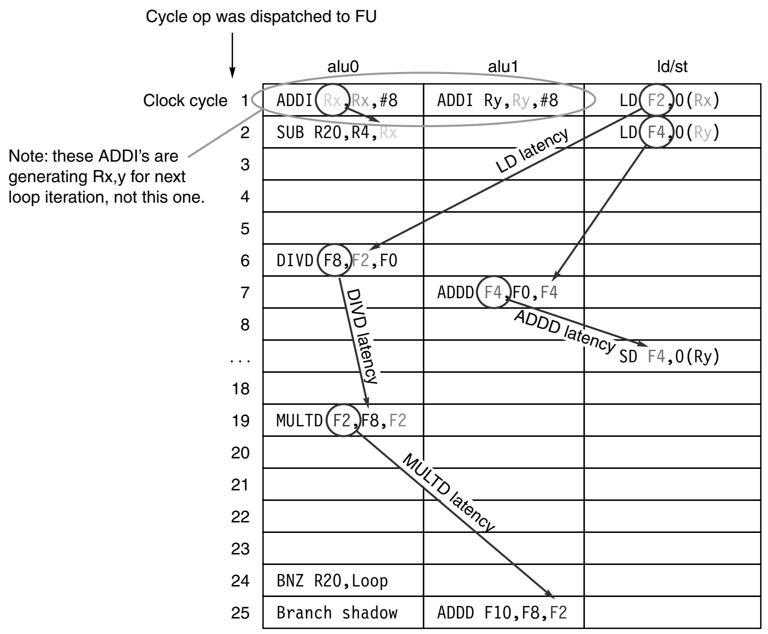
c. No cycles lost w/ correct dynamic prediction. A dynamic branch predictor remembers that when the branch instruction was fetched in the past, it eventually turned out to be a branch, and this branch was taken. So a "predicted taken" will occur in the same cycle as the branch is fetched, and the next 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.12.

```
LD          F2,0(Rx)
DIVD        F8,F2,F0
MULTD       F2,F8,F2     ; reg renaming doesn't really help here, due to
                         ; true data dependencies on F8 and F2
LD          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
ADDD        F4,F0,F4
ADDD        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
SD          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         R20,R4,Rx
BNZ         R20,Loop
```

Figure S.12 Instructions in code where register renaming improves performance.

b. See Figure S.13. The number of clock cycles taken by the code sequence is 25.


Cycle op was dispatched to FU
Clock cycle
Note: these ADDI's are generating Rx,y for next loop iteration, not this one.
LD latency
DIVD latency
ADDD latency
MULTD latency

Clock cycle	alu0	alu1	ld/st
1	ADDI Rx,Rx,#8	ADDI Ry,Ry,#8	LD F2,0(Rx)
2	SUB R20,R4,Rx		LD F4,0(Ry)
3			
4			
5			
6	DIVD F8,F2,F0		
7		ADDD F4,F0,F4	
8			
...			SD F4,0(Ry)
18			
19	MULTD F2,F8,F2		
20			
21			
22			
23			
24	BNZ R20,Loop		
25	Branch shadow	ADDD F10,F8,F2	

Figure S.13 Number of clock cycles taken by the code sequence.

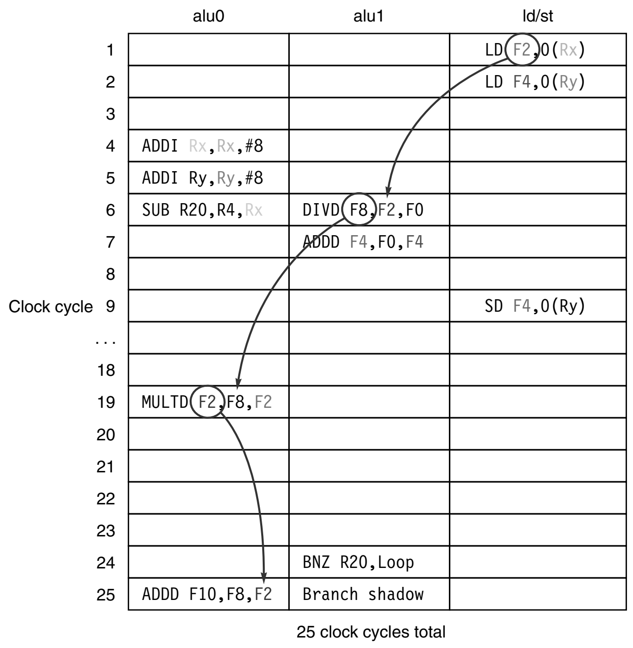
c. See Figures S.14 and S.15. The bold instructions are those instructions that are present in the RS, and ready for dispatch. Think of this exercise from the Reservation Station's point of view: at any given clock cycle, it can only "see" the instructions that were previously written into it, that have 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.

0	1		2		3		4		5		6	
	LD	**F2, 0(Rx)**	LD	F2, 0(Rx)	LD	F2, 0(Rx)	LD	F2, 0(Rx)	LD	F2, 0(Rx)	LD	F2, 0(Rx)
	DIVD	**F8,F2,F0**	**DIVD**	**F8,F2,F0**	**DIVD**	**F8,F2,F0**	**DIVD**	**F8,F2,F0**	**DIVD**	**F8,F2,F0**	**DIVD**	**F8,F2,F0**
	MULTD	F2,F8,F2	**MULTD**	**F2,F8,F2**	**MULTD**	**F2,F8,F2**	**MULTD**	**F2,F8,F2**	**MULTD**	**F2,F8,F2**	**MULTD**	**F2,F8,F2**
	LD	F4, 0(Ry)	**LD**	**F4, 0(Ry)**	LD	F4, 0(Ry)	LD	F4, 0(Ry)	LD	F4, 0(Ry)	LD	F4, 0(Ry)
	ADDD	F4,F0,F4	ADDD	F4,F0,F4	**ADDD**	**F4,F0,F4**	**ADDD**	**F4,F0,F4**	**ADDD**	**F4,F0,F4**	**ADDD**	**F4,F0,F4**
	ADDD	F10,F8,F2	ADDD	F10,F8,F2	**ADDD**	**F10,F8,F2**	**ADDD**	**F10,F8,F2**	**ADDD**	**F10,F8,F2**	**ADDD**	**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
	SD	F4,0(Ry)	SD	F4,0(Ry)	SD	F4,0(Ry)	SD	F4,0(Ry)	**SD**	**F4,0(Ry)**	**SD**	**F4,0(Ry)**
	SUB	R20,R4,Rx	SUB	R20,R4,Rx	SUB	R20,R4,Rx	SUB	R20,R4,Rx	**SUB**	**R20,R4,Rx**	**SUB**	**R20,R4,Rx**
	BNZ	20,Loop	BNZ	20,Loop	BNZ	20,Loop	BNZ	20,Loop	BNZ	20,Loop	**BNZ**	**20,Loop**

First 2 instructions appear in RS

Candidates for dispatch in bold

Figure S.14 Candidates for dispatch.

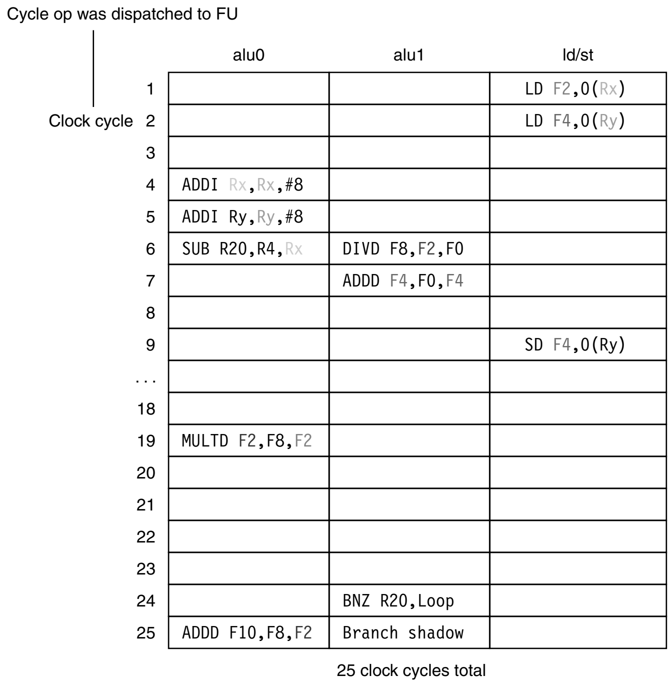


Clock cycle	alu0	alu1	ld/st
1			LD F2,0(Rx)
2			LD F4,0(Ry)
3			
4	ADDI Rx,Rx,#8		
5	ADDI Ry,Ry,#8		
6	SUB R20,R4,Rx	DIVD F8,F2,F0	
7		ADDD F4,F0,F4	
8			
9			SD F4,0(Ry)
. . .			
18			
19	MULTD F2,F8,F2		
20			
21			
22			
23			
24		BNZ R20,Loop	
25	ADDD F10,F8,F2	Branch shadow	

25 clock cycles total

Figure S.15 Number of clock cycles required.

d. See Figure S.16.



Cycle op was dispatched to FU

Clock cycle	alu0	alu1	ld/st
1			LD F2,0(Rx)
2			LD F4,0(Ry)
3			
4	ADDI Rx,Rx,#8		
5	ADDI Ry,Ry,#8		
6	SUB R20,R4,Rx	DIVD F8,F2,F0	
7		ADDD F4,F0,F4	
8			
9			SD F4,0(Ry)
. . .			
18			
19	MULTD F2,F8,F2		
20			
21			
22			
23			
24		BNZ R20,Loop	
25	ADDD F10,F8,F2	Branch shadow	

25 clock cycles total

Figure S.16 Speedup is (execution time without enhancement)/(execution time with enhancement) = 25/(25 – 6) = 1.316.

1. Another ALU: 0% improvement
2. Another LD/ST unit: 0% improvement
3. Full bypassing: critical path is `LD -> Div -> MULT -> ADDD`. 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.17.

