L

Solutions to Case Study Exercises

L.1 Chapter 1 Solutions

Case Study 1: Chip Fabrication Cost

1.1 a. $$\text{Yield} = \left(1 + \frac{0.7 \times 1.99}{4.0}\right)^{-4} = 0.28$$

b. $$\text{Yield} = \left(1 + \frac{0.75 \times 3.80}{4.0}\right)^{-4} = 0.12$$

c. The Sun Niagara is substantially larger, since it places 8 cores on a chip rather than 1.

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

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{3.89} - \frac{\pi \times 30}{\text{sqrt}(2 \times 3.89)} = 182 - 33.8 = 148$$

$$\text{Cost per die} = \frac{\$500}{148 \times 0.36} = \$9.38$$

b. $$\text{Yield} = \left(1 + \frac{.7 \times 1.86}{4.0}\right)^{-4} = 0.32$$

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.86} - \frac{\pi \times 30}{\text{sqrt}(2 \times 1.86)} = 380 - 48.9 = 331$$

$$\text{Cost per die} = \frac{\$500}{331 \times .32} = \$4.72$$

c. $9.38 × .4 = $3.75

d. Selling price = ($9.38 + $3.75) × 2 = $26.26
Profit = $26.26 – $4.72 = $21.54

e. Rate of sale = 3 × 500,000 = 1,500,000/month
Profit = 1,500,000 × $21.54 = $32,310,000
$1,000,000,000/$32,310,000 = 31 months

1.3 a. $$\text{Yield} = \left(1 + \frac{.75 \times 3.80/8}{4.0}\right)^{-4} = 0.71$$

$$\text{Prob of error} = 1 - 0.71 = 0.29$$

b. Prob of one defect = $0.29 \times 0.71^7 \times 8 = 0.21$
Prob of two defects = $0.29^2 \times 0.71^6 \times 28 = 0.30$
Prob of one or two = $0.21 \times 0.30 = 0.51$

c. $0.71^8 = .06$ (now we see why this method is inaccurate!)

d. $0.51/0.06 = 8.5$

e. $x \times \$150 + 8.5x \times \$100 - (9.5x \times \$80) - 9.5x \times \$1.50 = \$200{,}000{,}000$
$x = 885{,}938$ 8-core chips, 8,416,390 chips total

Case Study 2: Power Consumption in Computer Systems

1.4 a. $.70x = 79 + 2 \times 3.7 + 2 \times 7.9x = 146$

b. $4.0\text{ W} \times .4 + 7.9\text{ W} \times .6 = 6.34\text{ W}$

c. The 7200 rpm drive takes 60 s to read/seek and 40 s idle for a particular job. The 5400 rpm disk requires 4/3 × 60 s, or 80 s to do the same thing. Therefore, it is idle 20% of the time.

1.5 a. $$\frac{14\text{ KW}}{(79\text{ W} + 2.3\text{ W} + 7.0\text{ W})} = 158$$

b. $$\frac{14\text{ KW}}{(79\text{ W} + 2.3\text{ W} + 2 \times 7.0\text{ W})} = 146$$

c. $$\text{MTTF} = \frac{\dfrac{1}{9 \times 10^6} + 8 \times \dfrac{1}{4500} + \dfrac{1}{3 \times 10^4} = \dfrac{8 \times 2000 + 300}{9 \times 10^6} = \dfrac{16301}{9 \times 10^6}}{\dfrac{1}{\text{Failure rate}} = \dfrac{9 \times 10^6}{16301} = 522\text{ hours}}$$

1.6 a. See Figure L.1.

	Sun Fire T2000	IBM x346
SPECjbb	213	91.2
SPECweb	42.4	9.93

Figure L.1 Power/performance ratios.

b. Sun Fire T2000

c. More expensive servers can be more compact, allowing more computers to be stored in the same amount of space. Because real estate is so expensive, this is a huge concern. Also, power may not be the same for both systems. It can cost more to purchase a chip that is optimized for lower power consumption.

1.7 a. 50%

b. $$\frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.50)^2 \times (F \times 0.50)}{V^2 \times F} = 0.5^3 = 0.125$$

c. $$= \frac{.70}{(1-x) + x/2};\ x = 60\%$$

d. $\frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.70)^2 \times (F \times 0.50)}{V^2 \times F} = 0.7^2 \times 0.5 = 0.245$

Case Study 3: The Cost of Reliability (and Failure) in Web Servers

1.8 a. 14 days × \$1.4 million⁄day = \$19.6 million
\$4 billion – \$19.6 million = \$3.98 billion

b. Increase in total revenue: 4.8/3.9 = 1.23

In the fourth quarter, the rough estimate would be a loss of 1.23 × \$19.6 million = \$24.1 million.

c. Losing \$1.4 million × .50 = \$700,000 per day. This pays for \$700,000/\$7,500 = 93 computers per day.

d. It depends on how the 2.6 million visitors are counted.

If the 2.6 million visitors are not unique, but are actually visitors each day summed across a month: 2.6 million × 8.4 = 21.84 million transactions per month. \$5.38 × 21.84 million = \$117 million per month.

If the 2.6 million visitors are assumed to visit every day: 2.6 million × 8.4 × 31 = 677 million transactions per month. \$5.38 × 677 million = \$3.6 billion per month, which is clearly not the case, or else their online service would not make money.

1.9 a. FIT = 10^9⁄MTTF
MTTF = 10^9⁄FIT = 10^9⁄100 = 10,000,000

b. $\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{10^7}{10^7 + 24} = \text{about } 100\%$

1.10 Using the simplifying assumption that all failures are independent, we sum the probability of failure rate of all of the computers:

$\text{Failure rate} = 1000 \times 10^{-7} = 10^{-4} = \frac{10^5}{10^9}\ \text{FIT} = 10^5\text{, therefore MTTF} = \frac{10^9}{10^5} = 10^4$

1.11 a. Assuming that we do not repair the computers, we wait for how long it takes for 3,334 computers to fail.

3,334 × 10,000,000 = 33,340,000,000 hours

b. Total cost of the decision: \$1,000 × 10,000 computers = \$10 million

Expected benefit of the decision: Gain a day of downtime for every 33,340,000,000 hours of uptime. This would save us \$1.4 million each 3,858,000 years. This would definitely not be worth it.

Case Study 4: Performance

1.12 a. See Figure L.2.

Chip	Memory performance	Dhrystone performance
Athlon 64 X2 4800+	1.14	1.36
Pentium EE 840	1.08	1.24
Pentium D 820	1	1
Athlon 64 X2 3800+	0.98	1.13
Pentium 4	0.91	0.5
Athlon 64 3000+	0.98	0.5
Pentium 4 570	1.17	0.74
Widget X	2.33	0.33

Figure L.2 Performance of several processors normalized to the Pentium 4 570.

b. See Figure L.3.

Chip	Arithmetic mean	Arithmetic mean of normalized
Athlon 64 X2 4800+	12070.5	1.25
Pentium EE 840	11060.5	1.16
Pentium D 820	9110	1
Athlon 64 X2 3800+	10035	1.05
Pentium 4	5176	0.95
Athlon 64 3000+	5290.5	0.95
Pentium 4 570	7355.5	0.77
Processor X	6000	1.33

Figure L.3 Arithmetic mean of several processors.

c. The arithmetic mean of the original performance shows that the Athlon 64 X2 4800+ is the fastest processor.

The arithmetic mean of the normalized processors shows that Processor X is the fastest processor.

d. Single processors: .05

Dual processors: 1.17

e. *Solutions will vary.*

f. Dual processors gain in CPU performance (exhibited by the Dhrystone performance), but they do not necessarily increase in memory performance. This makes sense because, although they are doubling the processing power, dual processors do not change the memory hierarchy very much. Benchmarks that exercise the memory often do not fit in the size of the cache, so doubling the cache does not help the memory benchmarks substantially. In some applications, however, they could gain substantially due to the increased cache available.

1.13 a. Pentium 4 570: .4 × 3,501 + .6 × 11,210 = 8,126

Athlon 64 X2 4,800+: .4 × 3,423 + .6 × 20,718 = 13,800

b. 20,718/7,621 = 2.7

c. $x \times 3{,}501 + (1x) \times 11{,}210 = x \times 3{,}000 + (1x) \times 15{,}220$

$x = .89$

.89/.11 = 8x ratio of memory to processor computation

1.14 a. Amdahl's Law:

$$\frac{1}{.6 + .4/2} = 1.25\text{x speedup}$$

b. Amdahl's Law:

$$\frac{1}{.01 + .99/2} = 1.98\text{x speedup}$$

c. Amdahl's Law:

$$\frac{1}{.2 + .8 \times (.6 + .4/2)} = 1.19\text{x speedup}$$

d. Amdahl's Law:

$$\frac{1}{.8 + .2 \times (.01 + .99/2)} = 1.11\text{x speedup}$$

L.2 Chapter 2 Solutions

Case Study 1: Exploring the Impact of Microarchitectural Techniques

2.1 The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 37, as shown in Figure L.4. How did I come up with that number? 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 + 3
        MULTD   F2,F0,F2     1 + 4
        DIVD    F8,F2,F0     1 + 10
        LD      F4,0(Ry)     1 + 3
        ADDD    F4,F0,F4     1 + 2
        ADDD    F10,F8,F2    1 + 2
        SD      F4,0(Ry)     1 + 1
        ADDI    Rx,Rx,#8     1
        ADDI    Ry,Ry,#8     1
        SUB     R20,R4,Rx    1
        BNZ     R20,Loop     1 + 1
                             ____
        cycles per loop iter  37
```

Figure L.4 Baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35.

2.2 How many cycles would the loop body in the code sequence in Figure 2.35 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 27, as shown in Figure L.5. 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 + 3
       <stall>
       <stall>
       <stall>
       MULTD  F2,F0,F2    1 + 4
       <stall>
       <stall>
       <stall>
       <stall>
       DIVD   F8,F2,F0    1 + 10
       LD     F4,0(Ry)    1 + 3
       <stall due to LD latency>
       <stall due to LD latency>
       <stall due to LD latency>
       ADDD   F4,F0,F4    1 + 2
       <stall due to DIVD 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 + 2
       SD     F4,0(Ry)    1 + 1
       ADDI   Rx,Rx,#8    1
       ADDI   Ry,Ry,#8    1
       SUB    R20,R4,Rx   1
       BNZ    R20,Loop    1 + 1
       <stall due to BNZ>
                          ____
       cycles per loop iter  27
```

Figure L.5 Number of cycles required by the loop body in the code sequence in Figure 2.35.

2.3 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 24, as shown in Figure L.6. The `LD` goes first, as before, and the `MULTD` must wait for it through 3 extra latency cycles. After the the `MULTD` comes the `DIVD`, which again has no opportunity to "move up" in the schedule, due to the `DIVD`'s data dependency on the `MULTD`. But there's an `LD` following the `DIVD` that does not depend on the `DIVD`. That `LD` can therefore execute in the 2nd pipe, at the same time as the `DIVD`. 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.

	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>
	MULTD F2,F0,F2	;	<nop>
	<stall for MULTD latency>	;	<nop>
	<stall for MULTD latency>	;	<nop>
	<stall for MULTD latency>	;	<nop>
	<stall for MULTD latency>	;	<nop>
	DIVD F8,F2,F0	;	**LD F4,0(Ry)**
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	ADDD 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	;	**SD F4,0(Ry)**
	ADDI Rx,Rx,#8	;	**ADDI Ry,Ry,#8**
	SUB R20,R4,Rx	;	**BNZ R20,Loop**
	<stall due to BNZ>	;	<nop>
	cycles per loop iter 24		

Figure L.6 Number of cycles required per loop.

2.4 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.

2.5 Figure L.7 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 2.35. The number of cycles that this reordered code takes is 22.

```
         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>
         MULTD   F2,F0,F2             ;   ADDD    F4,F0,F4
         <stall for MULTD latency>    ;   <stall for ADDD latency>
         <stall for MULTD latency>    ;   <stall for ADDD latency>
         <stall for MULTD latency>    ;   SD      F4,0(Ry)
         <stall for MULTD latency>    ;   <nop>
         DIVD    F8,F2,F0             ;   <nop>
         <stall for DIVD latency>     ;   <nop>
         <stall for DIVD latency>     ;   <nop>                   #ops:    11
         <stall for DIVD latency>     ;   <nop>                   #nops:   (22 × 2) – 11 = 33
         <stall for DIVD latency>     ;   <nop>
         <stall for DIVD latency>     ;   <nop>
         <stall for DIVD latency>     ;   <nop>
         <stall for DIVD latency>     ;   <nop>
         <stall for DIVD latency>     ;   <nop>
         ADDI    Rx,Rx,#8             ;   ADDI    Ry,Ry,#8
         SUB     R20,R4,Rx            ;   <nop>
         ADDD    F10,F8,F2            ;   BNZ     R20,Loop
         <stall due to BNZ>           ;   <stall due to BNZ>

         cycles per loop iter 22
```

Figure L.7 Number of cycles taken by reordered code.

2.6 a. Fraction of all cycles, counting both pipes, wasted in the reordered code shown in Figure L.7:

33 wasted out of total of 2 × 22 opportunities

33/44 = 0.75

b. Results of hand-unrolling two iterations of the loop from code shown in Figure L.8:

- Cycles per loop iter: 23
- Each trip through the loop accomplishes two loops worth of algorithm work. So effective cycles per loop is 23/2 = 11.5.
- Speedup = 22/11.5

  = 1.91

	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>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	MULTD	**F2,F0,F2**	;	**ADDD**	**F4,F0,F4**
	MULTD	F2,F0,F2	;	ADDD	F4,F0,F4
	<stall for MULTD latency>		;	<nop>	
	<stall for MULTD latency>		;	**SD**	**F4,0(Ry)**
	<stall for MULTD latency>		;	SD	F4,0(Ry)
	DIVD	**F8,F2,F0**	;	<nop>	
	DIVD	F8,F2,F0	;	<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>	
	<stall due to DIVD latency>		;	<nop>	
	ADDI	**Rx,Rx,#16**	;	**ADDI**	**Ry,Ry,#16**
	SUB	**R20,R4,Rx**	;	<nop>	
	ADDD	**F10,F8,F2**	;	<nop>	
	ADDD	F10,F8,F2	;	**BNZ**	**R20,Loop**
	<stall due to BNZ>		;	<nop>	

Figure L.8 Hand-unrolling two iterations of the loop from code shown in Figure L.7.

2.7 Consider the code sequence in Figure 2.36. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependencies are maintained. Show the resulting code. (*Hint:* See Figure 2.37.) Answer shown in Figure L.9.

`Loop:`	`LD`	`T9,0(Rx)`
`I0:`	`MULTD`	`T10,F0,T9`
`I1:`	`DIVD`	`T11,F0,T9`
`I2:`	`LD`	`T12,0(Ry)`
`I3:`	`ADDD`	`T13,F0,T12`
`I4:`	`ADDD`	`T14,T11,T9`
`I5:`	`SD`	`T15,0(Ry)`

Figure L.9 Register renaming.

2.8 The rename table has arbitrary values at clock cycle *N* – 1. Look at the next two instructions (`I0` and `I1`): `I0` targets the F5 register, and `I1` will write the F9 register. This means that in clock cycle *N,* the rename table will have had its entries 5 and 9 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 12 and 13 come along; 12 will overwrite F5, and 13 will write F2. This means the rename table's entry 5 gets written again, this time to 11 (the next available T reg), and rename table entry 2 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.