Cycle op was dispatched to FU

Clock cycle	alu0	alu1	ld/st
1			LD F2,0(Rx)
2			LD F2,0(Rx)
3			LD F4,0(Ry)
4	ADDI Rx,Rx,#8		
5	ADDI Ry,Ry,#8		
6	SUB R20,R4,Rx	DIVD F8,F2,F0	
7		DIVD F8,F2,F0	
8		ADDD F4,F0,F4	
9			
. . .			SD F4,0(Ry)
18			
19	MULTD F2,F8,F2		
20	MULTD F2,F8,F2		
21			
22			
23			
24			
25	ADDD F10,F8,F2	BNZ R20,Loop	
26	ADDD F10,F8,F2	Branch shadow	

26 clock cycles total

Figure S.17 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.

Exercises

3.13 a. See Figure S.18.

Clock cycle	Unscheduled code		Scheduled code	
1	`DADDIU`	`R4,R1,#800`	`DADDIU`	`R4,R1,#800`
2	`L.D`	`F2,0(R1)`	`L.D`	`F2,0(R1)`
3	`stall`		`L.D`	`F6,0(R2)`
4	`MUL.D`	`F4,F2,F0`	`MUL.D`	`F4,F2,F0`
5	`L.D`	`F6,0(R2)`	`DADDIU`	`R1,R1,#8`
6	`stall`		`DADDIU`	`R2,R2,#8`
	`stall`		`DSLTU`	`R3,R1,R4`
	`stall`		`stall`	
	`stall`		`stall`	
7	`ADD.D`	`F6,F4,F6`	`ADD.D`	`F6,F4,F6`
8	`stall`		`stall`	
9	`stall`		`stall`	
10	`stall`		`BNEZ`	`R3,foo`
11	`S.D`	`F6,0(R2)`	`S.D`	`F6,-8(R2)`
12	`DADDIU`	`R1,R1,#8`		
13	`DADDIU`	`R2,R2,#8`		
14	`DSLTU`	`R3,R1,R4`		
15	`stall`			
16	`BNEZ`	`R3,foo`		
17	`stall`			

Figure S.18 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. See Figure S.19.

Clock cycle	Scheduled code	
1	`DADDIU`	`R4,R1,#800`
2	`L.D`	`F2,0(R1)`
3	`L.D`	`F6,0(R2)`
4	`MUL.D`	`F4,F2,F0`

Figure S.19 The code must be unrolled three times to eliminate stalls after scheduling.

5	L.D	F2,8(R1)
6	L.D	F10,8(R2)
7	MUL.D	F8,F2,F0
8	L.D	F2,8(R1)
9	L.D	F14,8(R2)
10	MUL.D	F12,F2,F0
11	ADD.D	F6,F4,F6
12	DADDIU	R1,R1,#24
13	ADD.D	F10,F8,F10
14	DADDIU	R2,R2,#24
15	DSLTU	R3,R1,R4
16	ADD.D	F14,F12,F14
17	S.D	F6,-24(R2)
18	S.D	F10,-16(R2)
19	BNEZ	R3,foo
20	S.D	F14,-8(R2)

Figure S.19 *Continued*

c. See Figures S.20 and S.21.

Unrolled 6 times:

Cycle	Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
1	L.D F1,0(R1)	L.D F2,8(R1)			
2	L.D F3,16(R1)	L.D F4,24(R1)			
3	L.D F5,32(R1)	L.D F6,40(R1)	MUL.D F1,F1,F0	MUL.D F2,F2,F0	
4	L.D F7,0(R2)	L.D F8,8(R2)	MUL.D F3,F3,F0	MUL.D F4,F4,F0	
5	L.D F9,16(R2)	L.D F10,24(R2)	MUL.D F5,F5,F0	MUL.D F6,F6,F0	
6	L.D F11,32(R2)	L.D F12,40(R2)			
7					DADDIU R1,R1,48
8					DADDIU R2,R2,48

Figure S.20 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.

9			ADD.D F7,F7,F1	ADD.D F8,F8,F2	
10			ADD.D F9,F9,F3	ADD.D F10,F10,F4	
11			ADD.D F11,F11,F5	ADD.D F12,F12,F6	
12					DSLTU R3,R1,R4
13	S.D F7,-48(R2)	S.D F8,-40(R2)			
14	S.D F9,-32(R2)	S.D F10,-24(R2)			
15	S.D F11,-16(R2)	S.D F12,-8(R2)			BNEZ R3,foo

Figure S.20 ***Continued***

Unrolled 10 times:

Cycle	Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
1	L.D F1,0(R1)	L.D F2,8(R1)			
2	L.D F3,16(R1)	L.D F4,24(R1)			
3	L.D F5,32(R1)	L.D F6,40(R1)	MUL.D F1,F1,F0	MUL.D F2,F2,F0	
4	L.D F7,48(R1)	L.D F8,56(R1)	MUL.D F3,F3,F0	MUL.D F4,F4,F0	
5	L.D F9,64(R1)	L.D F10,72(R1)	MUL.D F5,F5,F0	MUL.D F6,F6,F0	
6	L.D F11,0(R2)	L.D F12,8(R2)	MUL.D F7,F7,F0	MUL.D F8,F8,F0	
7	L.D F13,16(R2)	L.D F14,24(R2)	MUL.D F9,F9,F0	MUL.D F10,F10,F0	DADDIU R1,R1,48
8	L.D F15,32(R2)	L.D F16,40(R2)			DADDIU R2,R2,48
9	L.D F17,48(R2)	L.D F18,56(R2)	ADD.D F11,F11,F1	ADD.D F12,F12,F2	
10	L.D F19,64(R2)	L.D F20,72(R2)	ADD.D F13,F13,F3	ADD.D F14,F14,F4	
11			ADD.D F15,F15,F5	ADD.D F16,F16,F6	
12			ADD.D F17,F17,F7	ADD.D F18,F18,F8	DSLTU R3,R1,R4
13	S.D F11,-80(R2)	S.D F12,-72(R2)	ADD.D F19,F19,F9	ADD.D F20,F20,F10	
14	S.D F13,-64(R2)	S.D F14,-56(R2)			
15	S.D F15,-48(R2)	S.D F16,-40(R2)			
16	S.D F17,-32(R2)	S.D F18,-24(R2)			
17	S.D F19,-16(R2)	S.D F20,-8(R2)			BNEZ R3,foo

Figure S.21 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.

3.14 a. See Figure S.22.

Iteration	Instruction	Issues at	Executes/ Memory	Write CDB at	Comment
1	`L.D F2,0(R1)`	1	2	3	First issue
1	`MUL.D F4,F2,F0`	2	4	19	Wait for F2 Mult rs [3–4] Mult use [5–18]
1	`L.D F6,0(R2)`	3	4	5	Ldbuf [4]
1	`ADD.D F6,F4,F6`	4	20	30	Wait for F4 Add rs [5–20] Add use [21–29]
1	`S.D F6,0(R2)`	5	31		Wait for F6 Stbuf1 [6–31]
1	`DADDIU R1,R1,#8`	6	7	8	
1	`DADDIU R2,R2,#8`	7	8	9	
1	`DSLTU R3,R1,R4`	8	9	10	
1	`BNEZ R3,foo`	9	11		Wait for R3
2	`L.D F2,0(R1)`	10	12	13	Wait for BNEZ Ldbuf [11–12]
2	`MUL.D F4,F2,F0`	11	~~14~~ 19	34	Wait for F2 Mult busy Mult rs [12–19] Mult use [20–33]
2	`L.D F6,0(R2)`	12	13	14	Ldbuf [13]
2	`ADD.D F6,F4,F6`	13	35	45	Wait for F4 Add rs [14–35] Add use [36–44]
2	`S.D F6,0(R2)`	14	46		Wait for F6 Stbuf [15–46]
2	`DADDIU R1,R1,#8`	15	16	17	
2	`DADDIU R2,R2,#8`	16	17	18	
2	`DSLTU R3,R1,R4`	17	18	20	
2	`BNEZ R3,foo`	18	20		Wait for R3
3	`L.D F2,0(R1)`	19	21	22	Wait for BNEZ Ldbuf [20–21]
3	`MUL.D F4,F2,F0`	20	~~23~~ 34	49	Wait for F2 Mult busy Mult rs [21–34] Mult use [35–48]
3	`L.D F6,0(R2)`	21	22	23	Ldbuf [22]
3	`ADD.D F6,F4,F6`	22	50	60	Wait for F4 Add rs [23–49] Add use [51–59]

Figure S.22 Solution for exercise 3.14a.

3	S.D F6,0(R2)	23	55		Wait for F6 Stbuf [24–55]
3	DADDIU R1,R1,#8	24	25	26	
3	DADDIU R2,R2,#8	25	26	27	
3	DSLTU R3,R1,R4	26	27	28	
3	BNEZ R3,foo	27	29		Wait for R3

Figure S.22 ***Continued***

b. See Figure S.23.

Iteration	Instruction	Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D F2,0(R1)	1	2	3	
1	MUL.D F4,F2,F0	1	4	19	Wait for F2 Mult rs [2–4] Mult use [5]
1	L.D F6,0(R2)	2	3	4	Ldbuf [3]
1	ADD.D F6,F4,F6	2	20	30	Wait for F4 Add rs [3–20] Add use [21]
1	S.D F6,0(R2)	3	31		Wait for F6 Stbuf [4–31]
1	DADDIU R1,R1,#8	3	4	5	
1	DADDIU R2,R2,#8	4	5	6	
1	DSLTU R3,R1,R4	4	6	7	INT busy INT rs [5–6]
1	BNEZ R3,foo	5	7		INT busy INT rs [6–7]
2	L.D F2,0(R1)	6	8	9	Wait for BEQZ
2	MUL.D F4,F2,F0	6	10	25	Wait for F2 Mult rs [7–10] Mult use [11]
2	L.D F6,0(R2)	7	9	10	INT busy INT rs [8–9]
2	ADD.D F6,F4,F6	7	26	36	Wait for F4 Add RS [8–26] Add use [27]
2	S.D F6,0(R2)	8	37		Wait for F6
2	DADDIU R1,R1,#8	8	10	11	INT busy INT rs [8–10]

Figure S.23 Solution for exercise 3.14b.

2	DADDIU R2,R2,#8	9	11	12	INT busy INT rs [10–11]
2	DSLTU R3,R1,R4	9	12	13	INT busy INT rs [10–12]
2	BNEZ R3,foo	10	14		Wait for R3
3	L.D F2,0(R1)	11	15	16	Wait for BNEZ
3	MUL.D F4,F2,F0	11	17	32	Wait for F2 Mult rs [12–17] Mult use [17]
3	L.D F6,0(R2)	12	16	17	INT busy INT rs [13–16]
3	ADD.D F6,F4,F6	12	33	43	Wait for F4 Add rs [13–33] Add use [33]
3	S.D F6,0(R2)	14	44		Wait for F6 INT rs full in 15
3	DADDIU R1,R1,#8	15	17		INT rs full and busy INT rs [17]
3	DADDIU R2,R2,#8	16	18		INT rs full and busy INT rs [18]
3	DSLTU R3,R1,R4	20	21		INT rs full
3	BNEZ R3,foo	21	22		INT rs full

Figure S.23 *Continued*

3.15 See Figure S.24.

Instruction	Issues at	Executes/Memory	Write CDB at
ADD.D F2,F4,F6	1	2	12
ADD R1,R1,R2	2	3	4
ADD R1,R1,R2	3	5	6
ADD R1,R1,R2	4	7	8
ADD R1,R1,R2	5	9	10
ADD R1,R1,R2	6	11	12 (CDB conflict)

Figure S.24 Solution for exercise 3.15.

3.16 See Figures S.25 and S.26.

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”

Figure S.25 Individual branch outcomes, in order of execution. Misprediction rate = 3/9 = .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”

Figure S.26 Individual branch outcomes, in order of execution. Misprediction rate = 3/9 = .33.

3.17 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: CPI_{noBTB} and CPI_{BTB} and the resulting speedup given by the BTB:

$$Speedup = \frac{CPI_{noBTB}}{CPI_{BTB}} = \frac{CPI_{base} + Stalls_{base}}{CPI_{base} + Stalls_{BTB}}$$

$$Stalls_{noBTB} = 15\% \times 2 = 0.30$$

To compute $Stalls_{BTB}$, 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

Figure S.27 Weighted penalties for possible branch outcomes.

Therefore:

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

$$Speedup = \frac{1.0 + 0.30}{1.0 + 0.097} = 1.2$$

3.18 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% × (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% × (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 **MIPS code (answers may vary)**

```
       li      $r1,#0              # initialize k
loop:  l.s     $f0,0($RtipL)       # load all values for first
                                   expression
       l.s     $f1,0($RclL)
       l.s     $f2,4($RtipL)
       l.s     $f3,4($RclL)
       l.s     $f4,8($RtipL)
       l.s     $f5,8($RclL)
       l.s     $f6,12($RtipL)
       l.s     $f7,12($RclL)
       l.s     $f8,0($RtipR)
       l.s     $f9,0($RclR)
       l.s     $f10,4($RtipR)
       l.s     $f11,4($RclR)
       l.s     $f12,8($RtipR)
       l.s     $f13,8($RclR)
       l.s     $f14,12($RtipR)
       l.s     $f15,12($RclR)
       mul.s   $f16,$f0,$f1        # first four multiplies
       mul.s   $f17,$f2,$f3
       mul.s   $f18,$f4,$f5
       mul.s   $f19,$f6,$f7
       add.s   $f20,$f16,$f17      # accumulate
       add.s   $f20,$f20,$f18
       add.s   $f20,$f20,$f19
       mul.s   $f16,$f8,$f9        # second four multiplies
       mul.s   $f17,$f10,$f11
       mul.s   $f18,$f12,$f13
       mul.s   $f19,$f14,$f15
       add.s   $f21,$f16,$f17      # accumulate
       add.s   $f21,$f21,$f18
       add.s   $f21,$f21,$f19
       mul.s   $f20,$f20,$f21      # final multiply
       st.s    $f20,0($RclP)       # store result
       add     $RclP,$RclP,#4      # increment clP for next
                                   expression
       add     $RtiPL,$RtiPL,#16   # increment tiPL for next
                                   expression
       add     $RtiPR,$RtiPR,#16   # increment tiPR for next
                                   expression
       addi    $r1,$r1,#1
       and     $r2,$r2,#3          # check to see if we should
                                   increment clL and clR (every
                                   4 bits)
       bneq    $r2,skip
       add     $RclL,$RclL,#16     # increment tiPL for next loop
                                   iteration
       add     $RclR,$RclR,#16     # increment tiPR for next loop
                                   iteration
skip:  blt     $r1,$r3,loop        # assume r3 = seq_length * 4
```

VMIPS code (answers may vary)

```
       li      $r1,#0              # initialize k
       li      $VL,#4              # initialize vector length
loop:  lv      $v0,0($RclL)
       lv      $v1,0($RclR)
       lv      $v2,0($RtipL)       # load all tipL values
       lv      $v3,16($RtipL)
       lv      $v4,32($RtipL)
       lv      $v5,48($RtipL)
       lv      $v6,0($RtipR)       # load all tipR values
       lv      $v7,16($RtipR)
       lv      $v8,32($RtipR)
       lv      $v9,48($RtipR)
       mulvv.s $v2,$v2,$v0         # multiply left
                                   sub-expressions
       mulvv.s $v3,$v3,$v0
       mulvv.s $v4,$v4,$v0
       mulvv.s $v5,$v5,$v0
       mulvv.s $v6,$v6,$v1         # multiply right
                                   sub-expression
       mulvv.s $v7,$v7,$v1
       mulvv.s $v8,$v8,$v1
       mulvv.s $v9,$v9,$v1
       sumr.s  $f0,$v2             # reduce left sub-expressions
       sumr.s  $f1,$v3
       sumr.s  $f2,$v4
       sumr.s  $f3,$v5
       sumr.s  $f4,$v6             # reduce right
                                   sub-expressions
       sumr.s  $f5,$v7
       sumr.s  $f6,$v8
       sumr.s  $f7,$v9
       mul.s   $f0,$f0,$f4         # multiply left and right
                                   sub-expressions
```

```
mul.s   $f1,$f1,$f5
mul.s   $f2,$f2,$f6
mul.s   $f3,$f3,$f7
s.s     $f0,0($Rclp)        # store results
s.s     $f1,4($Rclp)
s.s     $f2,8($Rclp)
s.s     $f3,12($Rclp)
add     $RtiPL,$RtiPL,#64   # increment tiPL for next
                            expression
add     $RtiPR,$RtiPR,#64   # increment tiPR for next
                            expression
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    $r1,$r1,#1
blt     $r1,$r3,loop        # assume r3 = seq_length
```

4.2 MIPS: loop is 41 instructions, will iterate 500 × 4 = 2000 times, so roughly 82000 instructions
VMIPS: loop is also 41 instructions but will iterate only 500 times, so roughly 20500 instructions

4.3
```
1.    lv                        # clL
2.    lv                        # clR
3.    lv        mulvv.s         # tiPL 0
4.    lv        mulvv.s         # tiPL 1
5.    lv        mulvv.s         # tiPL 2
6.    lv        mulvv.s         # tiPL 3
7.    lv        mulvv.s         # tiPR 0
8.    lv        mulvv.s         # tiPR 1
9.    lv        mulvv.s         # tiPR 2
10.   lv        mulvv.s         # tiPR 3
11.   sumr.s
12.   sumr.s
13.   sumr.s
14.   sumr.s
15.   sumr.s
16.   sumr.s
17.   sumr.s
18.   sumr.s
```

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

4.4 In this case, the 16 values could be loaded into each vector register, performing vector multiplies from four iterations of the loop in single vector multiply instructions. This could reduce the iteration count of the loop by a factor of 4. However, without a way to perform reductions on a subset of vector elements, this technique cannot be applied to this code.

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*16+AA]*clL_s[A] +
tiPL[k*16+AC]*clL_s[C] + tiPL[k*16+AG]*clL_s[G] +
tiPL[k*16+AT]*clL_s[T])*(tiPR[k*16+AA]*clR_s[A] +
tiPR[k*16+AC]*clR_s[C] + tiPR[k*16+AG]*clR_s[G] +
tiPR[k*16+AT]*clR_s[T]);

    clP[k*4+1] = (tiPL[k*16+CA]*clL_s[A] +
tiPL[k*16+CC]*clL_s[C] + tiPL[k*16+CG]*clL_s[G] +
tiPL[k*16+CT]*clL_s[T])*(tiPR[k*16+CA]*clR_s[A] +
tiPR[k*16+CC]*clR_s[C] + tiPR[k*16+CG]*clR_s[G] +
tiPR[k*16+CT]*clR_s[T]);

    clP[k*4+2] = (tiPL[k*16+GA]*clL_s[A] +
tiPL[k*16+GC]*clL_s[C] + tiPL[k*16+GG]*clL_s[G] +
tiPL[k*16+GT]*clL_s[T])*(tiPR[k*16+GA]*clR_s[A] +
tiPR[k*16+GC]*clR_s[C] + tiPR[k*16+GG]*clR_s[G] +
tiPR[k*16+GT]*clR_s[T]);

    clP[k*4+3] = (tiPL[k*16+TA]*clL_s[A] +
tiPL[k*16+TC]*clL_s[C] + tiPL[k*16+TG]*clL_s[G] +
tiPL[k*16+TT]*clL_s[T])*(tiPR[k*16+TA]*clR_s[A] +
tiPR[k*16+TC]*clR_s[C] + tiPR[k*16+TG]*clR_s[G] +
tiPR[k*16+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*16+AA + blockIdx.x*2*500*16]
tipL[threadIdx.x*16+AC + blockIdx.x*2*500*16]
…
tipL[threadIdx.x*16+TT + blockIdx.x*2*500*16]