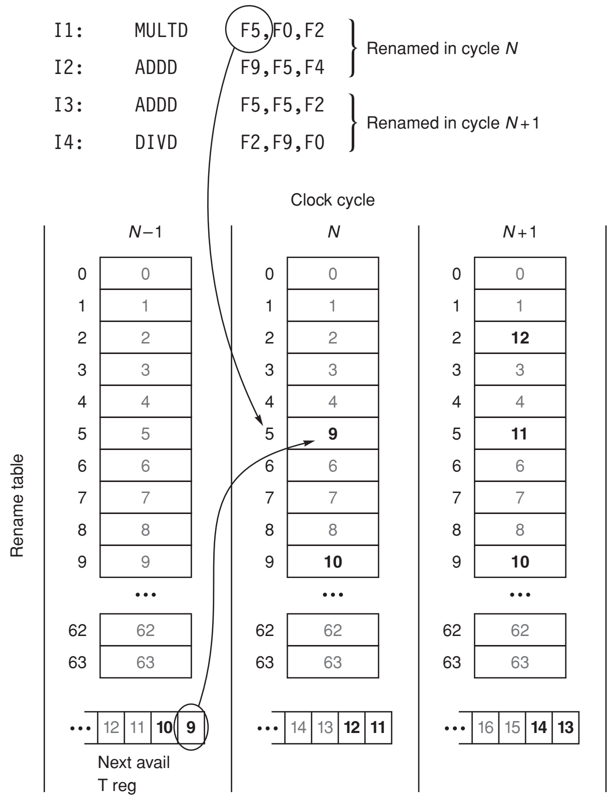


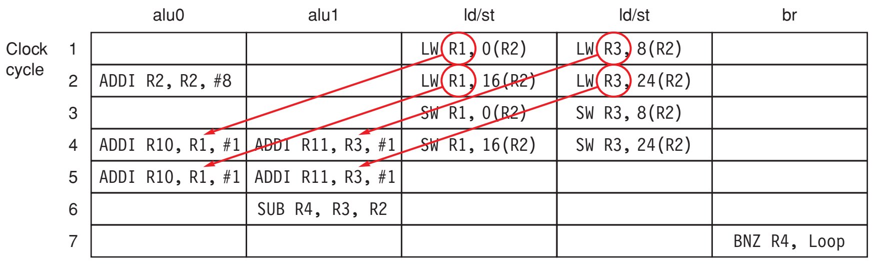
Figure L.10 Cycle-by-cycle state of the rename table for every instruction of the code in Figure 2.38.

2.9 The value of R1 when the sequence has been executed is shown in Figure L.11.

`ADD`	`R1, R1, R1;`	`5 + 5 -> 10`
`ADD`	`R1, R1, R1;`	`10 + 10 -> 20`
`ADD`	`R1, R1, R1;`	`20 + 20 -> 40`

Figure L.11 Value of R1 when the sequence has been executed.

2.10 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 L.12.



Clock cycle	alu0	alu1	ld/st	ld/st	br
1			`LW R1, 0(R2)`	`LW R3, 8(R2)`	
2	`ADDI R2, R2, #8`		`LW R1, 16(R2)`	`LW R3, 24(R2)`	
3			`SW R1, 0(R2)`	`SW R3, 8(R2)`	
4	`ADDI R10, R1, #1`	`ADDI R11, R3, #1`	`SW R1, 16(R2)`	`SW R3, 24(R2)`	
5	`ADDI R10, R1, #1`	`ADDI R11, R3, #1`			
6		`SUB R4, R3, R2`			
7					`BNZ R4, Loop`

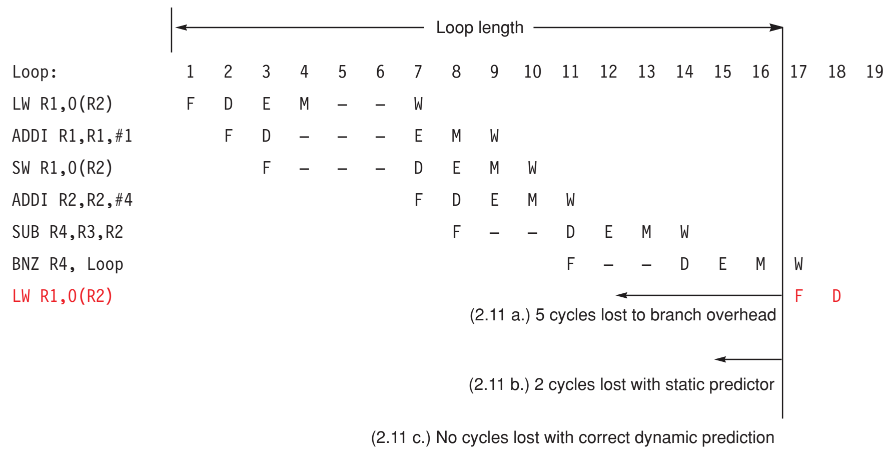
Figure L.12 Example of an event that yields wrong results. What could go wrong with this? If an interrupt is taken between clock cycles 1 and 4, then the results of the `LW` at cycle 2 will end up in `R1`, instead of the `LW` at cycle 1. Bank stalls and ECC stalls will cause the same effect—pipes will drain, and the last writer wins, a classic WAW hazard. All other "intermediate" results are lost.

2.11 The convention is that an instruction does not enter the execution phase until all of its operands are ready. So the first instruction, `LW R1,0(R2)`, 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 that `LD` is available at the execution unit, any subsequent instructions (especially that `ADDI R1, R1, #1`) cannot enter the `E` stage, and must therefore stall at the `D` stage.

a. 5 cycles are 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 5 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 are lost with 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 still losing 2 clock cycles here.

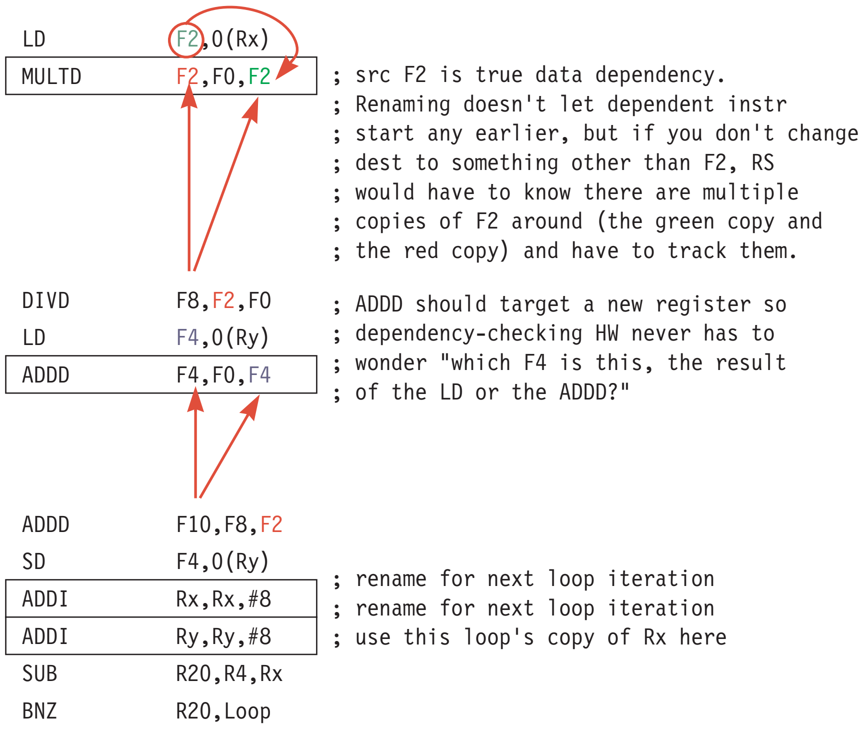
c. No cycles are lost with correct dynamic prediction. A dynamic branch predictor remembers that when the branch instruction was fetched in the past, it eventually turned out to be a branch, and this branch was taken. So a "predicted taken" will occur in the same cycle as the branch is fetched, and the next fetch after that will be to the presumed target. If correct, we've saved all of the latency cyclles seen in 2.11 (a) and 2.11 (b). If not, we have some cleaning up to do.


Loop length
(2.11 a.) 5 cycles lost to branch overhead
(2.11 b.) 2 cycles lost with static predictor
(2.11 c.) No cycles lost with correct dynamic prediction

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

Figure L.13 Phases of each instruction per clock cycle for one iteration of the loop.

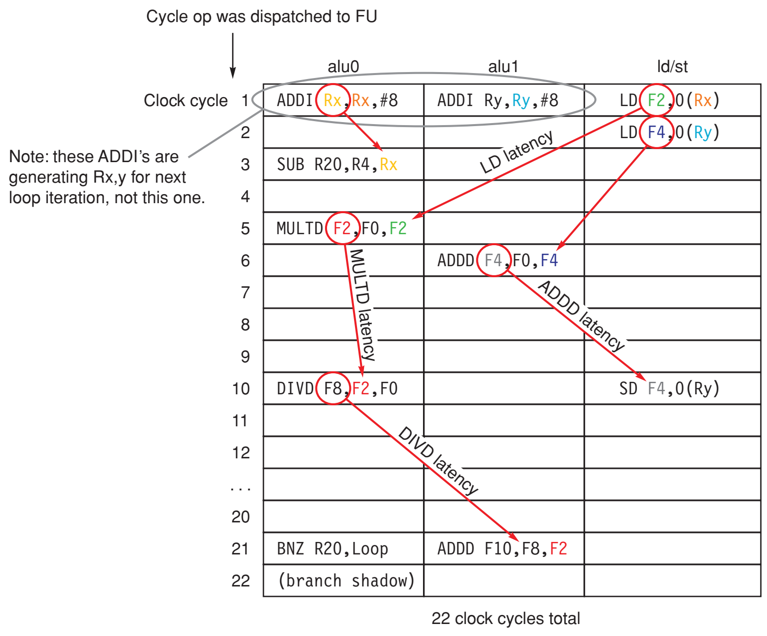
2.12 a. Instruction in code where register renaming improves persormance are shown in Figure L.14.



```
LD      F2,0(Rx)
MULTD   F2,F0,F2     ; src F2 is true data dependency.
                     ; Renaming doesn't let dependent instr
                     ; start any earlier, but if you don't change
                     ; dest to something other than F2, RS
                     ; would have to know there are multiple
                     ; copies of F2 around (the green copy and
                     ; the red copy) and have to track them.
DIVD    F8,F2,F0     ; ADDD should target a new register so
LD      F4,0(Ry)     ; dependency-checking HW never has to
ADDD    F4,F0,F4     ; wonder "which F4 is this, the result
                     ; of the LD or the ADDD?"
ADDD    F10,F8,F2
SD      F4,0(Ry)
ADDI    Rx,Rx,#8     ; rename for next loop iteration
ADDI    Ry,Ry,#8     ; rename for next loop iteration
SUB     R20,R4,Rx    ; use this loop's copy of Rx here
BNZ     R20,Loop
```

Figure L.14 Instructions in code where register renaming improves performance.

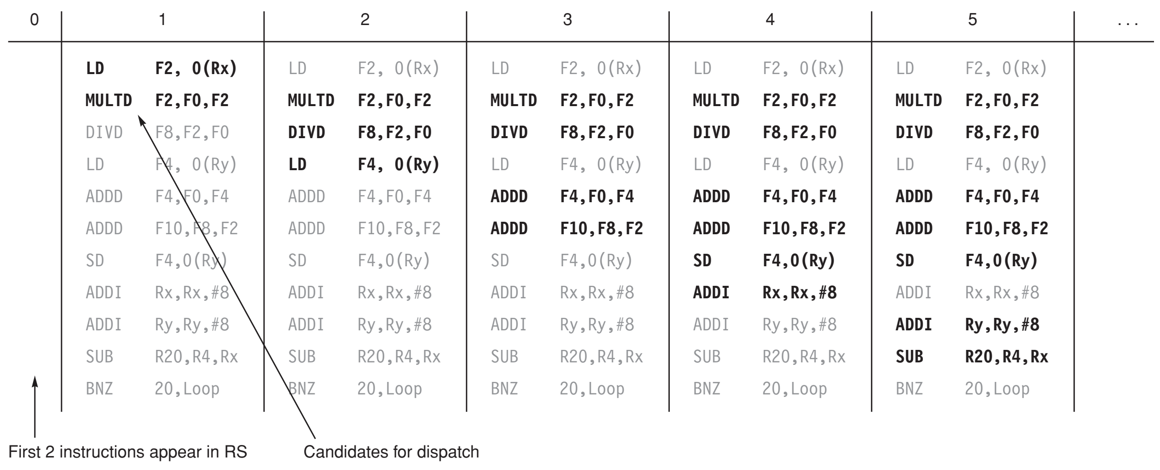
b. Number of clock cycles taken by the code sequence is shown in Figure L.15.


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
MULTD latency
ADDD latency
DIVD latency
22 clock cycles total

Clock cycle	alu0	alu1	ld/st
1	`ADDI Rx,Rx,#8`	`ADDI Ry,Ry,#8`	`LD F2,0(Rx)`
2			`LD F4,0(Ry)`
3	`SUB R20,R4,Rx`		
4			
5	`MULTD F2,F0,F2`		
6		`ADDD F4,F0,F4`	
7			
8			
9			
10	`DIVD F8,F2,F0`		`SD F4,0(Ry)`
11			
12			
. . .			
20			
21	`BNZ R20,Loop`	`ADDD F10,F8,F2`	
22	`(branch shadow)`		

Figure L.15 Number of clock cycles taken by the code sequence.

c. See Figures L.16 and L.17 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	. . .
	LD F2, 0(Rx)	LD F2, 0(Rx)	LD F2, 0(Rx)	LD F2, 0(Rx)	LD F2, 0(Rx)	
	MULTD F2,F0,F2	**MULTD F2,F0,F2**	**MULTD F2,F0,F2**	**MULTD F2,F0,F2**	**MULTD F2,F0,F2**	
	DIVD F8,F2,F0	**DIVD F8,F2,F0**	**DIVD F8,F2,F0**	**DIVD F8,F2,F0**	**DIVD F8,F2,F0**	
	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 F10,F8,F2	ADDD F10,F8,F2	**ADDD F10,F8,F2**	**ADDD F10,F8,F2**	**ADDD F10,F8,F2**	
	SD F4,0(Ry)	SD F4,0(Ry)	SD F4,0(Ry)	**SD F4,0(Ry)**	**SD F4,0(Ry)**	
	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**	
	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	

Figure L.16 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	MULTD F2,F0,F2	ADDI Ry,Ry,#8	
6	SUB R20,R4,Rx	ADDD F4,F0,F4	
7			
8			
9			
10	DIVD F8,F2,F0		SD F4,0(Ry)
11			
12			
. . .			
20			
21	BNZ R20,Loop	ADDD F10,F8,F2	
22	(Branch shadow)		

Figure L.17 Number of clock cycles required. 22 clock cycles total: lucked out; second LD became visible in time to execute in cycle 2.

d. See Figure L.18.

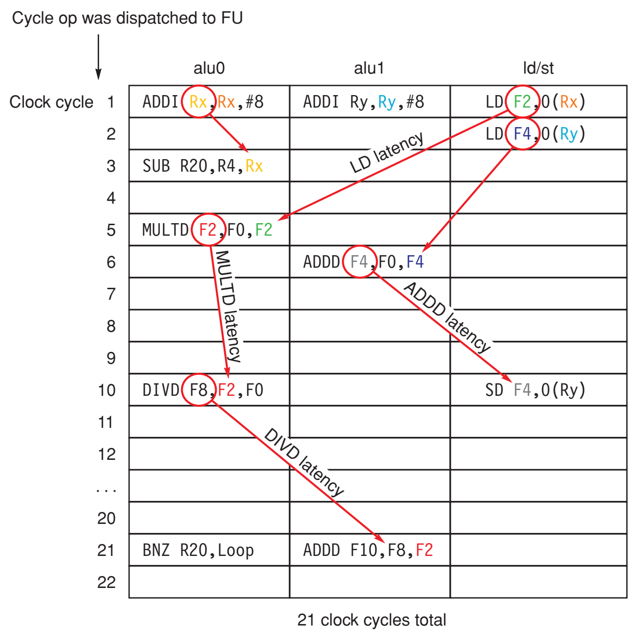
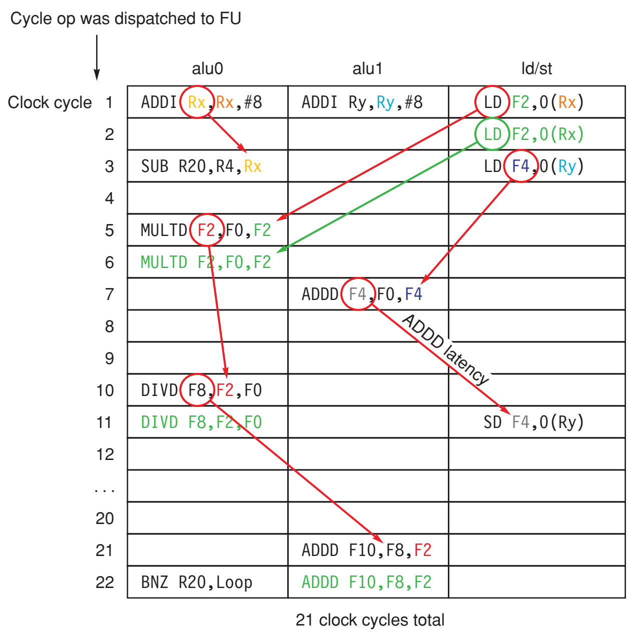


Figure L.18 Speedup is (execution time without enhancement) / (execution time with enhancement) = 21 / (21 – 5) = 1.31.

1. Another ALU: 0% improvement.
2. Another LD/ST unit: 0% improvement.
3. Full bypassing: critical path is `LD -> Mult -> Div -> ADDD`. Bypassing would save 1 cycle from latency of each, so 2 cycles total.
4. Cutting longest latency in half: divider is longest at 10 cycles. This would save 5 cycles total.

e. See Figure L.19.



Clock cycle	alu0	alu1	ld/st
1	`ADDI Rx,Rx,#8`	`ADDI Ry,Ry,#8`	`LD F2,0(Rx)`
2			`LD F2,0(Rx)`
3	`SUB R20,R4,Rx`		`LD F4,0(Ry)`
4			
5	`MULTD F2,F0,F2`		
6	`MULTD F2,F0,F2`		
7		`ADDD F4,F0,F4`	
8			
9			
10	`DIVD F8,F2,F0`		
11	`DIVD F8,F2,F0`		`SD F4,0(Ry)`
12			
. . .			
20			
21		`ADDD F10,F8,F2`	
22	`BNZ R20,Loop`	`ADDD F10,F8,F2`	

Figure L.19 Number of clock cycles required to do two loops' worth of work. Critical path is `LD -> MULTD -> DIVD -> ADDD`. If RS schedules second 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, no extra clock cycle is needed.

Case Study 2: Modeling a Branch Predictor

2.13 For this exercise, please refer to the Case Study Resource Files directory on the companion CD. The resources for Case Study 2 include a `php_out` text file, which is the expected output of the *C* program the reader is asked to write for this exercise.

L.3 Chapter 3 Solutions

Case Study: Dependences and Instruction-Level Parallelism

3.1 a. Figure L.20 shows the dependence graph for the C code in Figure 3.14. Each node in Figure L.20 corresponds to a line of C statement in Figure 3.14. Note that each node 6 in Figure L.20 starts an iteration of the `for` loop in Figure 3.14.

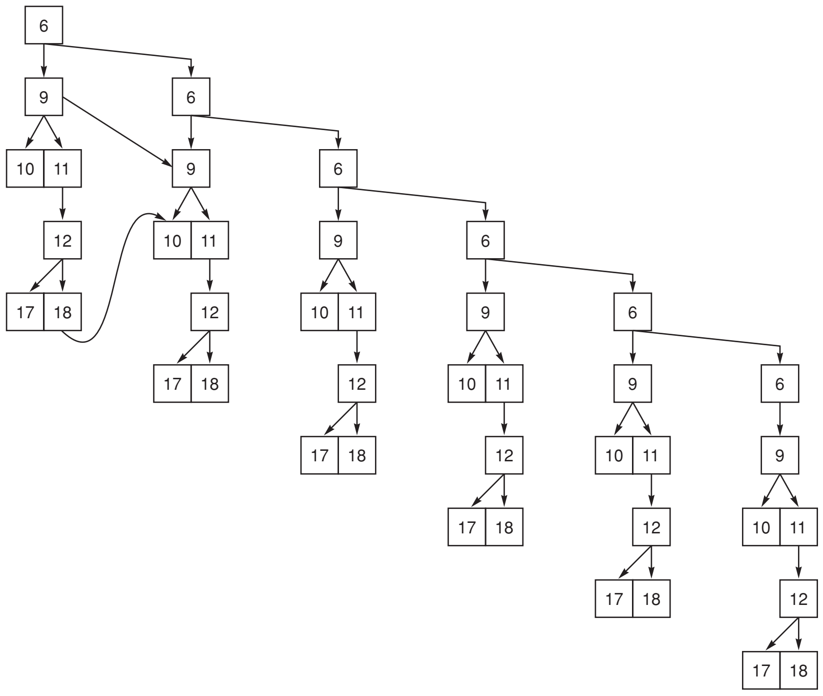


Figure L.20 Dynamic dependence graph for six insertions under the ideal case.

Since we are assuming that each line in Figure 3.14 corresponds to one machine instruction, Figure L.20 can be viewed as the instruction-level dependence graph. A data true dependence exists between line 6 and line 9. Line 6 increments the value of `i`, and line 9 uses the value of `i` to index into the element array. This is shown as an arc from node 6 to node 9 in Figure L.20. Line 9 of Figure 3.14 calculates the `hash_index` value that is used by lines 10 and 11 to index into the element array, causing true dependences from line 9 to line 10 and line 11. This is reflected by arcs going from node 9 to node 10 and node 11 in Figure L.20. Line 11 in Figure 3.14 initializes ptrCurr, which is used by line 12. This is reflected as a true dependence arc from node 11 to node 12 in Figure L.20.

Note that node 15 and node 16 are not reflected in Figure L.20. Recall that all buckets are initially empty and each element is being inserted into a different bucket. Therefore, the while loop body is never entered in the ideal case.

Line 12 of Figure 3.14 enforces a control dependence over line 17 and line 18. The execution of line 17 and line 18 hinges upon the test that skips the while loop body. This is shown as control dependence arcs from node 12 to node 17 and node 18.

There is a data output dependence from Line 9 of one iteration to Line 9 of the next iteration. This is due to the fact that both dynamic instructions need to write into the same variable hash_index. For simplicity, we omitted the data output dependence from Line 10 of one iteration to itself in the next iteration due to variable ptrUpdate as well as the dependence from Line 11 of one iteration to itself in the next iteration due to variable ptrCurr.

There is a data antidependence from Line 17 of one iteration to Line 10 of the next iteration. This is due to the fact that Line 17 needs to read from variable ptrUpdate before Line 10 of the next iteration overwrites its contents. The reader should verify that there are also data anti--dependences from Lines 10 and 11 of one iteration to Line 9 of the next iteration, from Line 18 to Line 10 of the next iteration, and from Line 12 to Line 11 for the next iteration.

Note that we have also omitted some data true dependence arcs from Figure L.20. For example, there should be a true dependence arc from node 10 to node 17 and node 18. This is because line 10 of Figure 3.14 initializes ptrUpdate, which is used by lines 17 and 18. These dependences, however, do not impose any more constraints than what is already imposed by the control dependence arcs from node 12 to node 17 and node 18. Therefore, we omitted these dependence arcs from Figure L.20 in favor of simplicity. The reader should identify any other omitted data true dependences from Figure L.20.

In the ideal case, all for loop iterations are independent of each other once the for loop header (node 6) generates the i value needed for the iteration. Node 6 of one iteration generates the i value needed by the next iteration. This is reflected by the dependence arc going from node 6 of one iteration to node 6 of the next iteration. There are no other dependence arcs going from any node in a for loop iteration to subsequent iterations. This is because each for loop iteration is working on a different bucket. The changes made by line 18 (×ptrUpdate=) to the pointers in each bucket will not affect the insertion of data into other buckets. This allows for a great deal of parallelism.

Recall that we assume that each statement in Figure 3.14 corresponds to one machine instruction and takes 1 clock cycle to execute. This makes the latency of nodes in Figure L.20 1 cycle each. Therefore, each horizontal row of Figure L.20 represents the instructions that are ready to execute at a clock cycle. The width of the graph at any given point corresponds to the amount of instruction-level parallelism available during that clock cycle.

b. As shown in Figure L.20, each iteration of the outer `for` loop has 7 instructions. It iterates 1024 times. Thus, 7168 instructions are executed.

   The `for` loop takes 4 cycles to enter steady state. After that, one iteration is completed every clock cycle. Thus the loop takes 4 + 1024 = 1028 cycles to execute.

c. 7168 instructions are executed in 1028 cycles. The average level of ILP available is 7168/1028 = 6.973 instructions per cycle.

d. See Figure L.21. Note that the cross-iteration dependence on the `i` value calculation can easily be removed by unrolling the loop. For example, one can unroll the loop once and change the usage of the array index usage of the

```
6     for (i = 0; i < N_ELEMENTS; i+=2)
        {
7         Element *ptrCurr, **ptrUpdate;
8         int hash_index;

          /* Find the location at which the new element is to be inserted. */
9         hash index = element[i].value & 1023;
10        ptrUpdate = &bucket[hash_index];
11        ptrCurr = bucket[hash_index];
          /* Find the place in the chain to insert the new element. */
12        while (ptrCurr &&
13               ptrCurr->value <= element[i].value)
14          {
15              ptrUpdate = &ptrCurr->next;
16            ptrCurr = ptrCurr->next;
            }

          /* Update pointers to insert the new element into the chain. */
17        element[i].next = *ptrUpdate;
18        *ptrUpdate = &element[i];

9′        hash_index = element[i+1].value & 1023;
10′       ptrUpdate = $bucket[hash_index];
11′       ptrCurr = bucket[hash_index];
12        while (ptrCurr &&
13               ptrCurr->value <= element[i+1].value)
14          {
15              ptrUpdate = &$ptrCurr->next;
16            ptrCurr = ptrCurr->next;
            }

          /* Update pointers to insert the new element into the chain. */
17        element[i+1].next = *ptrUpdate;
18        *ptrUpdate = &$element[i+1];
        }
```

Figure L.21 Hash table code example.

unrolled iteration to `element[i+1]`. Note that the two resulting parts of the `for` loop body after unrolling transformation are completely independent of each other. This doubles the amount of parallelism available. The amount of parallelism is proportional to the number of unrolls performed. Basically, with the ideal case, a compiler can easily transform the code to expose a very large amount of parallelism.

e. Figure L.22 shows the time frame in which each of these variables needs to occupy a register. The first iteration requires 4 registers. The reader should be able to tell some variables can occupy a register that is no longer needed by another variable. For example, the `hash_index` of iteration 2 can occupy the same register occupied by the `hash_index` of iteration 1. Therefore, the overlapped execution of the next iteration uses only 2 additional registers.

Similarly the third and the fourth iteration each requires another one register. Each additional iteration requires another register. By the time the fifth iteration starts execution, it does not add any more register usage since the register for `i` value in the first iteration is no longer needed. As long as the hardware has no fewer than 8 registers, the parallelism shown in Figure 3.15 can be fully realized. However, if the hardware provides fewer than 8 registers, one or more of the iterations will need to be delayed until some of the registers are freed up. This would result in a reduced amount of parallelism.

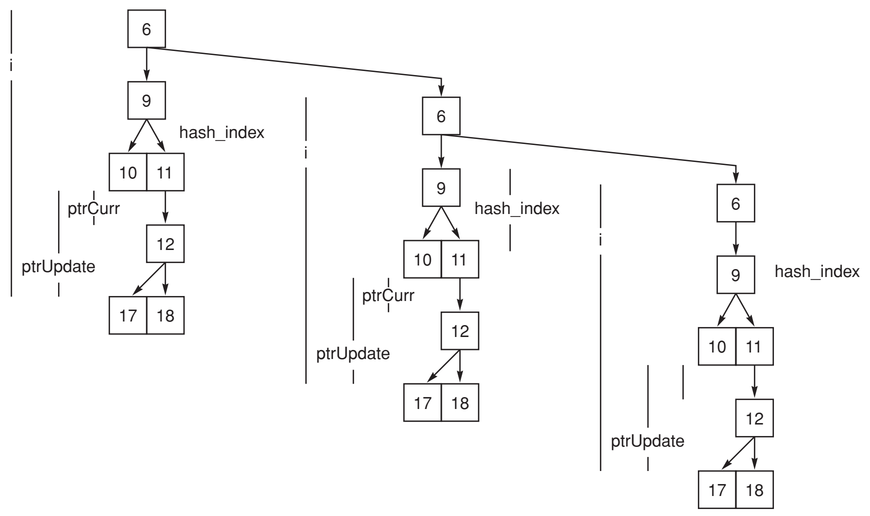


Figure L.22 Register lifetime graph for the ideal case.

f. See Figure L.23. Each iteration of the `for` loop has 7 instructions. In a processor with an issue rate of 3 instructions per cycle, it takes about 2 cycles for the processor to issue one iteration. Thus, the earliest time the instructions in the second iteration can be even considered for execution is 2 cycles after the first iteration. This delays the start of the next iteration by 1 clock cycle.

Figure L.24 shows the instruction issue timing of the 3-issue processor. Note that the limited issue rate causes iterations 2 and 3 to be delayed by 1 clock cycle. It causes iteration 4, however, to be delayed by 2 clock cycles. This is a repeating pattern.

Cycle			
1	6	9	10
2	11	12	17
3	18	6	9
4	10	11	12
5	17	18	6
6	9	10	11
7	12	17	18
8	6	9	10

Figure L.23 Instruction issue timing.

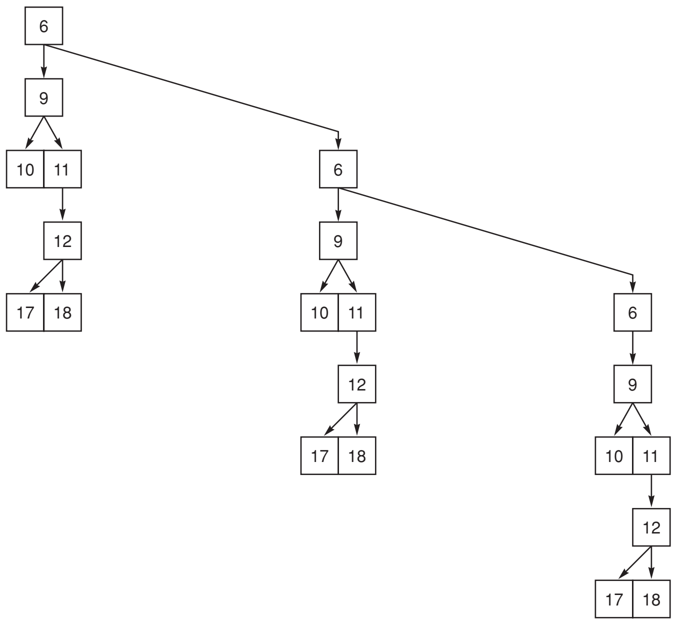


Figure L.24 Execution timing due to limited issue rate for the ideal case.