tipR[threadIdx.x*16+AA + (blockIdx.x*2+1)*500*16]
tipR[threadIdx.x*16+AC +1 + (blockIdx.x*2+1)*500*16]
…
tipR[threadIdx.x*16+TT +15+ (blockIdx.x*2+1)*500*16]
```

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, 24000     # 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, since 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         $VL,44         # perform the first 44 ops
       li         $r1,0          # initialize index
loop:  lv         $v1,a_re+$r1   # load a_re
       lv         $v3,b_re+$r1   # load b_re
       mulvv.s    $v5,$v1,$v3    # a+re*b_re
       lv         $v2,a_im+$r1   # load a_im
       lv         $v4,b_im+$r1   # load b_im
       mulvv.s    $v6,$v2,$v4    # a+im*b_im
       subvv.s    $v5,$v5,$v6    # a+re*b_re - a+im*b_im
       sv         $v5,c_re+$r1   # store c_re
       mulvv.s    $v5,$v1,$v4    # a+re*b_im
       mulvv.s    $v6,$v2,$v3    # a+im*b_re
       addvv.s    $v5,$v5,$v6    # a+re*b_im + a+im*b_re
       sv         $v5,c_im+$r1   # store c_im
       bne        $r1,0,else     # check if first iteration
       addi       $r1,$r1,#44    # first iteration,
                                 increment by 44
       j loop                    # guaranteed next iteration
else:  addi       $r1,$r1,#256   # not first iteration,
                                 increment by 256
skip:  blt        $r1,1200,loop  # next iteration?
```

c.

```
1.     mulvv.s    lv        # a_re * b_re (assume already
                            # loaded), load a_im
2.     lv         mulvv.s   # load b_im, a_im*b_im
3.     subvv.s    sv        # subtract and store c _re
4.     mulvv.s    lv        # a_re*b_im, load next a_re vector
5.     mulvv.s    lv        # a_im*b_re, load next b_re vector
6.     addvv.s    sv        # 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. mulvv.s                    # a_re*b_re
2. mulvv.s                    # a_im*b_im
3. subvv.s  sv                # subtract and store c_re
4. mulvv.s                    # a_re*b_im
5. mulvv.s  lv                # a_im*b_re, load next a_re
6. addvv.s  sv   lv   lv   lv    # 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.

```
li          $VL,4
addvv.s     $v0(0),$v0(4)
addvv.s     $v0(8),$v0(12)
addvv.s     $v0(16),$v0(20)
addvv.s     $v0(24),$v0(28)
addvv.s     $v0(32),$v0(36)
addvv.s     $v0(40),$v0(44)
addvv.s     $v0(48),$v0(52)
addvv.s     $v0(56),$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 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) × (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 × .80 × .85 × 0.70 × 10 cores × 32/4 = 57.12 GFLOPs/s

b. **Option 1:**

1.5 GHz × .80 × .85 × .70 × 10 cores × 32/2 = 114.24 GFLOPs/s (speedup = 114.24/57.12 = 2)

Option 2:

1.5 GHz × .80 × .85 × .70 × 15 cores × 32/4 = 85.68 GFLOPs/s (speedup = 85.68/57.12 = 1.5)

Option 3:

1.5 GHz × .80 × .95 × .70 × 10 cores × 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 × 16 × 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 × 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 a. P0: read 120 → P0.B0: (S, 120, 0020) returns 0020

b. P0: write 120 ← 80 → P0.B0: (M, 120, 0080)
P3.B0: (I, 120, 0020)

c. P3: write 120 ← 80 → P3.B0: (M, 120, 0080)

d. P1: read 110 → P1.B2: (S, 110, 0010) returns 0010

e. P0: write 108 ← 48 → P0.B1: (M, 108, 0048)
P3.B1: (I, 108, 0008)

f. P0: write 130 ← 78 → P0.B2: (M, 130, 0078)
M: 110 ← 0030 (writeback to memory)

g. P3: write 130 ← 78 → P3.B2: (M, 130, 0078)

5.2 a. P0: read 120, Read miss, satisfied by memory
P0: read 128, Read miss, satisfied by P1's cache
P0: read 130, Read miss, satisfied by memory, writeback 110
Implementation 1: 100 + 40 + 10 + 100 + 10 = 260 stall cycles
Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles

b. P0: read 100, Read miss, satisfied by memory
P0: write 108 ← 48, Write hit, sends invalidate
P0: write 130 ← 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. P1: read 120, Read miss, satisfied by memory
P1: read 128, Read hit
P1: read 130, Read miss, satisfied by memory
Implementation 1: 100 + 0 + 100 = 200 stall cycles
Implementation 2: 100 + 0 + 100 = 200 stall cycles

d. P1: read 100, Read miss, satisfied by memory
P1: write 108 ← 48, Write miss, satisfied by memory, write back 128
P1: write 130 ← 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.28

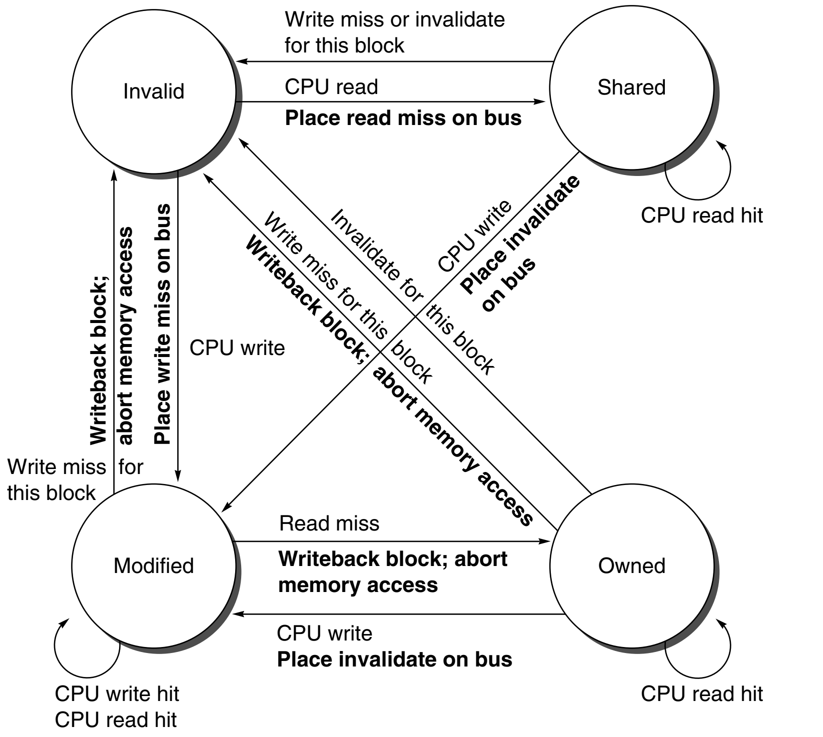

Invalid
Shared
Modified
Owned
Write miss or invalidate for this block
CPU read
Place read miss on bus
CPU read hit
Invalidate for this block
Write miss for this block
Writeback block; abort memory access
CPU write
Place invalidate on bus
CPU write
Place write miss on bus
Write miss for this block
Writeback block; abort memory access
Read miss
Writeback block; abort memory access
CPU write
Place invalidate on bus
CPU write hit
CPU read hit
CPU read hit

Figure S.28 Protocol diagram.

5.4 (Showing results for implementation 1)

a. P1: read 110, Read miss, P0's cache

P3: read 110, Read miss, MSI satisfies in memory, MOSI satisfies in P0's cache

P0: read 110, Read hit

MSI: 40 + 10 + 100 + 0 = 150 stall cycles

MOSI: 40 + 10 + 40 + 10 + 0 = 100 stall cycles

b. P1: read 120, Read miss, satisfied in memory

P3: read 120, Read hit

P0: read 120, Read miss, satisfied in memory

Both protocols: 100 + 0 + 100 = 200 stall cycles

c. P0: write 120 ← 80, Write miss, invalidates P3

P3: read 120, Read miss, P0's cache

P0: read 120, Read hit

Both protocols: 100 + 40 + 10 + 0 = 150 stall cycles

d. P0: write 108 ← 88, Send invalidate, invalidate P3
   P3: read 108, Read miss, P0's cache
   P0: write 108 ← 98, Send invalidate, invalidate P3
   Both protocols: 15 + 40 + 10 + 15 = 80 stall cycles

5.5 See Figure S.29

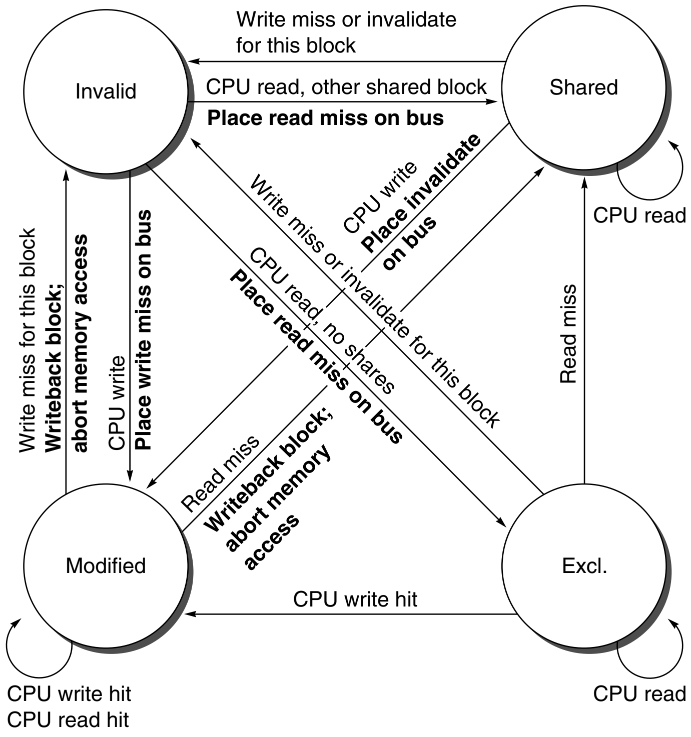

Invalid
Shared
Modified
Excl.
Write miss or invalidate for this block
CPU read, other shared block
Place read miss on bus
CPU write
Place invalidate on bus
CPU read hit
Write miss or invalidate for this block
CPU read, no shares
Place read miss on bus
Read miss
Write miss for this block
Writeback block; abort memory access
CPU write
Place write miss on bus
Read miss
Writeback block; abort memory access
CPU write hit
CPU write hit
CPU read hit
CPU read hit

Figure S.29 Diagram for a MESI protocol.