The reduction of parallelism due to limited instruction issue rate can be calculated based on the number of clock cycles needed to execute the `for` loop. Since the number of instructions in the `for` loop remains the same, any increase in execution cycle results in decreased parallelism. It takes 5 cycles for the first iteration to complete. After that, one iteration completes at cycles 7, 9, 12, 14, 16, 19, ... Thus the `for` loop completes in 5 + 2 × 645 + 3 × 342 = 5 + 1290 + 684 = 1979 cycles. When compared to part (b), limiting the issue rate to 3 instructions per cycle reduces the amount of parallelism to about half!

g. In order to achieve the level of parallelism shown in Figure 3.15, we must assume that the instruction window is large enough to hold instructions 17, 18 of the first iteration, instructions 12, 17, 18 of the second iteration, instructions 10, 11, 12, 17, and 18 of the third iteration as well as instructions 9, 10, 11, 12, 17, 18 of the second iteration when instruction 6 of the third iteration is considered for execution. If the instruction window is not large enough, the processor would be stalled before instruction 6 of the third iteration can be considered for execution. This would increase the number of clocks required to execute the `for` loop, thus reducing the parallelism. The minimal instruction window size for the maximal ILP is thus 17 instructions. Note that this is a small number. Part of the reason is that we picked a scenario where there is no dependence across `for` loop iterations and that there is no other realistic resource constraints. The reader should verify that, with more realistic execution constraints, much larger instruction windows will be needed in order to support available ILP.

3.2 a Refer to Figures 3.14 and 3.15. The insertions of data elements 0 and 1 are still parallel. The insertion of 1024, on the other hand, is made into the same bucket as element 0. The execution of lines 17 and 18 for element 0 can affect the execution of lines 11, 12, 13, 15, and 16 for element 1024. That is, the new element 0 inserted into bucket 0 will affect the execution of the linked list traversal when inserting element 1024 into the same bucket. In Figure L.25, we show a dependence arc going from node 18 of the first iteration to node 11 of the third iteration. These dependences did not exist in the ideal case shown in Figure L.20. We show only the dependence from node 18 of one iteration to node 11 of a subsequent iteration for simplicity. There are similar dependence arcs to nodes 12, 13, 15, 16, but they do not add to more constraints than the ones we are showing. The reader is nevertheless encouraged to draw all the remaining dependence arcs for a complete study. Note that these new dependences skip iteration 2, thus still allowing substantial instruction-level parallelism.

b. The number of instructions executed for each `for` loop iteration increases as the numbers of data elements hashed into bucket 0 and bucket 1 increase. Iterations 1 and 2 have 7 instructions each. Iterations 3 and 4 have 11 instructions each. The reader should be able to verify that iterations 5 and 6 of the `for` loop have 15 instructions each. Each successive pair of iterations will

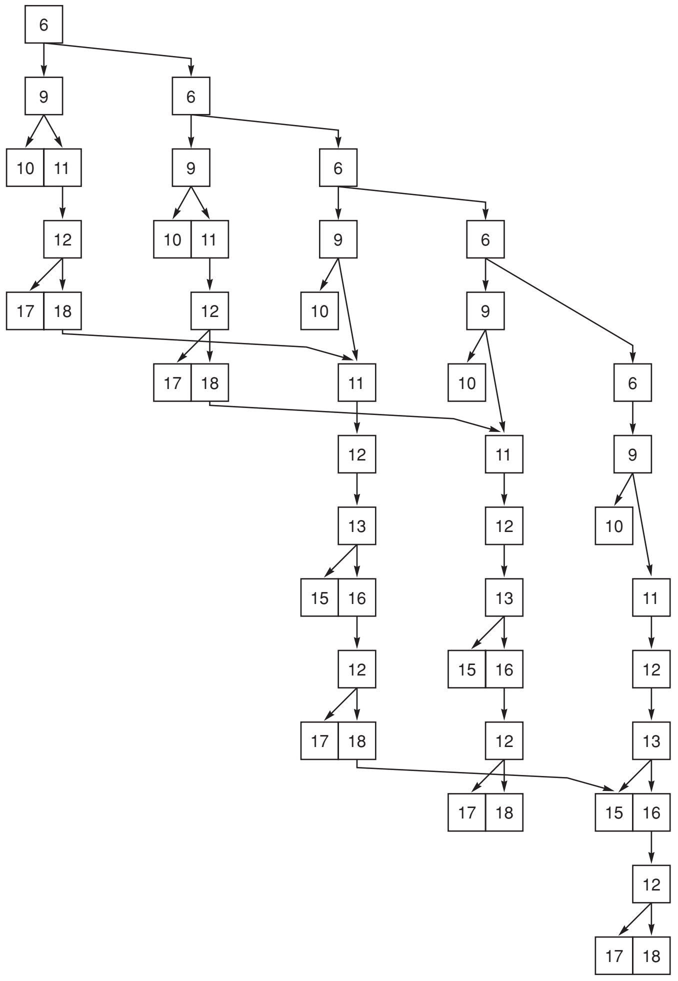


Figure L.25 Dynamic dependence graph of the hash table code when the data element values are 0, 1, 1024, 1025, 2048, 2049, 3072, 3073, ...

have 4 more instructions each than the previous pair. This will continue to iterations 1023 and 1024. The total number of instructions can be expressed as the following series:

$$\begin{aligned} & 2 \times (7 + 11 + 15 + \ldots + (7 + (N - 1) \times 4)) \\ &= 2 \times N/2 \times (14 + (N - 1) \times 4)) \\ &= \mathrm{N} \times (4 \times N + 10) \\ &= 4N^2 + 10N \end{aligned}$$

where N is the number of pairs of data elements.

In our case, N is 512, so the total number of dynamic instructions executed is $4 \times 2^{18} + 10 \times 2^9 = 1{,}048{,}576 + 5120 = 1{,}053{,}696$. Note that there are many more instructions executed when the data elements are hashed into two buckets, rather than evenly into the 1024 buckets. This is why it is important to design the hash functions so that the data elements are hashed evenly into all the buckets.

c. The number of clock cycles required to execute all the instructions can be derived by examining the critical path in Figure L.25. The critical path goes from node 6 of iteration 1 to nodes 6, 9, 11, 12, 18 of iteration 2, and then to nodes 11, 12, 13, 16, 12, 18 of iteration 3. Note that the critical path length contributed by each iteration forms the following series:

$$1, 5, 3 + 3, 3 + 2 \times 3, \ldots 3 + (N - 1) \times 3$$

The total length of the critical path is the sum of contributions

$$\begin{aligned} & 1 + 5 + (3 + 3) + (3 + 2 \times 3) + (3 + 3 \times 3) + (3 + (N - 1) \times 3) \\ = {} & 6 + 6 + ((N - 2)/2) \times (6 + (N + 1) \times 3) \end{aligned}$$

where N is the number of pairs of data elements.

In our case, N is 512, so the total number of clock cycles for executing all the instructions in the dynamic dependence graph is

$$\begin{aligned} & 12 + 255 \times (6 + (513) \times 3) \\ = {} & 12 + 255 \times (1545) \\ = {} & 12 + 393{,}975 \\ = {} & 393{,}987 \end{aligned}$$

d. The amount of instruction-level parallelism available is the total number of instructions executed divided by the critical path length. The answer is

$$\begin{aligned} & 1{,}053{,}696/393{,}987 \\ = {} & 2.67 \end{aligned}$$

Note that the level of instruction-level parallelism has been reduced from 6.973 in the ideal case to 2.67. This is due to the additional dependences you observed in part (a). There is an interesting double penalty when elements are hashed into the same bucket: the total number of instructions executed increases and the amount of instruction-level parallelism decreases. In a processor, these double penalties will likely interact and reduce the performance of the hash table code much more than the programmer expected. This demonstrates a phenomenon in parallel processing machines. The algorithm designer often needs to pay attention not only to the effect of algorithms on the total number of instructions executed but also to their effect on the parallelism available to the hardware.

e. In the worst case, all new data elements are entered into the same bucket and they come in ascending order. One such sequence would be 0, 1024, 2048, 3072, 4096, . . . The level of serialization depends on the accuracy of the memory disambiguation mechanism. In the worst case, the linked-list traversal of an iteration will not be able to start until the linked-list updates of all previous iterations are complete. Such serialization will essentially eliminate any overlap between the `while` loop portions across `for` loop iterations. Note also that this sequence will also cause more instructions to be executed.

f. With perfect memory disambiguation, node 18 for element 0 still affect the execution of node 11 for element 1024. However, after the insertion of element 0 is complete, node 18 of one `for` loop iteration will only affect the execution of node 16 of the penultimate `while` loop iteration of the next `for` loop iteration. This can greatly increase the overlap of successive `for` loop iterations. Also, the number of critical path clock cycles contributed by each `for` loop iteration becomes constant: 6 cycles (node 16 → 12 → 13 → 16 → 12 → 18). This makes the total clock cycles as determined by the critical path to be:

$$5 + 6 \times 1023 = 6143 \text{ cycles}$$

Speculation allows much more substantial overlap of loop iterations and confers a degree of immunity to increased memory latency, as subsequent operations can be initiated while previous stores remain to be resolved. In cases such as this, where the addresses of stores depend on the traversal of large data structures, this can have a substantial effect. Speculation always brings with it the cost of maintaining speculative information and the risk of misspeculation (with the cost of recovery). These are factors that should not be ignored in selecting an appropriate speculation strategy.

g. The total number of instructions executed is

$$\begin{aligned} & 7 + 11 + \ldots (7 + 4 \times (N - 1)) \\ = \; & (N/2) \times (14 + 4 \times (N - 1)) \\ = \; & N \times (7 + 2 \times (N - 1)) \end{aligned}$$

where N is the number of data elements.

In our case, N is 1024. The total number of instructions is 2,102,272.

h. The level of instruction-level parallelism is

$$2{,}102{,}272/6143 = 342.2$$

Beyond perfect memory disambiguation, the keys to achieving such a high level of instruction-level parallelism are (1) a large instruction window and (2) perfect branch prediction.

i. The key to achieving the level of instruction-level parallelism in part (h) is to overlap the `while` loop execution of one `for` loop iteration with those of many subsequent `for` loop iterations. This requires that the instruction windows be large enough to contain instructions from many `for` loop iterations. The number of instructions in each for-loop iteration increases very quickly as the number of elements increase. For example, by the time the last element is inserted into the hash table, the number of instructions in the `for` loop iteration will reach

$$\begin{aligned} & 7 + 4 \times (N - 1) \qquad \text{where } N \text{ is } 1024 \\ = \; & 4099 \end{aligned}$$

Since each iteration of the `for` loop provides only a small amount of parallelism, it is natural to conclude that many `for` loop iterations must overlap in order to achieve instruction-level parallelism of 342.2. Any instruction window less than 4099 will likely cut down the instruction-level parallelism to less than, say, 10 instructions per cycle.

j. The exit branch of the `while` loop will likely cause branch prediction misses since the number of iterations taken by the `while` loop changes with every `for` loop iteration. Each such branch prediction miss disrupts the overlapped execution across `for` loop iterations. This means that the execution must reenter the steady state after the branch prediction miss is handled. It will introduce at least three extra cycles into total execution time, thus reducing the average level of ILP available. Assuming that the mispredictions will happen to all `for` loop iterations, they will essentially bring the level of instruction-level parallelism back down to that of a single `for` loop iteration, which will be somewhere around 1.5.

   Aggressive but inaccurate branch prediction can lead to the initiation of many instruction executions that will be squashed when misprediction is detected. This can reduce the efficiency of execution, which has implications for power consumed and for total performance in a multithreaded environment. Sometimes the off-path instructions can initiate useful memory subsystem operations early, resulting in a small performance improvement.

k. A static data dependence graph is constructed with nodes representing static instructions and arcs representing control flows and data dependences. Figure L.26 shows a simplified static data dependence graph for the hash table code. The heavy arcs represent the control flows that correspond to the iteration

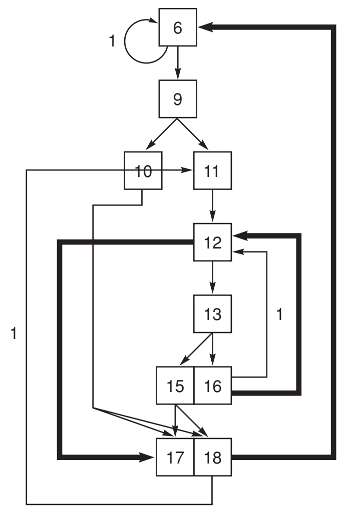


Figure L.26 (Simplified) static dependence graph of the hash table code with control flow and worst-case dependence arcs.

control of the `while` loop and the `for` loop. These loop control flows allow the compiler to represent a very large number of dynamic instructions with a small number of static instructions.

The worst-case dependences from one iteration to a future iteration are expressed as dependence arcs marked with "dependence distance," a value indicating that the dependences go across one or more iterations. For example, there is a dependence from node 6 of one iteration of the `for` loop to itself in the next iteration. This is expressed with a "1" value on the dependence arc to indicate that the dependence goes from one iteration to the next.

The memory dependence from node 18 of one iteration of the `for` loop to node 11 of a future iteration is shown as an arc of dependence distance 1. This is conservative since the dependence may not be on the immediate next iteration. This, however, gives the worst-case constraint so that the compiler will constrain its actions to be on the safe side.

The static dependence graph shown in Figure L.26 is simplified in that it does not contain all the dependence arcs. Those dependence arcs that do not impose any further scheduling constraints than the ones shown are omitted for clarity. One such example is the arc from node 10 to nodes 17 and 18. The dependence does not impose any additional scheduling constraints than the arcs 11 → 12 → 13 → 16 do. They are shown as dashed lines for illustration purposes. The reader should compare the static dependence graph in Figure L.26 with the worst-case dynamic dependence graph since they should capture the same constraints.

L.4 Chapter 4 Solutions

Case Study 1: Simple, Bus-Based Multiprocessor

4.1 a. P0.B0: (S, 120, 00, 20), read returns 20

b. P0.B0: (M, 120, 00, 80), P15.B0: (I, 120, 00, 20)

c. P15.B0: (M, 120, 00, 80)

d. P0.B2: (S, 110, 00, 30), P1.B2: (S, 110, 00, 30), M[110]: (00, 30), read returns 30

e. P0.B1: (M, 108, 00, 48), P15.B1: (I, 108, 00, 08)

f. P0.B2: (M, 130, 00, 78), M[110]: (00, 30)

g. P0.B2: (M, 130, 00, 78)

4.2 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 + 70 + 10 + 100 + 10 = 290 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

4.3 See Figure L.27.

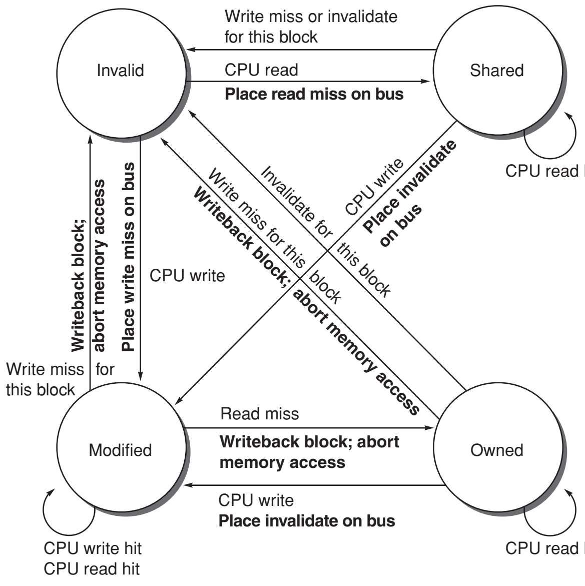

Invalid
Shared
Modified
Owned
Write miss or invalidate for this block
CPU read
Place read miss on bus
CPU read hit
Invalidate for this block
Write miss for this block
Writeback block; abort memory access
CPU write
Place invalidate on bus
Writeback block; abort memory access
Place write miss on bus
CPU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

Figure L.27 Protocol diagram.

4.4 a. `P1: read 110` Read miss, P0's cache
`P15: read 110` Read miss, MSI satisfies in memory, MOSI satisfies in P0's cache
`P0: read 110` Read hit
MSI: 70 + 10 + 100 + 0 = 180 stall cycles
MOSI: 70 + 10 + 70 + 10 + 0 = 160 stall cycles

b. `P1: read 120`Read miss, satisfied in memory
`P15: 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 P15
`P15: read 120` Read miss, P0's cache
`P0: read 120` Read hit
Both protocols: 100 + 70 + 10 + 0 = 180 stall cycles

d. `P0: write 108 <-- 88` Send invalidate, invalidate P15
`P15: read 108` Read miss, P0's cache
`P0: write 108 <-- 98` Send invalidate, invalidate P15
Both protocols: 15 + 70 + 10 + 15 = 110 stall cycles

4.5 See Figure L.28.

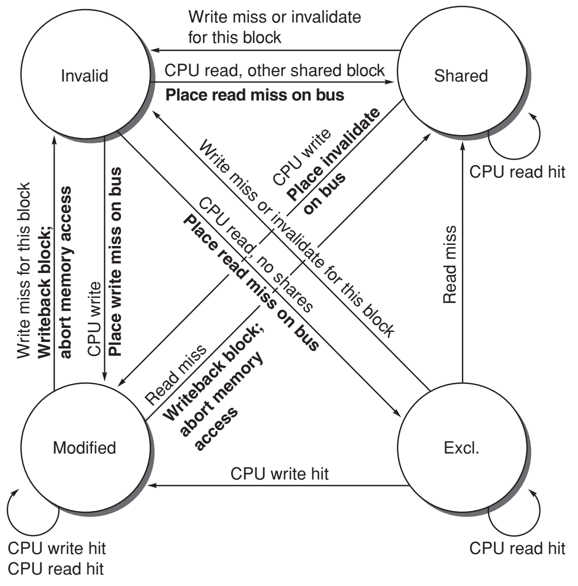


Figure L.28 Diagram for a MESI protocol.

This version of the MESI protocol does *not* supply data in the E state. Some versions do.

4.6 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 + 70 + 10 = 195 stall cycles
MESI: 100 + 0 + 70 + 10 = 180 stall cycles

4.7 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 P15 will stall until P0's critical section ends (ping-ponging the block back and forth) 1000 cycles later. P0 will stall for (about) 70 cycles while it fetches the block to invalidate it; then P1 takes 70 cycles to acquire it. P1's critical section is 1000 cycles, plus 70 to handle the write miss at release. Finally, P15 grabs the block for a final 70 cycles of stall. So, P0 stalls for 100 cycles to acquire, 10 to give it to P1, 70 to release the lock, and a final 10 to hand it off to P1, for a total of 190 stall cycles. P1 essentially stalls until P0 releases the lock, which will be 100 + 1000 + 10 + 70 = 1180 cycles, plus 70 to get the lock, 10 to give it to P15, 70 to get it back to release the lock, and a final 10 to hand it back to P15. This is a total of 1340 stall cycles. P15 stalls until P1 hands it off the released lock, which will be 1180 + 70 + 10 + 1000 + 70 = 2270 cycles. Finally, P15 gets the lock 70 cycles later, so it stalls a total of 2330 cycles.

b. Test-and-test-and-set will have many fewer stall cycles than test-and-set 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 bus transactions, the stall cycles will be 3 read memory misses (300), 1 upgrade (15) and 1 write miss to a cache (70 + 10) and 1 write miss to memory (100), 1 read cache miss to cache (70 + 10), 1 write miss to memory (100), 1 read miss to cache and 1 read miss to memory (70 + 10 + 100), followed by an upgrade (15) and a write miss to cache (70 + 10), and finally a write miss to cache (70 + 10) followed by a read miss to cache (70 + 10) and an upgrade (15). So approximately 1125 cycles total.

c. Approximately 19 bus transactions. The first processor to win arbitration for the bus 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 70 cycles, this will occur about 14 times (14). 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 bus 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).

Case Study 2: A Snooping Protocol for a Switched Network

4.8 a. `P0: read 120` Read miss, service in memory
P0: I --> IS^{AD} --> IS^{D} --> S

b. `P0: write 120 <-- 80` Write miss, service in memory, invalidates P15
P0: I --> IM^{AD} --> IM^{D} --> M
P15: S --> I

c. `P15: write 120 <-- 80` Write to S block, send invalidate
P15: S --> SM^{A} --> M

d. `P1: read 110` Read to I block, serviced from P0's cache
P1: I --> IS^{AD} --> IS^{D} --> S
P0: M --> S

e. `P0: write 108 <-- 48` Write to S block, send invalidate, invalidate P15
P0: S --> SM^{A} --> M
P15: S --> I

f. `P0: write 130 <-- 78` Replace block in M, and write miss to block, serviced in memory
P0.110: M --> MI^{A} --> I
P0.130: I --> IM^{AD} --> IM^{D} --> M

g. `P15: write 130 <-- 78` Write miss, serviced in memory
P15: I --> IM^{AD} --> IM^{D} --> M

4.9 a. `P0: read 120` Read miss, service in memory
`P1: read 120` Read miss, service in memory
P0: I --> IS^{AD} --> IS^{D} --> S
P1: I --> IS^{AD} --> IS^{D} --> S

b. `P0: read 120` Read miss, service in memory
`P1: write 120 <-- 80` Read miss, service in memory, invalidate P0
P0: I --> IS^{AD} --> IS^{D} --> S --> I (note that P0 stalls the address network until the data arrives since it cannot respond yet)
P1: I --> IM^{AD} --> IM^{D} --> M

c. `P0: write 120` <-- 80 Write miss, service in memory
`P1: read 120` Read miss, service in P0's cacheP0: I --> IM^{AD} --> IM^{D} --> M --> S (note that P0 stalls the address network until the data arrives since it cannot respond yet)
P1: I --> IS^{AD} --> IS^{D} --> S

d. `P0: write 120 <-- 80` Write miss, service in memory
`P1: write 120 <-- 90` Write miss, service in P0's cache
P0: I --> IM^{AD} --> IM^{D} --> M --> I (note that P0 stalls the address network until the data arrives since it cannot respond yet)
P1: I --> IM^{AD} --> IM^{D} --> M

e. `P0: replace 110` Write back M block
`P1: read 110` Read miss, serviced in memory since P0 wins the race
P0: M --> MI^{A} --> I
P1: I --> IS^{AD} --> IS^{D} --> S

f. `P1: write 110 <-- 80` Write miss, service in P0's cache
`P0: replace 110` Write back M block aborted since P1's GetM wins the race
P1: I --> IM^{AD} --> IM^{D} --> M
P0: M --> MI^{A} --> II^{A} --> I

g. `P1: read 110` Read miss, service in P0's cache
`P0: replace 110` Write back M block aborted since P1's GetM wins the race
P1: I --> IS^{AD} --> IS^{D} --> S
P0: M --> MI^{A} --> II^{A} --> I

4.10 a. `P0: read 120` Read miss, service in memory
P0.latency: Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 = 157
P0.occupancy: Osend_req + Orcv_data = 1 + 4 = 5
Mem.occupancy: Oread_memory = 20

b. `P0: write 120 <-- 80` Write miss, service in memory
P0.latency: Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 = 157
P0.occupancy: Osend_req + Orcv_data = 1 + 4 = 5
Mem.occupancy: Oread_memory = 20

c. `P15: write 120 <-- 80` Write to S block, send invalidate
P15.latency: Lsend_req
4 cycles
P15.occupancy: Osend_req = 1

d. `P1: read 110` Read miss, service in P0's cache
P1.latency: Lsend_req + Lreq_msg + Lsend_data + Ldata_msg + Lrcv_data = 4 + 8 + 20 + 30 + 15 = 78
P1.occupancy: Osend_req + Orcv_data = 1 + 4 = 5
P0.occupancy: Osend_data = 4

e. `P0: read 120` Read miss, service in memory
`P15: read 128` Read miss, service in P1's cache, serialize at address network
P0.latency: Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 = 157
P0.occupancy: Osend_req + Orcv_data = 1 + 4 = 5
Mem.occupancy: Oread_memory = 20
P15.latency: Oreq_msg + Lsend_req + Lreq_msg + Lsend_data + Ldata_msg + Lrcv_data = 1 + 4 + 8 + 20 + 30 + 15 = 77
P15.occupancy: Osend_req + Orcv_data = 1 + 4 = 5
P1.occupancy: Osend_data = 4

f. `P0: read 100` Read miss, service in memory
`P1: write 110 <-- 78` Write miss, service in P0's cache
P0.latency: Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 =157
P0.occupancy: Osend_req + Osend_data + Orcv_data = 1 + 4 + 4 = 9
Mem.occupancy: Oread_memory = 20
P1.latency: Oreq_msg + Lsend_req + Lreg_msg + Lsend_data + Ldata_msg + Lrev_data = 1 + 4 + 8 + 20 + 30 + 15 = 78
P1.occupancy: Osend_req + Orev_data = 1 + 4 = 5

g. `P0: write 100 <-- 28` Write miss, service in memory
`P1: write 100 <-- 48` Write miss, service in P0's cache (when it gets it)
P0.latency: Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 = 157
P0.occupancy: Osend_req + Orcv_data + Osend_data = 1 + 4 + 4 = 9
Mem.occupancy: Oread_memory = 20
P1.latency: (Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data) + Lsend_data + Ldata_msg + Lrcv_data = 157 + 20 + 30 + 15 = 222
P1.occupancy: Osend_req + Orcv_data = 1 + 4 = 5

4.11 a. See Figure L.29.

State	GetS	GetM	Data
S	send Data/S	send Data/M	err
M	-/MS^D	—	save Data/MS^A
MS^A	-/S	—	err
MS^D	z	z	save Data/S

Figure L.29 Table to specify the memory controller protocol.

b.

`P1: read 110`	Read miss, P0 responds with data to P1 and memory
`P15: read 110`	Read miss, satisfied from memory

P1's read miss will force P0 to respond with data both to P1 and to memory and then transition to state S. The memory must receive the data and respond to all future requests. Because the address network and data network are not synchronized with each other, the memory may receive P1's GetS message or P0's Data message in either order. If it receives the data first, then it transitions to MS^A and waits for P1's GetS to complete the transition to S. Thus P15's GetS will find the block in state S in memory. The trickier case, which actually occurs much more frequently, occurs when P1's GetS message arrives before the Data. This causes the memory to transition to MS^D, to await the data. If P15's message also arrives before the data, the memory controller cannot respond (since it doesn't have the data). Instead, it must block the request until the data arrives (denoted by the 'z' in the protocol).

4.12 See Figure L.30.

State	Read	Write	Replace-ment	OwnReq	Other GetS	Other GetM	Other Inv	Other PutM	Data
I	send GetS/ IS^{AD}	send GetM/ IM^{AD}	error	error	—	—	—	—	error
S	do Read	send Inv/ SM^{A}	I	error	—	I	I	—	error
O	do Read	send Inv OM^{A}	send PutM/MI^{A}	error	send Data/O	send Data/I	I	—	error
M	do Read	do Write	send PutM/MI^{A}	error	send Data/S	send Data/I	—	—	error
IS^{AD}	z	z	z	IS^{D}	—	—	—	—	save Data /IS^{A}
IM^{AD}	z	z	z	IM^{D}	—	—	—	—	save Data/IM^{A}
IS^{A}	z	z	z	do Read/S	—	—	—	—	error
IM^{A}	z	z	z	do Write/M	—	—	—	—	error
SM^{A}	z	z	z	M	—	II^{A}	II^{A}	—	error
OM^{A}	z	z	z	M	send Data/ OM^{A}	send Data/II^{A}	IIA	—	error
MI^{A}	z	z	z	send Data/I	send Data/ MI^{A}	send Data/II^{A}	—	—	error
II^{A}	z	z	z	I	—	—	—	—	error
IS^{D}	z	z	z	error	—	z	—	—	save Data, do Read/S
IM^{D}	z	z	z	error	z	—	—	—	save Data, do Write/M

Figure L.30 Solution to Exercise 4.12.

4.13 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 simple bus-based system, this is typically done by having two wired-or signals, Shared and Owned, that are driven by the caches and monitored by caches and memory. If a node makes a request (e.g., GetS), the memory monitors the Owned signal to see if it should respond or not. Similarly, if no node asserts Shared, the requesting node knows that no other caches have copies, and it can set the state to E. In a switched protocol, the pipelined nature of the interconnect makes it more difficult (but not impossible) to implement the Shared and Owned.

4.14 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 (157 stall cycles)

`P0: read 108`Hit, but must wait for preceding operation: TSO = 0, SC = 157

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 (157 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 (157 stall cycles)

`P0: write 110 <-- 90` Hit, but must wait for preceding operation: TSO = 0, SC = 157

4.15 a. The default protocol handles a write miss in Lsend_req + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 4 + 8 + 100 + 30 + 15 = 157 stall cycles. The optimization allows the write to be retired immediately after the node sees its own request message: Lsend_req + Lreq_msg = 4 + 8 = 12 stall cycles. Since all memory stalls directly stall an in-order core, this is an important optimization.

b. The same analysis holds for the out-of-order core. The difference is that the out-of-order core has better latency tolerance, so the impact will be reduced. Nonetheless, even an out-of-order core has trouble hiding long memory latencies.

Case Study 3: Simple Directory-Based Coherence

4.16 a. `P0: read 100`
P0.B0: (S, 100, 00 00)
M.100: (DS, {P0}, 00 00)

b. `P0: read 128`
P0.B1: (S, 128, 00 68)
P1.B1: (S, 128, 00 68)
M.128: (DS, {P0,P1}, 00 68)

c. `P0: write 128 <-- 78`
P0.B1: (M, 128, 00 78)
P1.B1: (I, 128, 00 68)
M.128: (DM, {P0}, 00 20)
Memory is not updated on a write to a block that is M in another cache.

d. `P0: read 120`
P0.B0: (S, 120, 00 20)
M.120: (DS, {P0,P15}, 00 20)

e. `P0: read 120`
`P1: read 120`
P0.B0: (S, 120, 00 20)
P1.B0: (S, 120, 00 20)
M.120: (DS, {P0,P1,P15}, 00 20)

f. `P0: read 120`
`P1: write 120 <-- 80`
P0.B0: (I, 120, 00 20)
P1.B0: (M, 120, 00 80)
P15.B0: (I, 120, 00 20)
M.120: (DM, {P1}, 00 20)

g. `P0: write 120 <-- 80`
`P1: read 120`
P0.B0: (S, 120, 00 80)
P1.B0: (S, 120, 00 80)
P15.B0: (I, 120, 00 20)
M.120: (DS, {P0,P1}, 00 20)

h. `P0: write 120 <-- 80`
`P1: write 120 <-- 90`
P0.B0: (I, 120, 00 80)
P1.B0: (M, 120, 00 90)
P15.B0: (I, 120, 00 20)
M.120: (DM, {P1}, 00 20)
Memory is not updated on a write to a block that is M in another cache.