5.6 a. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
   p0: write 100 ← 40, MSI: send invalidate, MESI: silent transition from E to M
   MSI: 100 + 15 = 115 stall cycles
   MESI: 100 + 0 = 100 stall cycles

b. p0: read 120, Read miss, satisfied in memory, sharers both to S
   p0: write 120 ← 60, Both send invalidates
   Both: 100 + 15 = 115 stall cycles

c. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E
   p0: read 120, Read miss, memory, silently replace 120 from S or E
   Both: 100 + 100 = 200 stall cycles, silent replacement from E

d. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E

   p1: write 100 ← 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. p0: read 100, Read miss, satisfied in memory, no sharers MSI: S, MESI: E

   p0: write 100 ← 60, MSI: send invalidate, MESI: silent transition from E to M

   p1: write 100 ← 40, Write miss, P0's cache, writeback data to memory

   MSI: 100 + 15 + 40 + 10 = 165 stall cycles

   MESI: 100 + 0 + 40 + 10 = 150 stall cycles

5.7 a. Assume the processors acquire the lock in order. P0 will acquire it first, incurring 100 stall cycles to retrieve the block from memory. P1 and P3 will stall until P0's critical section ends (ping-ponging the block back and forth) 1000 cycles later. P0 will stall for (about) 40 cycles while it fetches the block to invalidate it; then P1 takes 40 cycles to acquire it. P1's critical section is 1000 cycles, plus 40 to handle the write miss at release. Finally, P3 grabs the block for a final 40 cycles of stall. So, P0 stalls for 100 cycles to acquire, 10 to give it to P1, 40 to release the lock, and a final 10 to hand it off to P1, for a total of 160 stall cycles. P1 essentially stalls until P0 releases the lock, which will be 100 + 1000 + 10 + 40 = 1150 cycles, plus 40 to get the lock, 10 to give it to P3, 40 to get it back to release the lock, and a final 10 to hand it back to P3. This is a total of 1250 stall cycles. P3 stalls until P1 hands it off the released lock, which will be 1150 + 40 + 10 + 1000 + 40 = 2240 cycles. Finally, P3 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 (15) and 1 write miss to a cache (40 + 10) and 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 and 1 read miss to memory (40 + 10 + 100), followed by an upgrade (15) and a write miss to cache (40 + 10), and finally a write miss to cache (40 + 10) followed by a read miss to cache (40 + 10) and an upgrade (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 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 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 (1).

5.8 Latencies in implementation 1 of Figure 5.36 are used.

a.	P0: write 110 ← 80	Hit in P0's cache, no stall cycles for either TSO or SC
	P0: read 108	Hit in P0's cache, no stall cycles for either TSO or SC
b.	P0: write 100 ← 80	Miss, TSO satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles)
	P0: read 108	Hit, but must wait for preceding operation: TSO = 0, SC = 100
c.	P0: write 110 ← 80	Hit in P0's cache, no stall cycles for either TSO or SC
	P0: write 100 ← 90	Miss, TSO satisfies write in write buffer (0 stall cycles) SC must wait until it receives the data (100 stall cycles)
d.	P0: write 100 ← 80	Miss, TSO 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: TSO = 0, SC = 100

Case Study 2: Simple Directory-Based Coherence

5.9

a.	P0,0: read 100	L1 hit returns 0x0010, state unchanged (M)
b.	P0,0: read 128	L1 miss and L2 miss will replace B1 in L1 and B1 in L2 which has address 108.
		L1 will have 128 in B1 (shared), L2 also will have it (DS, P0,0)
		Memory directory entry for 108 will become <DS, C1>
		Memory directory entry for 128 will become <DS, C0>

c, d, …, h: follow same approach

5.10 a. P0,0: write 100 ← 80, Write hit only seen by P0,0

b. P0,0: write 108 ← 88, Write "upgrade" received by P0,0; invalidate received by P3,1

c. P0,0: write 118 ← 90, Write miss received by P0,0; invalidate received by P1,0

d. P1,0: write 128 ← 98, Write miss received by P1,0.

5.11 a. See Figures S.30 and S.31

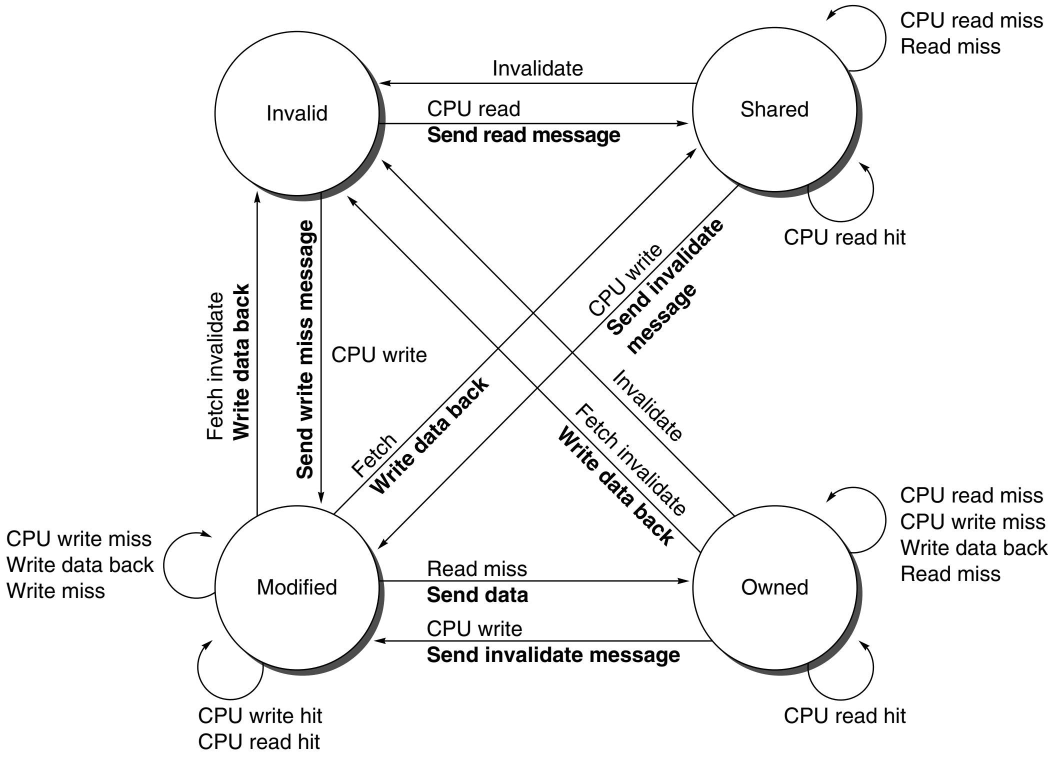


Figure S.30 Cache states.

5.12 The Exclusive state (E) combines properties of Modified (M) and Shared (S). The E state allows silent upgrades to M, allowing the processor to write the block without communicating this fact to memory. It also allows silent downgrades to I, allowing the processor to discard its copy with notifying memory. The memory must have a way of inferring either of these transitions. In a directory-based system, this is typically done by having the directory assume that the node is in state M and forwarding all misses to that node. If a node has silently downgraded to I, then it sends a NACK (Negative Acknowledgment) back to the directory, which then infers that the downgrade occurred. However, this results in a race with other messages, which can cause other problems.

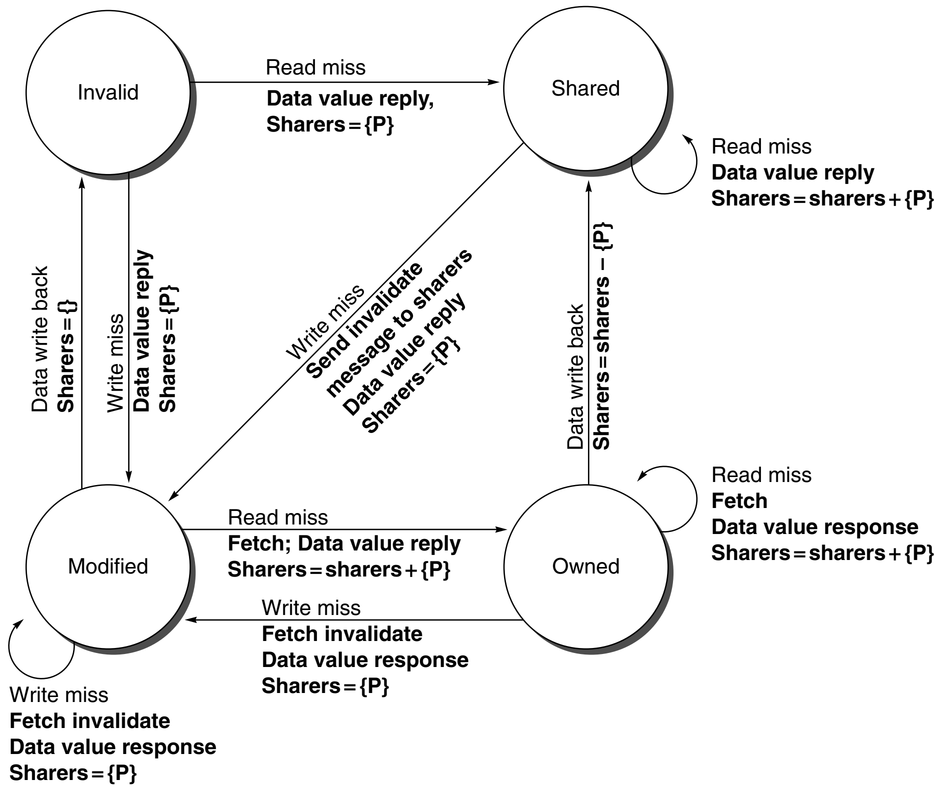


Figure S.31 **Directory states.**

Case Study 3: Advanced Directory Protocol

5.13 a. P0,0: read 100 — Read hit

b. P0,0: read 120 — Miss, will replace modified data (B0) and get new line in shared state