4.17	a.	`P0: write 100 <-- 80`	No messages, hits in P0's cache
	b.	`P0: write 108 <-- 88`	Send invalidate to P15
	c.	`P0: write 118 <-- 90`	Send invalidate to P1
	d.	`P1: write 128 <-- 98`	Send fetch/invalidate to P1

4.18 See Figures L.31 and L.32.

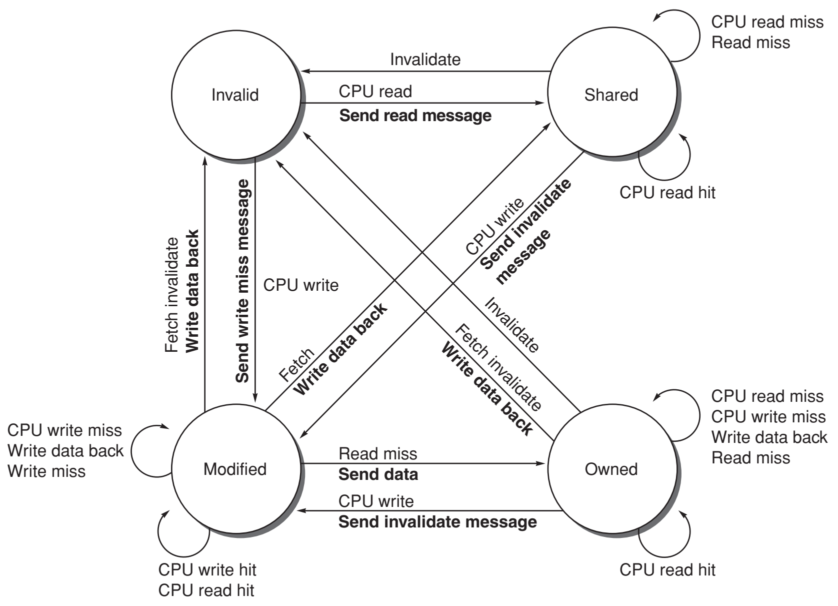


Figure L.31 Cache states.

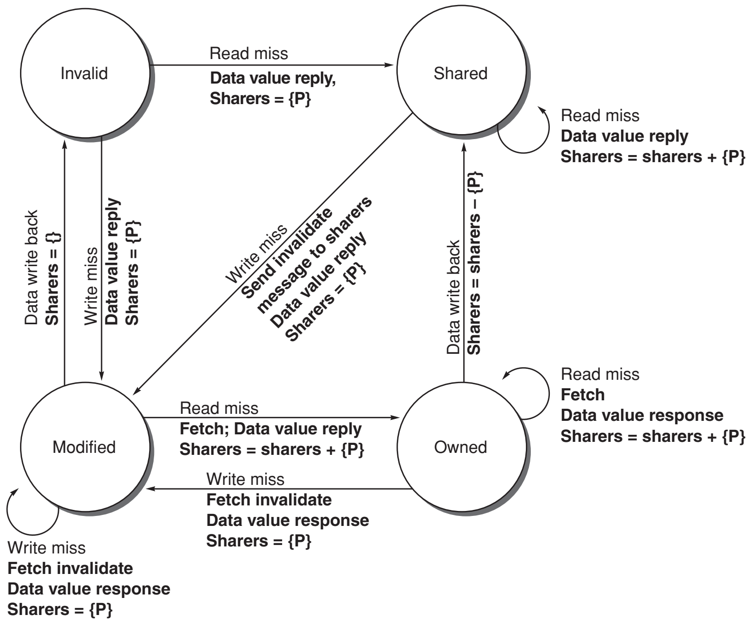


Figure L.32 Directory states.

4.19 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.

Case Study 4: Advanced Directory Protocol

4.20 a. `P0: read 100`
P0: I --> IS^D --> S Dir: DI --> DS {P0}

b. `P0: read 120`
P0: I --> IS^D --> S Dir: DS {P15} --> DS {P0,P15}

c. `P0: write 120 <-- 80`
P0: I --> IM^{AD} --> IM^{A} --> M
P15: S --> I Dir: DS {P15} --> DM {P0}

d. `P15: write 120 <-- 80`
P15: S --> IM^{AD} --> IM^{A} --> M
Dir: DS {P15} --> DM {P15}

e. `P1: read 110`
P1: I --> IS^{D} --> S
P0: M --> MS^{A} --> S
Dir: DM {P0} --> DS {P0,P1}

f. `P0: write 108 <-- 48`
P0: S --> IM^{AD} --> IM^{A} --> M
P15: S --> I
Dir: DS {P0,P15} --> DM {P0}

4.21 a. `P0: read 120`
`P1: read 120`
P0: I --> IS^{D} --> S
P1: I --> IS^{D} --> S
Dir: DS {P15} --> DS {P0,P15} --> DS {P0,P1,P15}

b. `P0: read 120`
`P1: write 120 <-- 80`
P0: I --> IS^{D} --> S --> I
P1: I --> IM^{AD} --> IM^{A} --> M
P15: S --> I
Dir: DS {P15} --> DS {P0,P15} --> DM {P1}

c. `P0: write 120`
`P1: read 120`
P0: I --> IM^{AD} --> IM^{A} --> M --> MS^{A} --> S
P1: I --> IS^{D} --> ISI^{D} --> I (GetM arrives at directory first, so INV arrives in IS^{D})
P15: S --> I
Dir: DS {P15} --> DM {P0} --> DMS^{D} {P0},{P1} --> DS {P0,P1}

d. `P0: write 120 <-- 80`
`P1: write 120 <-- 90`
P0: I --> IM^{AD} --> IM^{A} --> M --> I
P1: I --> IM^{AD} --> IM^{A} --> M
P15: S --> Dir: DS {P15} --> DM {P0} --> DM {P1}

e. `P0: replace 110`
`P1: read 110`
P0: M --> MI^{A} --> P1: I --> IS^{D} --> S
Dir: DM {P0} --> DI --> DS {P1}

f. `P1: write 110 <-- 80`
`P0: replace 110`
P0: M --> MI^A --> P1: I --> IM^{AD} --> IM^A --> Dir: DM {P0} --> DM {P1}

g. `P1: read 110`
`P0: replace 110`
P0: M --> MI^A --> MI^A --> P1: I --> IM^{AD} --> IM^A --> Dir: DM {P0} --> DMS^D{P0},{P1} --> DS {P1}

4.22 a. `P0: read 100` Miss, satisfied in memory
P0.latency: Lsend_msg + Lreq_msg + Lread_memory + Ldata_msg + Lrcv_data = 6 + 15 + 100 + 30 + 15 = 166

b. `P0: read 128` Miss, satisfied in P1's cache
P0.latency: Lsend_msg + Lreq_msg + Lsend_msg + Lreq_msg + Lsend_data + Ldata_msg + Lrcv_data = 6 + 15 + 6 + 15 + 20 + 30 + 15 = 107

c. `P0: write 128 <-- 68` Miss, satisfied in P1's cache
P0.latency: Lsend_msg + Lreq_msg + Lsend_msg + Lreq_msg + Lsend_data + Ldata_msg + Lrcv_data = 6 + 15 + 6 + 15 + 20 + 30 + 15 = 107

d. `P0: write 120 <-- 50` Miss, invalidate P15's cache
P0.latency: Lsend_msg + Lreq_msg + Lread_memory + max(Linv + Lreq_msg + Lsend_msg + Lreq_msg, Ldata_msg + Lrcv_data) + Lack = 6 + 15 + 100 + max(1 + 15 + 6 + 15, 30 + 15) + 4 = 125 + max(37,45) = 170

e. `P0: write 108 <-- 80` Write to S block, downgrade to I and send GetM
P0.latency: Lsend_msg + Lreq_msg + Lread_memory + max(Linv + Lreq_msg + Lsend_msg + Lreq_msg, Ldata_msg + Lrcv_data) + Lack = 6 + 15 + 100 + max(1 + 15 + 6 + 15, 30 + 15) + 4 = 125 + max(37,45) = 170

4.23 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.

4.24 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 L.33 and L.34.

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 L.33 Directory protocol cache controller transitions.

State	GetS	GetM	PutM (owner)	PutM (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	save 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	save Data, send PutM_Ack/DI	send PutM_Ack

Figure L.34 Directory controller transitions.

4.25 a.
```
P1: read 110
P15: write 110 <-- 90
```
In this problem, both P1 and P15 miss and send requests that race to the directory. Assuming that P1's GetS request arrives first, the directory will forward P1's GetS to P0, followed shortly afterwards by P15's GetM. If the network maintains point-to-point order, then P0 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: read 110
P0: replace 110
```
P1's GetS arrives at the directory and is forwarded to P0 before P0's PutM message arrives at the directory and sends the PutM_Ack. However, if the PutM_Ack arrives at P0 out of order (i.e., before the forwarded GetS), then this will cause P0 to transition to state I. In this case, the forwarded GetS will be treated as an error condition.

L.5 Chapter 5 Solutions

Case Study 1: Optimizing Cache Performance via Simple Hardware

5.1 a. The 2-way cache has an access time of 0.91 ns, the 4-way has an access time of 0.90 ns, and the direct-mapped is 0.59 ns. (Access times can vary depending on how the cache is broken down into subarrays by CACTI, so sometimes the results are counterintuitive.) Thus the 2-way takes 54% more time than the direct-mapped, while the 4-way takes 53% more time than the direct-mapped cache.

b. The access time increases from 0.89 ns to 0.91 ns to 0.96 ns when going from 16 KB to 32 KB to 64 KB, so the relative access times are 2.2% greater for the 32 KB and 7.8% greater for the 64 KB.

c. Looking at 2-way set-associative caches with 32 byte blocks, an 8 KB cache has an access time of 0.74 ns, while a 64 KB cache has an access time of 1.06 ns. The larger cache is 8 times larger but has a 43% larger access time. This growth is between square root and logarithmic, but closer to log.

d. The current version of CACTI states that 16 KB 8-way set-associative caches with 64 byte blocks have an access time of 0.88 ns. This has the lowest miss rate for 16 KB caches, except for fully associative caches, which would have an access time greater than 0.90 ns.

5.2 a. The average memory access time of the current cache is 0.0056 × 20 + (1 – 0.0056) = 1.11 cycles. The Average Memory Access Time (AMAT) of the way-predicted cache has three components: miss, hit with way prediction correct, and hit with way prediction mispredict: 0.0056 × 20 + (0.85 × 1 + (1 – 0.85) × 2) × (1 – 0.0056) = 1.26.

b. The access time of the 32 KB 2-way cache is 0.91 ns, while the 16 KB direct-mapped cache can be accessed in 0.56 ns. This provides 0.91/0.56 = 1.63, or 63% faster execution.

c. With a 1-cycle way misprediction penalty, AMAT is 1.26—as per part (a), but with a 15-cycle misprediction penalty, the AMAT becomes 0.0056 × 20 + (0.85 × 1 + (1 – 0.85) × 15) × (1 – 0.0056) = 3.19, for an increase of 3.19 – 1.26 = 1.93!

d. The serial access is 4.81 ns/3.53 ns = 1.36, or 36% slower, but dissipates 0.150 nJ/0.191 nJ = 0.79, or 21% less power.

5.3 a. The access time is 0.959 ns, while the cycle time is 0.371 ns, which could be potentially pipelined as finely as 0.959/0.371 = 2.58 pipe stages.

b. The baseline AMAT with deeper pipelining becomes 0.00367 × 40 + (1 – 0.00367) × 1 = 1.14. Adding another cycle 20% of the time would make it 1.34.

c. There are two banks. Assuming a random distribution of addresses and a steady stream of accesses, each access has a 50% probability of conflicting with the previous access. The miss rate is the same as a 64 KB 2-way cache, so the AMAT is 0.00367 × 20 + (0.50 × 1 + 0.50 × 2) × (1 – 0.00367) = 1.57 cycles.

5.4 a. With critical word first, the miss service would require 100 cycles. Without critical word first, it would require 100 cycles for the first 16 bytes and 16 cycles for each of the next 3 16-byte blocks, or 100 + (3 × 16) = 148 cycles.

b. It depends on the contribution to 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.

5.5 a. 16 bytes, to match the level 2 data cache write path.

b. Assume merging write buffer entries are 16 bytes wide. Since each store can write 4 bytes, a merging write buffer entry would fill up in 4 cycles. This is the write speed of the level 2 cache. In contrast, a nonmerging write buffer would take 4 cycles to write the 4 byte result of each store. This means the merging write buffer would be 4 times faster.

Case Study 2: Optimizing Cache Performance via Advanced Techniques

5.6 a. Each element is 8 bytes. The input and output blocks split the 16 KB cache, so each can utilize 8 KB, or 1024 elements. Thus each block should be 32 × 32 = 1024 elements.

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

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

5.7 The unblocked version does not fit in the cache, so each cache block of the row supplying 64 bytes/8 = 8 elements needs to be fetched once, while each column element needs to be fetched once. This requires 9 prefetches for processing 8 elements. Each inner loop would require the maximum of 9 prefetches times 2 cycles = 18 cycles, and 8 loop iterations times 2 cycles for operations = 16 cycles. The prefetches are the rate-limiting step, so the number of iterations per cycle would be 18/8 = 2.25.

5.8 a. A sequential stream buffer would not help with fetching the column entries for the matrix transposition. The round-robin allocation policy combined with only two stream buffers would also mean that useful data prefetched in response to row prefetches would be replaced before it could be used by useless prefetches of sequential blocks after column misses. So the performance would be the same as without prefetching.

b. A sequential stream buffer would not help with fetching the column entries for the matrix transposition. The round-robin allocation policy combined with only two stream buffers would also mean that useful data prefetched in response to row prefetches would be replaced before it could be used by useless prefetches of sequential blocks after column misses. So 0% are useful.

5.9 a. *Solutions will vary.*

b. *Solutions will vary.*

Case Study 3: Main Memory Technology and Optimizations

5.10 a. A 1 GB DRAM with parity or ECC effectively has 9-bit bytes and would require 18 512 Mb DRAMs. To create 72 output bits, each one would have to output 72/18 = 4 bits.

b. A burst length of 8 reads out 32 bytes.

c. The DDR2-667 DIMM peak bandwidth ratio is 667/533 = 1.25, or 25% for reads to active pages.

d. This is the scenario given in the figure. This requires 12 cycles of a 266 MHz clock, or 12 × (1/266 MHz) = 45 ns.

e. The latency to an already active bank is 8 clock cycles, versus 12 if a bank activate is required. The access requiring a bank activate is 12/8 = 1.5, or 50% longer.

5.11 The costs of the two systems are \$130 + \$800 = \$930 with the DDR2-667 DIMM and \$100 + \$800 = \$900 with the DDR2-533 DIMM. The latency to service a level 2 miss is 14 × (1/333 MHz) = 42 ns with the DDR2-667 DIMM and 12 × (1/266 MHz) = 45 ns with the DDR-533 DIMM. The CPI added by the level 2 misses in the case of DDR2-667 is 0.00333 × 42 = 0.140, giving a total of 1.5 + 0.140 = 1.64. Meanwhile the CPI added by the level 2 misses for DDR2-533 is 0.00333 × 45 = 0.150, giving a total of 1.5 + 0.150 = 1.65. Thus the difference in performance is only 1.65/1.64 = 1.006, or 0.6%, while the cost is \$930/\$900 = 1.033, or 3.3% greater. The cost/performance of the DDR2-667 system is 1.64 × 930 = 1525, while the cost/performance of the DDR2-533 system is 1.65 × 900 = 1485, so the DDR2-533 system is a better value.

5.12 The cores will be executing 8 cores × 3 GHz/2.0 CPI = 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 × 32 bytes = 2560 MB/sec.

5.13 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.

Case Study 4: Virtual Machines

5.14 a. Yes. Each application could be run on a virtual machine on the new CMP-based server, providing the illusion it was running on its own machine with the original operating system. Thus higher-performance hardware could host several applications simultaneously on a single server.

b. Yes. If an application was running on a virtual machine, viruses, worms, and spyware should only affect the virtual machine running the application and not the entire host machine.

c. No. VMs do not generally provide higher performance, and applications with large memory usage may have many TLB faults, which incur virtualization overhead.

d. Yes. New virtual machines running the application can be relatively easily created on machines not normally used for that application in order to provide extra capacity.

e. Yes. Old versions of the operating system can be hosted on VMs running on modern machines.

5.15 a. Programs with large amounts of I/O, programs with large working sets (and hence many TLB faults or paging), and other programs with a large number of system calls.

b. The slowdown above was 60% for 10%, so 30% system time would run 180% slower.

c. For pure virtualization the mean slowdown is 13.57, while for para virtualization the mean slowdown is 5.27.

d. Fork and exec sh has the lowest overhead. They do more work than many of the other system calls, so the overheads are relatively less.

5.16 The virtual machine running on top of another virtual machine would experience greater slowdowns than one running directly on a host.

5.17 a. Both VT-x and SVM add a mode that intercepts the operation of privilege-sensitive instructions and allows the correct operation to be performed.

b. AMD's SVM provides additional support for virtualization of virtual memory, but VT-x (as of the date of the computer paper) does not yet have this support.

Case Study 5: Putting It All Together: Highly Parallel Memory Systems

5.18 a. There are two levels of cache.

b. The first-level cache is 32 KB and has 64-byte blocks.

c. The miss penalty is approximately 4.8 ns.

d. The first-level cache is 4-way set associative.

e. The large miss penalty to go off-chip coupled with the approximate LRU replacement policy causes some misses when a perfect LRU policy could hold the data in the cache.

5.19 a. *Hint:* Walk through memory using stores and see if you can make writes pile up in a write buffer to see if it is write-through or write-back.

b. *Hint:* Instead of walking through memory using linked lists, directly walk through memory so multiple independent memory references can be generated.

c. *Hint:* Try to generate as many outstanding memory references as possible, using independent accesses.

5.20 a. *Hint:* First characterize the upper levels of the memory system with one copy of the program so you can learn how to best generate a high number of independent misses.

b. *Hint:* After characterizing the size and associativity of the last level of on-chip cache, generate a large number of independent misses for the last level of on-chip caching.

5.21 *Solutions will vary.*

L.6 Chapter 6 Solutions

Case Study 1: Deconstructing a Disk

6.1 The answers to this exercise can be determined from Figure L.35.

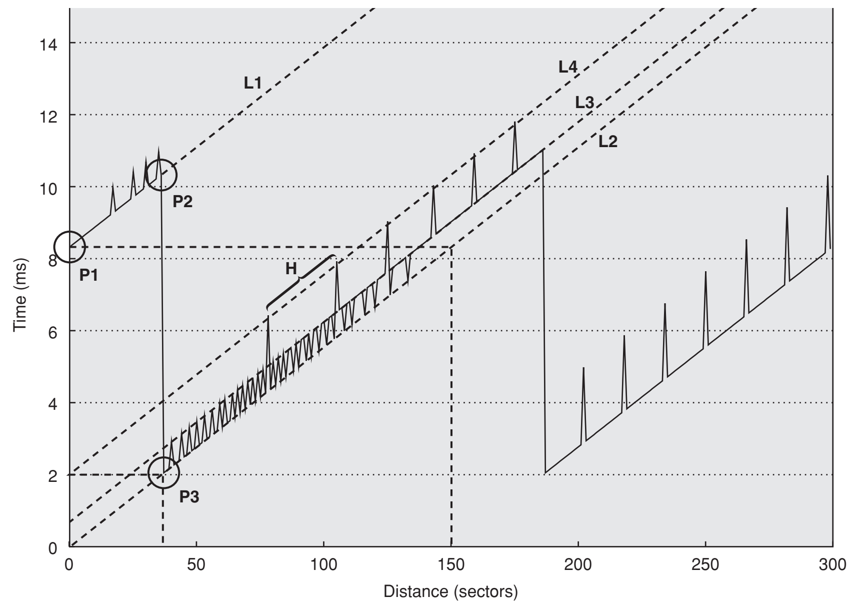


Figure L.35 Results from running Skippy on Disk Alpha.

a. 2.0 ms; determined at point 3 (P3) in the graph.

b. 8.33 ms; determined at point 1 (P1) in the graph, or (more robustly) the *y*-axis difference between line 1 (L1) and line 2 (L2). The simplest way to think about this is to consider the transition from P2 to P3. At P2, the algorithm incurs the time of a full rotation plus the minimum transfer time (worst case), and at P3, just the minimum transfer time (best case). The difference between the two cases is the rotation time.

c. 0.7 ms; determined by the difference between L2 and L3. When skipping over larger distances, occasionally a head switch must take place to move us onto a different track. Hence, these show up as small blips (on L3) just above the baseline (L2).

d. 2.1 ms; determined by the difference between L2 and L4. Similar to above, when a cylinder is exhausted, a switch must occur to the next one. These show up as (larger) blips (on L4) above the baseline (L2).

e. 15 heads; determined by counting the number of head switches that occur between the first two cylinder switches (labeled H on the graph).

6.2 The answers to this exercise can be determined from Figure L.36.

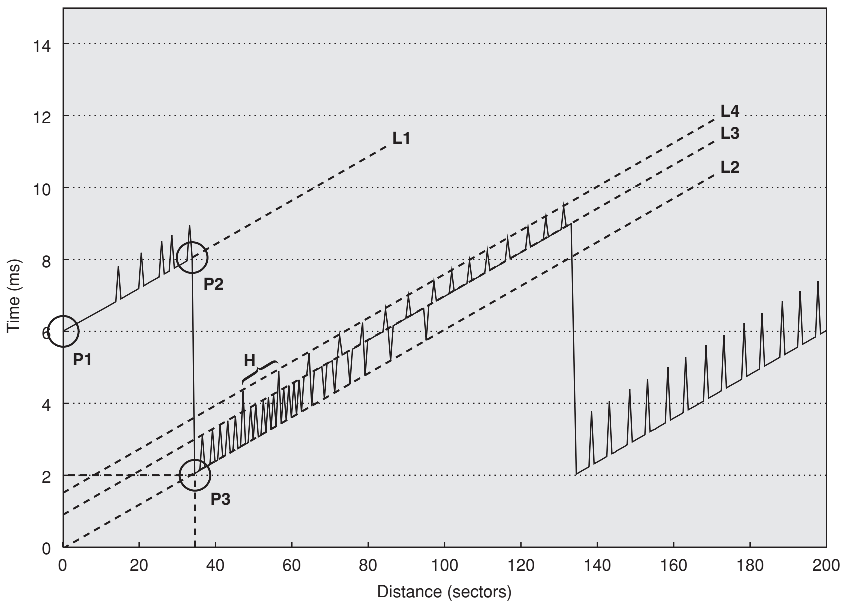


Figure L.36 Results from running Skippy on Disk Beta.

The minimal transfer time determines the *y*-axis value of point 3 (P3). The minimal transfer time divided by the rotational latency determines the *x*-axis value of point 3, as it reflects the number of sectors traversed in that time. The slope of the lines L1 . . . L4 is determined by dividing the rotational time by the number of sectors per track; L2 is the base line and essentially goes through the *y*-axis at *x* = 0, L3 is just a head switch time above L2, and L4 a cylinder switch time above L2. The most difficult aspect of drawing the graph, then, is determining exactly where each point goes. A simple algorithm that counts how many sectors have been traversed by each skip is needed; when the total exceeds the number of sectors per track, one must plot a point on the head switch line (L2). When all surfaces have been used, a cylinder switch must take place (L3); otherwise, each skip simply takes a little longer rotationally and hence must stay on L1.

6.3 *Solutions will vary.*

Case Study 2: Deconstructing a Disk Array

6.4 a.
- The pattern size is detected by the highest time, which first occurs at 64 KB. At that point, all requests are directed to the same disk, and hence take longer.
- The 0.4 time corresponds to requests getting spread across all disks; 0.8 to half the disks getting utilized; and 1.6 to just a single disk.
- We know that we are utilizing a single disk at 1.6 s and with a pattern of 64 KB. Further, we know that half the disks are utilized with a pattern size of 32 KB and achieve twice the bandwidth of a single disk; hence, we are using twice a single disk's bandwidth or two disks. If half the disks is two, then we must have four disks.
- The chunk size is 16 KB. With four disks and a repeating pattern every 64 KB, simple division tells us how much data must reside on each disk.

b. Your graph should look similar to Figure 6.26, with the following differences: the peaks would be at 1.25, 2.5, and 5 s (instead of 0.4, 0.8, and 1.6); these numbers correspond to taking 1000 requests that complete in 5 ms each and spreading them across 4, 2, and 1 drive, respectively. Because the chunk size is smaller, though, the *y*-axis blips would occur sooner, at 16 KB (2.5 s at half the pattern size) and 32 KB (5.0 s at the full pattern).

6.5 a.
- 8 KB. We see the dips every 8 KB, which corresponds to the requests getting spread across two drives (neighbors) instead of landing on the same disk.
- Once again, they reflect the difference in parallelism; the higher time implies requests directed to one disk, the lower time means two.

b. The dips would occur every 12 KB instead of every 8 KB. The times would be different as well, with a low time of 2.5 ms (when 1000 requests get spread across 2 disks, and each take 5 ms) and a high time of 5 ms (when all 1000 end up on the same disk).

6.6 a. 12 chunks. Divide the pattern size by chunk size.

b. The 0th chunk and the 11th, the 1st and 10th, the 2nd and 9th, and so on.

c. 6 disks. There are 12 chunks, and each chunk seems to be on the same disk as one other chunk. Hence, 12 divided by 2 gives us 6 disks.

d. Start at the left, numbering chunks 0, 1, 2, . . . , 5 across disks; then go to the next row, and instead number 11, 10, 9, . . . , 6. This pattern aligns the chunks correctly in a pairwise fashion.

6.7 Your graph should be 12 by 12, and the following chunk pairs would conflict (and hence have lighter shading): (0, 0), (1, 1), . . . , (11, 11), which fills the diagonal, (0, 3), (0, 7), (1, 4), (1, 9), (2, 7), (2, 10), (3, 5), (3, 8), (3, 11), and all pairwise complements (3, 0), (7, 0), and so on.

Case Study 3: RAID Reconstruction

6.8 RAID 4 allocates parity blocks to a single disk.

6.9 Assuming independent failures for the six disks and an MTTF for each disk of 1.2 M hours (shown in Figure 6.3), the expected time until a failure is

$$\frac{1.2\text{ M hours}}{6} = 200{,}000\text{ hours}$$

6.10 a. To perform reconstruction, the RAID 4 array must read the (data or parity) blocks from the five working disks and write (the data or parity blocks) to the reconstructed disk. Note that the reads of stripe $i + 1$ can proceed in parallel with the write to stripe i.

b. If reads and writes are proceeding in parallel (and we ignore the fact that the reads to the first stripe and the writes to the last stripe are not overlapped) and the bus is the limiting factor with 320 MB/sec, then the bandwidth delivered to/from each disk is

$$\frac{320\text{ MB/sec}}{6} = 53\text{ MB/sec}$$

Reading or writing 37 GB at 53 MB/sec requires time

$$\frac{37\text{ GB}}{53\text{ MB/sec}}$$

= 715 seconds = 11.9 minutes = 0.198 hours.

c. With a 10 MB/sec limit, the bandwidth delivered to/from each disk is

$$\frac{10\text{ MB/sec}}{6} = 1.67\text{ MB/sec}$$

Reading or writing 37 GB at 1.67 MB/sec requires time

$$\frac{37\text{ GB}}{1.67\text{ MB/sec}} = 22{,}733\text{ seconds} = 379\text{ minutes} = 6.3\text{ hours}$$

6.11 a. Probability, x, of a second failure during offline reconstruction:

$$\frac{5}{1.2\text{ M hours}} = \frac{x}{0.198\text{ hours}}$$

$$x = 8 \times 10^{-7}$$

b. Offine MTDL $= \dfrac{200{,}000\text{ hours}}{8 \times 10^{-7}} = 2.5 \times 10^{11}$ hours

c. Probability, x, of a second failure during on-line reconstruction:

$$\frac{5}{1.2\text{ M hours}} = \frac{x}{6.3\text{ hours}}$$

$$x = 2.63 \times 10^{-5}$$

d. Online MTDL $= \dfrac{200{,}000 \text{ hours}}{2.63 \cdot 10^{-5}} = 7.62 \times 10^{9}$ hours

6.12 Assume that the bus does not limit the IOPS for a random workload. With offline reconstruction, the performance during reconstruction is 0.

Performability = Probability$_{\text{nofailures}}$ × (285 IOPS × 5 disks)