P0,0: M → MI^A → I → IS^D → S   Dir: DM {P0,0} → DI { }

c. P0,0: write 120 ← 80 — Miss will replace modified data (B0) and get new line in modified state

P0,0: M → MI^A → I → IM^{AD} → IM^A → M

P3,1: S → I

Dir: DS {P3,0} → DM {P0,0}

d, e, f: steps similar to parts a, b, and c

5.14 a. P0,0: read 120 — Miss, will replace modified data (B0) and get new line in shared state

P0,0: M → MI^A → I → IS^D → S

	P1,0: read 120	Miss, will replace modified data (B0) and get new line in shared state P1,0: M $\rightarrow$ MI^A $\rightarrow$ I $\rightarrow$ IS^D $\rightarrow$ S Dir: DS {P3,0} $\rightarrow$ DS {P3,0; P0,0} $\rightarrow$ DS {P3,0; P0,0; P1,0}
b.	P0,0: read 120	Miss, will replace modified data (B0) and get new line in shared state P0,0: M $\rightarrow$ MI^A $\rightarrow$ I $\rightarrow$ IS^D $\rightarrow$ S
	P1,0: write 120 $\leftarrow$ 80	Miss will replace modified data (B0) and get new line in modified state P1,0: M $\rightarrow$ MI^A $\rightarrow$ I $\rightarrow$ IM^{AD} $\rightarrow$ IM^A $\rightarrow$ M P3,1: S $\rightarrow$ I Dir: DS {P3,1} $\rightarrow$ DS {P3,0; P1,0} $\rightarrow$ DM {P1,0}

c, d, e: steps similar to parts a and b

5.15

a.	P0,0: read 100	Read hit, 1 cycle
b.	P0,0: read 120	Read Miss, causes modified block replacement and is satisfied in memory and incurs 4 chip crossings (see underlined)
	Latency for P0,0:	Lsend_data + $\underline{\text{Ldata\_msg}}$ + Lwrite_memory + Linv + L_ack + $\underline{\text{Lreq\_msg}}$ + Lsend_msg + $\underline{\text{Lreq\_msg}}$ + Lread_memory + $\underline{\text{Ldata\_msg}}$ + Lrcv_data + 4 × chip crossings latency = 20 + 30 + 20 + 1 + 4 + 15 + 6 + 15 + 100 + 30 + 15 + 4 × 20 = 336

c, d, e: follow same steps as a and b

5.16 All protocols must ensure forward progress, even under worst-case memory access patterns. It is crucial that the protocol implementation guarantee (at least with a probabilistic argument) that a processor will be able to perform at least one memory operation each time it completes a cache miss. Otherwise, starvation might result. Consider the simple spin lock code:

```
tas:
DADDUI R2, R0, #1
lockit:
EXCH R2, 0(R1)
BNEZ R2, lockit
```

If all processors are spinning on the same loop, they will all repeatedly issue GetM messages. If a processor is not guaranteed to be able to perform at least one instruction, then each could steal the block from the other repeatedly. In the worst case, no processor could ever successfully perform the exchange.

5.17 a. The MS^A state is essentially a "transient O" because it allows the processor to read the data and it will respond to GetShared and GetModified requests from other processors. It is transient, and not a real O state, because memory will send the PutM_Ack and take responsibility for future requests.

b. See Figures S.32 and S.33

State	Read	Write	Replace-ment	INV	Forwarded_ GetS	Forwarded_ GetM	PutM_ Ack	Data	Last ACK
I	send GetS/IS	send GetM/IM	error	send Ack/I	error	error	error	error	error
S	do Read	send GetM/IM	I	send Ack/I	error	error	error	error	error
O	do Read	send GetM/OM	send PutM/OI	error	send Data	send Data/I	error	—	—
M	do Read	do Write	send PutM/MI	error	send Data/O	send Data/I	error	error	error
IS	z	z	z	send Ack/ISI	error	error	error	save Data, do Read/S	error
ISI	z	z	z	send Ack	error	error	error	save Data, do Read/I	error
IM	z	z	z	send Ack	IMO	IMI^A	error	save Data	do Write/M
IMI	z	z	z	error	error	error	error	save Data	do Write, send Data/I
IMO	z	z	z	send Ack/IMI	—	IMOI	error	save Data	do Write, send Data/O
IMOI	z	z	z	error	error	error	error	save Data	do Write, send Data/I
OI	z	z	z	error	send Data	send Data	/I	error	error
MI	z	z	z	error	send Data	send Data	/I	error	error
OM	z	z	z	error	send Data	send Data/IM	error	save Data	do Write/M

Figure S.32 Directory protocol cache controller transitions.

State	Read	Write	Replacement (owner)	INV (nonowner)
DI	send Data, add to sharers/DS	send Data, clear sharers, set owner/DM	error	send PutM_Ack
DS	send Data, add to sharers	send INVs to sharers, clear sharers, set owner, send Data/DM	error	send PutM_Ack
DO	forward GetS, add to sharers	forward GetM, send INVs to sharers, clear sharers, set owner/DM	send Data, send PutM_Ack/DS	send PutM_Ack
DM	forward GetS, add to requester and owner to sharers/DO	forward GetM, send INVs to sharers, clear sharers, set owner	send Data, send PutM_Ack/DI	send PutM_Ack

Figure S.33 Directory controller transitions.

5.18 a. P1,0: read 100

P3,1: write 100 $\leftarrow$ 90

In this problem, both P0,1 and P3,1 miss and send requests that race to the directory. Assuming that P0,1's GetS request arrives first, the directory will forward P0,1's GetS to P0,0, followed shortly afterwards by P3,1's GetM. If the network maintains point-to-point order, then P0,0 will see the requests in the right order and the protocol will work as expected. However, if the forwarded requests arrive out of order, then the GetX will force P0 to state I, causing it to detect an error when P1's forwarded GetS arrives.

b. P1,0: read 100

P0,0: replace 100

P1,0's GetS arrives at the directory and is forwarded to P0,0 before P0,0's PutM message arrives at the directory and sends the PutM_Ack. However, if the PutM_Ack arrives at P0,0 out of order (i.e., before the forwarded GetS), then this will cause P0,0 to transition to state I. In this case, the forwarded GetS will be treated as an error condition.

Exercises

5.19 The general form for Amdahl's Law is

$$Speedup = \frac{Execution\ time_{old}}{Execution\ time_{new}}$$

all that needs to be done to compute the formula for speedup in this multiprocessor case is 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)*. If we let *Execution time*$_{old}$ be 1, then the relative time for 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 1 and *p*. This yields

$$Execution\ time_{new} = \sum_{i=1}^{p} \frac{f(i,p)}{i}$$

Substituting this value for *Execution time*$_{new}$ into the speedup equation makes Amdahl's Law a function of the available processors, *p*.

5.20 a. (i) 64 processors arranged a as a ring: largest number of communication hops = 32 $\rightarrow$ communication cost = $(100 + 10 \times 32)$ ns = 420 ns.

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

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

b. Base CPI = 0.5 *cpi*

(i) 64 processors arranged a as a ring: Worst case CPI = $0.5 + 0.2/100 \times (420) = 1.34$ *cpi*

(ii) 64 processors arranged as 8x8 processor grid: Worst case CPI = $0.5 + 0.2/100 \times (240) = 0.98$ *cpi*

(iii) 64 processors arranged as a hypercube: Worst case CPI CPI = $0.5 + 0.2/100 \times (160) = 0.82$ *cpi*

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.

c. Since the CPU frequency and the number of instructions executed did not change, the answer can be obtained by the CPI for each of the topologies (worst case or average) by the base (no remote communication) CPI.

5.21 To keep the figures from becoming cluttered, the coherence protocol is split into two parts as was done in Figure 5.6 in the text. Figure S.34 presents the CPU portion of the coherence protocol, and Figure S.35 presents the bus portion of the protocol. In both of these figures, the 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.34 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.35. 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. Also, 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. Basically, it is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.

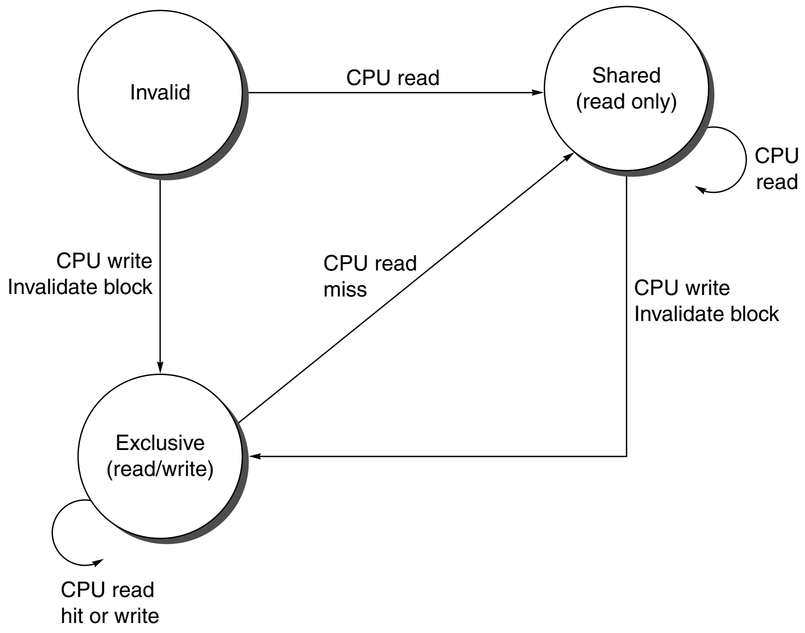


Figure S.34 CPU portion of the simple cache coherency protocol for write-through caches.