$$= \left(1 - \frac{0.198 \text{ hours}}{200{,}000 \text{ hours}}\right) \times 1425 \text{ IOPS}$$

= 1424.9986

6.13 The performability of offline reconstruction is slightly better:

Probability$_{\text{nofailures}}$ × 285 IOPS × 5 disks + Probability$_{\text{1failure}}$ × 0.70 × 285 IOPS × 5 disks

$$= \left(1 - \frac{6.3 \text{ hours}}{200{,}000 \text{ hours}}\right) \times 1425 \text{ IOPS} + \frac{6.3 \text{ hours}}{200{,}000 \text{ hours}} \times 997.5 \text{ IOPS}$$

= 1424.9551 + 0.0314213 = 1424.9865

6.14 To construct the missing blocks, one should first read all of the blocks from the four available disks (that fit in main memory) in order to obtain the best sequential read bandwidth. Then, only after all of the missing blocks have been reconstructed should those blocks be written out to the two repaired disks, to again obtain the best sequential write bandwidth. The XOR computations can be performed in the following order (note that other orderings are also possible):

Disk 0, Block 2: Reconstruction using diagonal parity.

Disk 3, Block 0: Reconstruction using row parity.

Disk 0, Block 0: Reconstruction using diagonal parity.

Disk 3, Block 3: Reconstruction using row parity.

Disk 0, Block 3: Reconstruction using diagonal parity.

Disk 3, Block 1: Reconstruction using row parity.

Disk 0, Block 1: Reconstruction using diagonal parity.

Disk 3, Block 4: Reconstruction using row parity (could have used diagonal parity from the beginning as well).

Case Study 4: Performance Prediction for RAIDs

6.15 a. 5 ms

b. Utilization = Arrival rate × Service time = 167 requests/sec × 5 ms = 83.5%

c. Wait time $= \text{Time server} \times \dfrac{\text{Util}}{1 - \text{Util}} = 5 \text{ ms} \times \dfrac{0.835}{1 - 0.835} = 25.3$ ms

d. $\dfrac{\text{Util}^2}{1-\text{Util}} = 4.23$

e. Wait time + Service time = 25.3 ms + 5 ms = 30.3 ms

6.16 a. Since the data is striped across the two disks, each disk now contains only 20 GB of data; therefore, the expected service time of a request is now only 2.5 ms.

b. This system of two disks cannot be modeled as an M/M/2 queue because the requests cannot go to either disk; instead a request must go to the particular disk holding the requested block. Therefore, we instead model the storage system as two independent M/M/1 queues. We assume that the request rate in each queue is half the original rate, 83 requests/sec.

c. Utilization = Arrival rate × Service time = 83 requests/sec × 2.5 ms = 21%

d. $\text{Wait time} = \text{Time server} \times \dfrac{\text{Util}}{1-\text{Util}} = 2.5\text{ ms} \times \dfrac{0.21}{1-0.21} = 0.665\text{ ms}$

e. $\dfrac{\text{Util}^2}{1-\text{Util}} = \dfrac{0.21^2}{1-0.21} = 0.056$

f. Wait time + Service time = 0.665 + 2.5 = 3.165

6.17 a. Each disk now contains the full 40 GB of data; therefore, the expected service time of a request is again 5.0 ms.

b. Since read requests can go to either disk, we model this system as a single M/M/2 queue.

c. $\text{Utilization} = \dfrac{\text{Arrival rate} \times \text{Service time}}{N\text{ servers}} = \dfrac{165\text{ requests/sec} \times 5\text{ ms}}{2} = 41.2\%$

d.
$$\begin{aligned}\text{Prob 0 tasks} &= \left(1 + \frac{(2 \times 0.412)^2}{2 \times (1-0.415)} + (2 \times 0.412)\right)^{-1}\\ &= (1 + 0.5774 + 0.824)^{-1}\\ &= 2.401^{-1}\\ \text{Prob tasks in Q} &\left(= \frac{(2 \times 0.412)^2}{2 \times (1-0.412)}\right) \times 2.401^{-1}\\ &= 0.240\\ \text{Wait time} &= \text{Time server} \times \frac{0.240}{2(1-\text{Util})}\\ &= 5.0\text{ ms} \times \frac{0.240}{2(1-0.412)}\\ &= 1.020\end{aligned}$$

e. Wait time + Service time = 1.020 + 5.0 ms = 6.020 ms

6.18 a. For a write-only workload, the requests must go to both disks. Therefore, this system performs identically to the single disk system.

b. The answers are identical as for Exercise 6.15.

Case Study 5: I/O Subsystem Design

6.19 If reliability is not a concern, then RAID 0 gives the best capacity and performance; with RAID 0, we waste no space for redundancy to recover from failures and each independent disk can be used to handle a random I/O request. Larger block sizes amortize the positioning costs, while smaller block sizes ensure that only needed data is actually transferred; therefore, the block size should roughly match the request size of 16 KB.

6.20 In our system, we want to purchase as many disks as possible. To best use our budget of \$28,000, we therefore maximize the number of disks, given the following constraints:

$$\text{System}_{\text{Cost}} = \text{CPU}_{\text{Cost}} + \text{Cntrler}_{\text{Cost}} \times \text{Cntrler}_{\text{Num}} + \text{Encl}_{\text{Cost}} \times \text{Encl}_{\text{Num}} + \text{Disk}_{\text{Cost}} \times \text{Disk}_{\text{Num}}$$

$$\text{Disk}_{\text{Num}} \le \text{Encl}_{\text{Num}} \times 8$$

$$\text{Disk}_{\text{Num}} \le \text{Cntrler}_{\text{Num}} \times 15$$

$$28{,}000 = 1000 + 250 \times \text{Disk}_{\text{Num}}/15 + 2000 \times \text{Disk}_{\text{Num}}/8 + 150 \times \text{Disk}_{\text{Num}}$$

$$\text{DiskNum} = \frac{27{,}000}{417} = 64.8$$

Since the number of each component must be an integer, we choose

$$\text{Disk}_{\text{Num}} = 64,\ \text{Cntrler}_{\text{Num}} = 5$$

and

$$\text{Encl}_{\text{Num}} = 8$$

a. Each disk can deliver 285 IOPS. The CPU is not the bottleneck and the 50,000 IOPS controller is not the bottleneck of the system (assuming 15 disks or fewer per string). If all disks are operating concurrently on the random requests, and the disks are the bottleneck of the system, then the storage system can deliver 18,240 IOPS.

b. $1000 + 250 \times 5 + 2000 \times 8 + 150 \times 64 = 27{,}850$

Note that even though we have \$150 remaining, it cannot be used for another disk, because we do not have space in any of the enclosures.

c. 64×37 GB = 2.31 TB

d.

$$\text{Failure rate} = \frac{1}{1{,}000{,}000} + \frac{5}{1{,}000{,}000} + \frac{5}{500{,}000} + \frac{8}{1{,}000{,}000} + \frac{8}{200{,}000} + \frac{8}{200{,}000} + \frac{64}{1{,}200{,}000}$$

$$= 0.0001573$$

$$\text{MTTF} = \frac{1}{\text{Failure rate}} = 6356 \text{ hours}$$

6.21 If an enclosure or controller fails, we want to ensure that only one of the replicas is affected; that is, each enclosure should contain a RAID 0 array and mirroring should be performed across RAID 0 arrays, which is RAID 01. Further, to ensure that one controller failure does not affect multiple enclosures, there should be exactly one controller per enclosure.

6.22 Given a RAID 01 array, to keep reliability as high as possible, we want to have as few components as possible, while meeting our capacity requirements. Given that we must supply 1 TB of storage space and each pair of disks supplies 37 GB, we must have 28 disk mirrors, or 56 total disks. To contain RAID 0 arrays of 56 disks, we must have 8 enclosures and 8 controllers.

a.

$$\text{Failure rate} = \frac{1}{1{,}000{,}000} + \frac{8}{1{,}000{,}000} + \frac{8}{500{,}000} + \frac{8}{1{,}000{,}000} + \frac{8}{200{,}000} + \frac{8}{200{,}000} + \frac{56}{1{,}200{,}000}$$

$$= 0.0001597$$

$$\text{MTTF} = \frac{1}{\text{Failure rate}} = 6263 \text{ hours}$$

b. To lose data in this system, we must lose two mirrored pairs of disks before the one is recovered (which requires 24 hours). Note that the failure of an enclosure or a controller will not lead to permanent data loss. The expected time before one disk fails is

$$\frac{1{,}200{,}000}{56} = 21{,}429 \text{ hours}$$

The likelihood of the second mirrored disk failing during the 24-hour recovery is

$$\frac{1}{1.2\text{M hours}} = \frac{x}{24 \text{ hours}}$$

$$x = 2 \times 10^{-5}$$

Therefore,

$$\text{MTDL} = \frac{21{,}429 \text{ hours}}{2 \times 10^{-5}} = 1.07 \times 10^{9} \text{ hours}$$

c. 28 × 37 GB = 1.01 TB

d. 1000 + 250 × 8 + 2000 × 8 + 150 × 56 = 27,400

e. A write-only workload must update both mirrors. Therefore, the IOPS is 28 × 285 IOPS = 7980 IOPS.

6.23 For reliability, we want as few components as possible; with larger disks, we are able to use half as many disks. Therefore, our system will now contain a total of 28 disks; we will have 4 enclosures and 4 controllers, each with a string of 7 disks.

Case Study 6: Dirty Rotten Bits

6.24 a. A half full file system of 37 GB has 18.5 GB of data. We need only checksums to detect single errors (no correction here), and therefore need a 20-byte MD5 for every 4 KB of each file. We don't need to look at the file distribution here because we know we have 18.5 GB of data in 4 KB blocks (roughly). Hence, we have

$$\frac{18.5\text{ GB}}{4\text{ KB}}\text{ blocks, or 4,849,664 blocks}$$

In terms of size, that is 92.5 MB of checksum data (20 bytes per 4 KB block).

Another way to get this: simply multiply the total data size (18.5 GB) by the additional fractional amount that the checksum will add per block $\left(\frac{20}{4096}\right)$.

As a percentage, this is 0.49% of the volume (again, 20 bytes extra per data block, or $\left(\frac{20}{4096}\right)$.

b. The answer here requires us to add a single parity block per file. Thus, we have to know the number of files in the volume; more specifically, how many files comprise an 18.5 GB volume? One way to figure this out is to assume you have 1000 files distributed by size as per the distribution above. Then you would have 266 1-KB files, 110 2-KB files, and so forth. By adding these up, we can find the total size of those 1000 files: 77.9668 MB. Hence, to find how many files are in 18.5 GB of data, just set up the following:

$$\frac{18.5\text{ GB}}{n\text{ files}} = \frac{77.9668\text{ MB}}{1000\text{ files}}$$

Solving for *n,* we get roughly 242,975 files. To get the total size of this, we just multiply by the size of the parity block, 4 KB, or roughly 949.1 MB (5.01% of the volume).

c. The file size distribution does not determine the checksum overhead. That said, bigger blocks mean less protection but with less overhead; if the 0.5% overhead for 20-byte checksums is too much, it is easy to increase the block size to reduce this number. If comparing to the space cost of recovery, the parity protection information takes up more space (by a factor of 10), so perhaps it is easy to justify more checksum data per file.

6.25 a. With 10 GB of data, it takes you roughly 100 seconds to scan all of the data at the 100 MB/sec rate. With 1 bit flip per GB per month, assuming a 30-day month, you get a bit flip on average once every 3 days in your 10 GB data set. Hence, to avoid a second bit flip corrupting another block in the same file, you should scan every few days or so to repair the likely flipped bit. A more sophisticated answer could take the file size distribution into account, but we leave that as an exercise for someone else.

b. Because it takes 100 seconds to scan a GB (and thus detect and potentially correct a single bit flip), if we get more than 1 bit flip every 100 seconds, we are in trouble. Translating to bit flips per GB per month, we know that there are 2,592,000 seconds per 30-day month, so we get roughly 2592 flips per GB per month.

6.26 a. A 40 MB file requires one 20-byte checksum per 4 KB block and 1 parity block. Hence, the additional write traffic is dominated by the checksums. It is easy to calculate that 40 MB implies 10,240 blocks, which leads to 200 KB of checksums plus a 4 KB parity block, or 204 KB of extra traffic. As a percentage of total traffic, this is miniscule: not quite 0.5%.

b. Now the costs go up quite a bit. For each write, we must do a "small write" RAID-style update of the parity block, as well as update the checksum information. An update to a particular block requires us logically to read the old data that was in the block, the old parity, compute the new parity (using the difference between the old and new data to derive the new parity), and then write the new data and new parity. If the parity block is cached (let's say this is the case), this will generate 3 I/Os (1 read and two writes) per logical write to the file. Hence, a workload that performs 10,000 writes will generate 30,000 total I/Os, or 117.2 MB of total traffic. In terms of extra traffic, this is 78.1 MB of extra I/O traffic. But we still need to add in checksums: 20 bytes for each of the 10,000 writes, or roughly 0.2 MB. Thus, 78.3 MB of extra I/O traffic is generated while writing out 10,000 4-KB random synchronous writes (which is 39.1 MB of traffic), roughly increasing the total traffic by 200%. In this case, parity-maintenance traffic dominates.

c. First, we must figure out how much checksum data is on disk. With a 10,000-block file, there must be 0.19 MB of checksums (roughly). Assume we read those into memory first, and then proceed to read the file sequentially. Thus, we get a single seek to the checksum file (4 ms), 0.19 MB read at 100 MB/sec (1.9 ms), another seek to the beginning of the file data (4 ms), and a long sequential read of the file (390.6 ms). Thus, the total time to read the file and checksums is 400.5 ms, of which 5.9 ms is due to the checksums. Overall, for sequential reads, the time penalty of checksums is small.

d. Assume that for each random read, you never have the checksum cached, and hence you must read it from disk too. Hence, for each random read, you must seek to the checksum (4 ms), read it from disk (negligible, just reading 20 bytes or more realistically a single block at 100 MB/sec), seek to the data block (4 ms), and read it too (again, negligible). Hence, the total time per read is 8 ms, and for all 10,000 reads, 80 seconds. The time penalty due to checksums is half of that time, or 40 seconds. Random reads, when checksums aren't cached, are much more costly.

6.27 *Solutions may vary.*

Case Study 7: Sorting Things Out

6.28 To answer this question, we must first build a model of sorting performance. We can then plug in the numbers from the table and see what the total performance is. To read and write the data as fast as possible: simply configure the system to have two of the fast I/O interconnects (each rated at 320 MB/sec) and then add disks to each bus until the bus is fully utilized (three fast disks, each rated at 110 MB/sec). Make sure the memory bus can handle this as well: because there is a copy during reading and writing, that means that data passes across the memory bus three times during I/O, which means the memory bus must be able to handle three times the bandwidth of the I/O subsystem. Hence, use the fastest memory system available here, the fast memory subsystem. For sorting, the time is determined by which is slower, the memory subsystem or the CPU (recall the problem states that sorting 1 MB of data requires 1 MIPS of CPU power and 1 MB/sec of memory bandwidth). Hence, pick the standard or fast CPU (memory bandwidth limits us to 2 GB/sec already, so no real reason to pick the fast CPU here). Assume further that we have to buy 2 GB of memory to make sure we can do this all in memory.

a. ■ Standard CPU: $1,000
   ■ Fast memory: 2 GB at $500/GB = $1000
   ■ 2 × fast I/O interconnect: $800
   ■ 6 × fast disks: $1800
   ■ **Total:** $4600

b. ■ Reading: 1.6 sec
   ■ Sorting: 0.51 sec
   ■ Writing: 1.6 sec
   ■ **Total:** 3.71 sec

c. In this system, it would have to be said that I/O is the bottleneck, as reading and writing time dominate.

6.29 This is a bit of a search problem. Probably the easiest way to do it is to write a little program that searches through the possibilities. The configuration we came up with is described below, although many other similarly performing configurations are possible.

a. ■ Standard memory (2 GB) : $400
   ■ Slow CPU: $200
   ■ Fast I/O + slow I/O: $450
   ■ Standard disks (5) [on fast I/O bus]: $600
   ■ Standard disk (1) [on slow I/O]: $120
   ■ **Total:** $1770

b.
- Read: 3.0 sec
- Sort: 1.024 sec
- Write: 3.0 sec
- **Total:** 7.024 sec

c. Once again, I/O is the bottleneck, which is good for a sorting benchmark. Interestingly, though, the I/O is memory-bus limited—adding more disks does not help even though the I/O bus has the potential capacity. The copying cost of moving data in memory during I/O is the ultimate limitation.

6.30 a. We assume the disk is the bottleneck and that it is kept busy at all times. When we read a chunk from one of the sorted runs, it takes the total of the time to position the disk $T_{\text{seek}} + T_{\text{rotate}}$ plus the transfer time T_{transfer}.

In this case, T_{seek} is 7 ms and T_{rotate} is half the full rotation time, or 3 ms. Hence, positioning time is 10 ms. The time to read in a 1 MB run is simply

$$\frac{1\ \text{MB}}{100\ \text{MB/sec}}$$

or 10 ms. Hence, the total time to read in a chunk from a run is 20 ms, which means the total time to read in all 100 runs of size 1 GB is

$$\frac{\text{Data} = 100 \times 1024\ \text{MB}}{\text{Run size} = 1\ \text{MB}} \times (\text{Time per run} = 20\ \text{ms})$$

which is 2048 seconds. We must also remember the time to write out the runs, which is very nearly the time to write out 100 GB at the sequential disk rate:

$$\frac{100\ \text{GB}}{100\ \text{MB/sec}}$$

or 1024 seconds. Hence, the total time to perform the second pass of the sort is 3070 seconds.

b. This basically changes the time to access each chunk from 20 ms (10 ms positioning and 10 ms transfer) to 110 ms (10 ms positioning plus 100 ms transfer). The time to read in all the runs is thus

$$\frac{\text{Data} = 100 \times 1024\ \text{MB}}{\text{Run size} = 10\ \text{MB}} \times (\text{Time per run} = 110\ \text{ms})$$

or 1126.4 seconds. Hence, the total time is 2150.4 seconds.

c. Peak disk efficiency would simply be transferring data sequentially the entire time, or taking 2048 seconds to complete the merge. Thus, the efficiency of the first case is

$$\frac{2048}{3070}$$

or 0.667. In the second case, by using larger blocks, we do much better, achieving

$$\frac{2048}{2150.4}$$

or 0.95.

6.31 *Solutions will vary.*