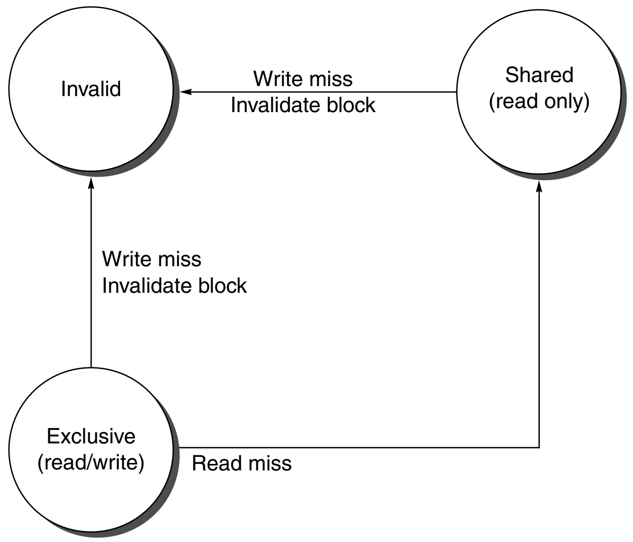


Figure S.35 Bus portion of the simple cache coherency protocol for write-through caches.

5.22 To augment the snooping protocol of Figure 5.7 with a Clean Exclusive state we assume that the cache can distinguish a read miss that will allocate a block destined to have the Clean Exclusive state from a read miss that will deliver a Shared block. Without further discussion we assume that there is some mechanism to do so.

The three states of Figure 5.7 and the transitions between them are unchanged, with the possible clarifying exception of renaming the Exclusive (read/write) state to Dirty Exclusive (read/write).

The new Clean Exclusive (read only) state should be added to the diagram along with the following transitions.

- from Clean Exclusive to Clean Exclusive in the event of a CPU read hit on this block or a CPU read miss on a Dirty Exclusive block
- from Clean Exclusive to Shared in the event of a CPU read miss on a Shared block or on a Clean Exclusive block
- from Clean Exclusive to Shared in the event of a read miss on the bus for this block
- from Clean Exclusive to Invalid in the event of a write miss on the bus for this block
- from Clean Exclusive to Dirty Exclusive in the event of a CPU write hit on this block or a CPU write miss
- from Dirty Exclusive to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block
- from Invalid to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block
- from Shared to Clean Exclusive in the event of a CPU read miss on a Dirty Exclusive block

Several transitions from the original protocol must change to accommodate the existence of the Clean Exclusive state. The following three transitions are those that change.

- from Dirty Exclusive to Shared, the label changes to CPU read miss on a Shared block
- from Invalid to Shared, the label changes to CPU miss on a Shared block
- from Shared to Shared, the miss transition label changes to CPU read miss on a Shared block

5.23 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 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 5.7 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.

5.24 a. The instruction execution component would be significantly sped up 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 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. So the speedup for this component would be lower.

The memory access time component would also be improved, 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 hidden under 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 be improved because they mainly consist of resource stalls, branch mispredictions, and the like. 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.

5.25 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 may need to be added to the program. The remainder of the cache block frame should not contain data that would cause false sharing misses. The sure solution is to pad with block 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 needed.

5.26 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.

5.27 If replacement hints are 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).

5.28 a. Considering first the storage requirements for nodes that are caches under the directory subtree:

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 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

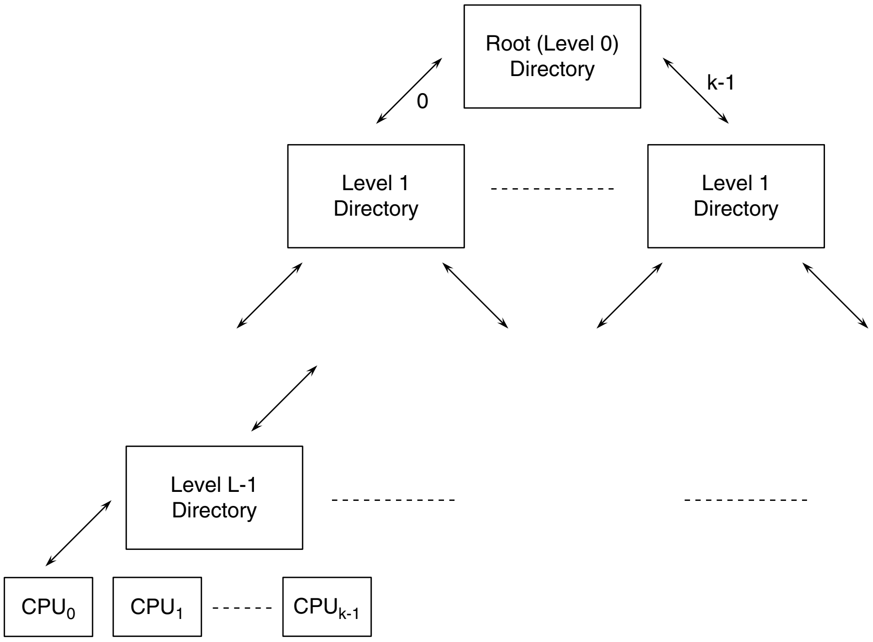


Figure S.36 Tree-based directory hierarchy (k-ary tree with l levels).

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^l – x)), where k^l 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. Simulation.

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

```
MOV R3, #1
LL  R2, 0(R1)
SC R3,  0(R1)
```

Typically this code would be put in a loop that spins until a 1 is returned in R3.

5.30 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.

5.31 The monitor has to be place 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 4 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.

5.32 a. Because flag is written only after A is written, we would expect C to be 2000, the value of A.

b. Case 1: If the write to flag reached P2 faster than the write to A.

Case 2: If the read to A was faster than the read to flag.

c. Ensure that writes by P1 are carried out in program order and that memory operations execute atomically with respect to other memory operations.

To get intuitive results of sequential consistency using barrier instructions, a barrier need to be inserted in P1 between the write to A and the write to flag.

5.33 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.

5.34 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.

5.35 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 × 1 = 45,978. The new servers provide a normalized performance of 1.1, thus the total servers required, y, is y × 1.1 = 45,978. Thus y = 41,799. The price of each server is 20% more than the baseline, and is thus 1.2 × 1450 = $1,740. 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 × 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 × watts per server + network load. Critical load = 41,799 × 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:

$$OPEX' = OPEX[(1-0.15)x + (1-x)]$$
$$CAPEX' = CAPEX[(1+0.2)y + (1-y)]$$
$$TCO' = OPEX' + CAPEX' = OPEX(1-0.15x) + CAPEX(1+0.2y)$$

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

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

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′:

$$TCO = TCO' = OPEX(1-0.15x) + CAPEX(1+Sy)$$
$$TCO - OPEX(1-0.15x) = CAPEX(1+Sy)$$

$$S = \frac{TCO - OPEX(1 - 0.15x) - CAPEX}{CAPEXy}$$

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 \$1,450, 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 × \$1,450 = \$1,160. 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:

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

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% × 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 \$1,450, a new server would cost \$3,000 – \$1,450 = \$1,550 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 \$2,000 – \$1,450 = \$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 5 nodes, each node will have to process 300 GB/5 = 60 GB. Thus 60 GB × 0.3 = 18 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 GB/100 MB/s = 180 seconds, and the time for local data access as 42 GB/200 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 GB × 10 s/GB = 600 seconds; given the reduce rate, the reduce will take 60 GB × 20 s/GB = 1200 seconds. The total expected execution time is therefore (180 + 210) × 2 + 600 + 1200 = 2508 seconds. The primary bottleneck is the reduce phase, while the total data transfer time is the secondary bottleneck. At 1,000 nodes, we have 300 GB/1000 = 300 MB per node. Thus 300 MB × 0.3 = 90 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 $(60 + 3) \times 2 + 30 + 60 = 216$. 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 $(18 + 7.2) \times 2 + 30 + 60 = 140.4$ 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 4 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 = $(4 + 1 + 1 + 1) \times 1$ hour + $(1 + 20) \times 5$ 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 $(1 - .999) \times (1 - .999) = 1e - 6$. Similarly the probability that three nodes go down is $1e - 9$.

b. Repeating the previous problem but this time with 1,000 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 – .943) × (1 – .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 – .943) × (1 – .943) × (1 – .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 half-way through the task. Thus if there is one failure, the expected runtime = 1 hour + 0.067 × (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 = $1\ hour + \sum_{n=1}^{n=inf} 0.067^n \times \left(\frac{1\ hour}{2}\right)$. 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 4,286 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 4,286 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 (.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 + 20833) × 5 minutes = 3439 + 1849 = 5288 hours. This yields 39.6% availability. Following the equations from above, we need (.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 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

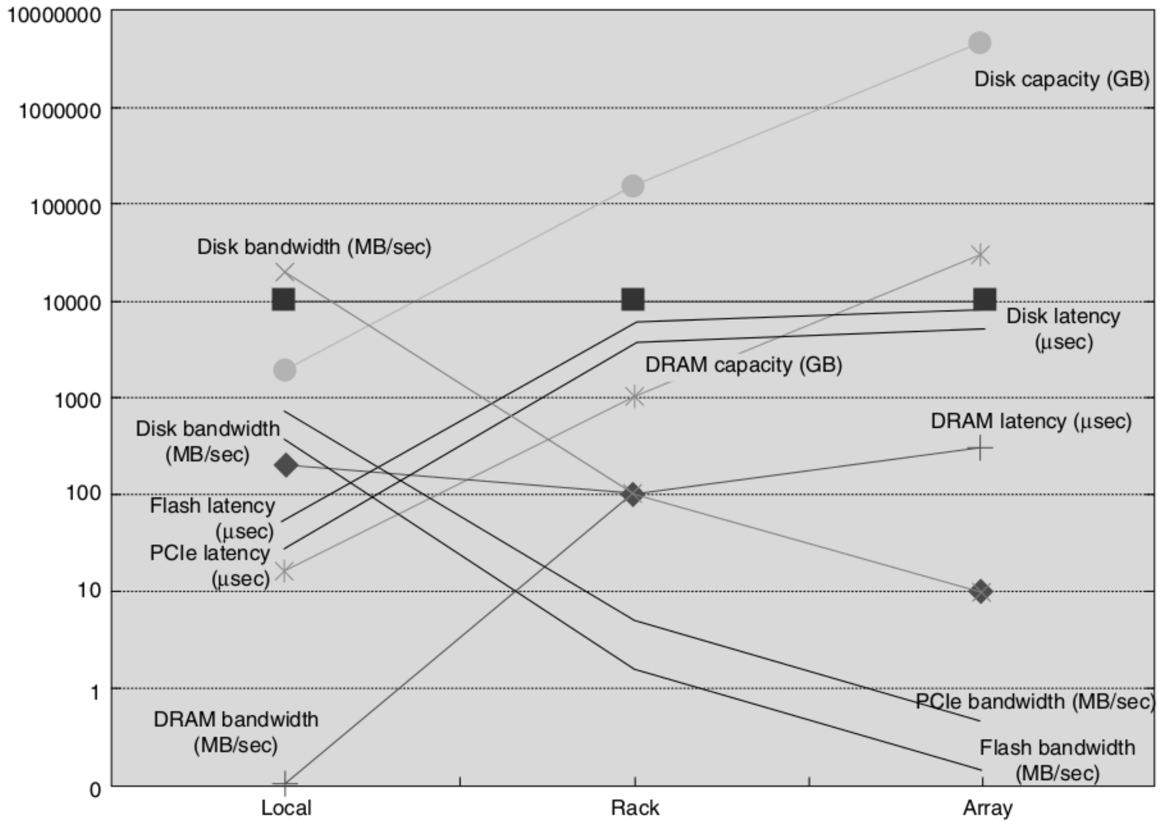


Figure S.1 Graph of latency, bandwidth, and capacity of the memory hierarchy of a WSC.

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 ≈ $\lceil 1.04d \rceil$ 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 $\lceil 1.04d \rceil$ 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/sec, or 250 Mb/sec. 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 × 250 GB)/(2 × 250 Mb/sec) = 8,000 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/sec) = 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 $30 \times 24 = 720$, 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 non-volatile 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 sec 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 $30\% \times 2\frac{2}{3} = 80\%$. 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 6.25(b), 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 × 0.78) + \$290,000 + (\$765,000 × 0.78) + \$170,000 + (\$475,000 × 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 $\lfloor 1.3X \rfloor - X$ additional servers. Similarly, at the rack and cluster level, we can afford to oversubscribe by around $\lfloor 1.2Y \rfloor - Y$ and $\lfloor 1.05Z \rfloor - Z$ servers, respectively. The total number of additional servers we can employ is given by $(\lfloor 1.3X \rfloor - X) + (\lfloor 1.2Y \rfloor - Y) + (\lfloor 1.2Z \rfloor - Z)$.

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 non-energy 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% × 98% × 98% × 99% = 94.79% efficient. A facility-wide UPS is 92% efficient, yielding a total efficiency of 94.79% × 92% = 87.21% efficiency. A battery being 99.99% efficient yields 94.79% × 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 10.72MW; 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 $66.1M 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 ~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 ~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) × 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 × electricity costs = (1 + 0.8) × (8 MW × 80%) × \$0.10 per kWH = \$1152 per hour. If we improve cooling efficiency by 20%, this yields a cooling-inefficiency of 0.8 × (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% × (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) × (8 MW × 80% × (1 – x)) × \$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%.

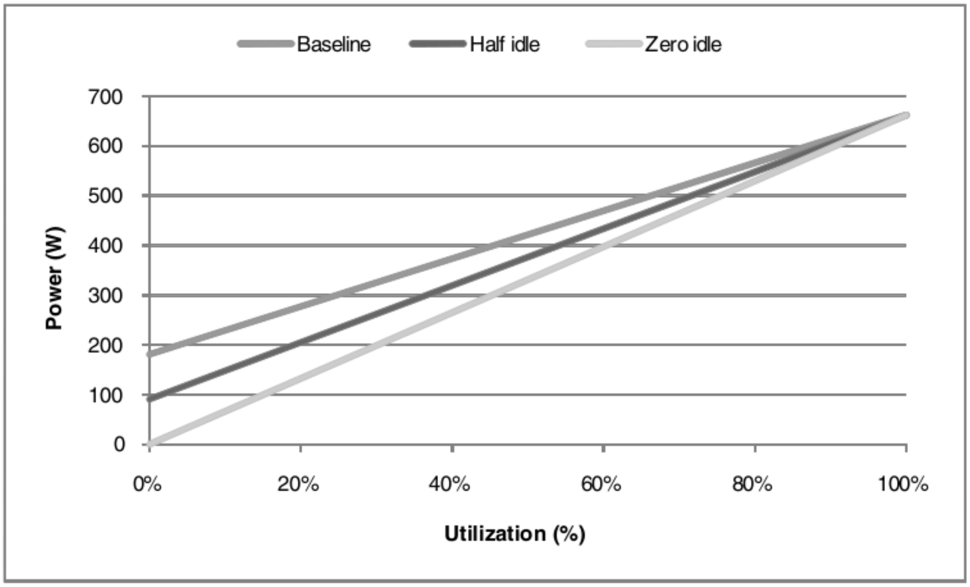
c. Overall, improving server energy efficiency will more directly lower costs rather than cooling efficiency. However, both are important to the total cost of the facility. Additionally, the datacenter operator may more directly have the ability to improve cooling efficiency, as opposed to server energy 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 deg. 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.

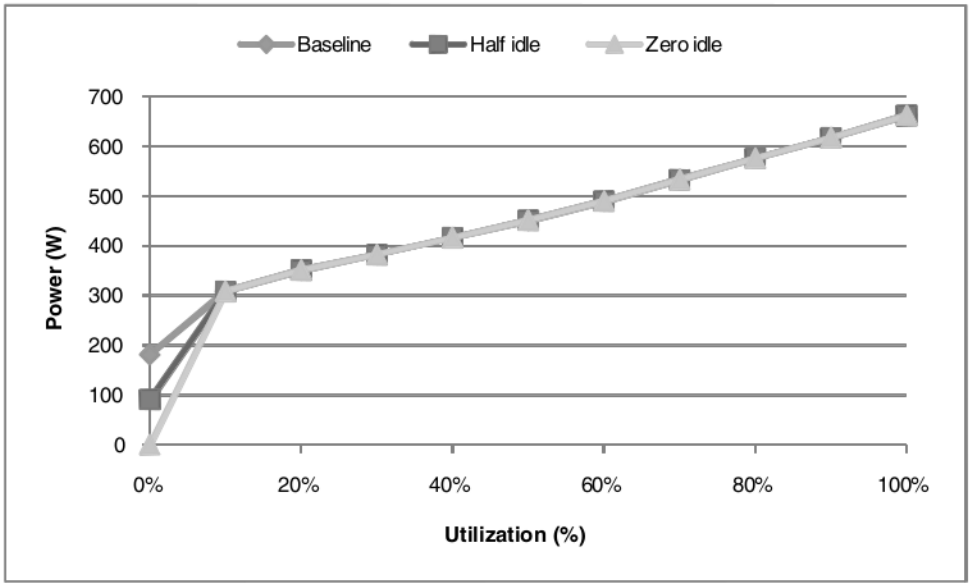
b. The second scenario is 3.23 KJ/5.26 KJ = 61.4% more efficient, providing 39.6% savings.

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.



b.

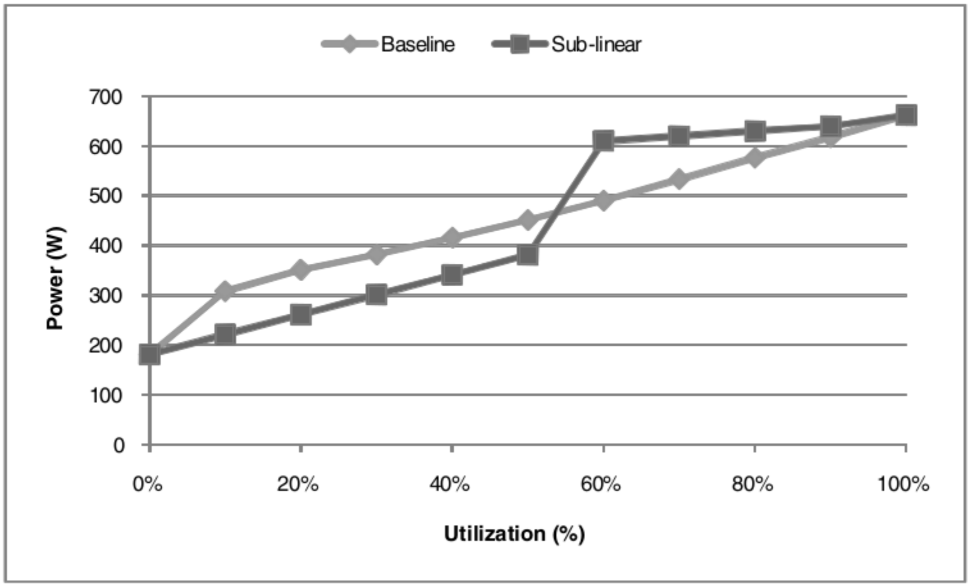


We can see that although reducing the idle power can help when the server is not utilized, there is still a strong non-linearity between 0% and 100% utilization, meaning that operating at other utilization points is likely less efficient than it can potentially be.

c. 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

d. One potential such non-linear curve is shown here:



This curve yields the following table:

	Power		Energy (WH)				Performance
	Servers	Performance	Baseline	Sub-linear	Baseline	Sub-linear	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:

	Servers	Performance	Power Case A	Power Case B	Energy (WH) Case A	Energy (WH) Case B	Performance 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:

	Servers	Performance	Power Case A	Power Case B	Power Linear	Energy (WH) Case A	Energy (WH) Case B	Energy (WH) Linear	Performance 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 2.295x and 2.044x 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 ~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 ~4X more than the baseline 50 ms response time) without a loss in revenue. They will tolerate some additional amount of latency (10X 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 rpm to 12,500 rpm, or approximately a 8.9% increase. Going from 10,000 rpm 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 12,5000 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 non-linear 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 2X 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 $50, and each server has 6 DIMMs, we obtain an uptime per dollar of 14.60 hrs/$ for the ECC DRAM, and 27.16 hrs/$ 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/hr 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.