B

Solutions to All Exercises for Instructors

Captain Kirk
You ought to sell an instruction and maintenance manual with this thing.

Cyrano Jones
If I did, what would happen to man's search for knowledge?

Star Trek
"The Trouble with Tribbles" (Dec. 29, 1967)

Introduction

Captain Kirk and Cyrano Jones were both right. Nonetheless, here is a set of solutions for all the exercises that admit to a specific solution. Generally, the solutions present a way of approaching the exercise as well as the required solution.

For an exercise with a solution in the form of a program to be written for a platform of the reader's choice or for free-wheeling discussion of a topic, typically the placeholder "No solution provided." will be given instead of a detailed solution. For some of these type exercises a solution sketch is provided.

"Starred" solutions, indicated by the @ symbol, are included in the hardcopy of the text and are available to all.

Note

Exercise 4.22 contains an error that is addressed in its solution here.

Chapter 1 Solutions

1.1 a. For each generation—every three years—DRAM density improves four times while magnetic disk density improves two times. Thus, DRAM cost per megabyte decreases twice as fast as does magnetic disk.

Generation (year)	1990	1993	1996	1999	2002	2005
DRAM $/MB / Disk $/MB	20	10	5	2.5	1.25	0.625

From this 1990 projection the conclusion is that disk drives, always much slower than DRAM, would by today be very similar in cost per unit of storage capacity. With non-volatility as the remaining selling point for disk drives, we would predict the magnetic disk industry to be facing a shrinking market.

b. The DRAM data in Figure 1.5 is difficult to extract for small dollar-per-chip values and is not corrected for inflation as is Figure 7.4. The following table shows a reasonable set of data and then derives the desired ratio.

Year	$ per DRAM chip	Disk $/GB	Ratio of costs, DRAM to disk
1983	$5 per 64 Kb	$150,000	4.3
1986	$5 per 256 Kb	$25,000	6.4
1989	$3 per 256 Kb	$10,000	9.6
1992	$13 per 4 Mb	$3000	8.7
1995	$5 per 4 Mb	$600	16.7
1998	$1.3 per 16 Mb	$75	8.7
2001	$0.7 per 64 Mb	$20	4.3

Disk prices are dropping very quickly at the present time, and the ratio is likely many times larger.

c. Assume that the relative disk volume scales linearly with disk capacity.

$$\text{Projected value} = 1990\ \text{value} \times \frac{30\ \text{GB}}{100\ \text{MB}} \times (1 - 0.3)^{\text{years}}$$

where years is the number of years forward from 1990. Then,

$$\text{Mass}_{2002} = 1000\ \text{g} \times \frac{30\ \text{GB}}{100\ \text{MB}} \times (1 - 0.3)^{12} = 4152\ \text{g}$$

$$\begin{aligned}\text{Height}_{2002} &= \text{Volume}_{2002} / \text{Drive bay area} \\ &= 1000\ \text{g} \times \frac{30\ \text{GB}}{100\ \text{MB}} \times (1 - 0.3)^{12} \\ &= 29.7\ \text{cm}\end{aligned}$$

d. Actual component cost of the $\$1000$ PC $= \$1000 \times 46.6\% = \$466$. Cost of components other than the hard disk is $\$466 \times 91\% = \$424$.

e. Cost of hard disk is $\$466 \times 9\% = \$42$.

Assume disk density did improve 60% per year from 1990 through 1996 and at 100% per year since 1997. Then by 2001 an improvement of only 30% per year would have lead to a higher hard disk cost of

$$\$42 \times \frac{(1 + 60\%)^6 (1 + 100\%)^5}{(1 + 30\%)^{11}} = \$1258.$$

Adding this to the cost of the other components and scaling comp0onent cost up to list price gives PC cost $= (\$424 + \$1258)/46.6\% = \$3609$. At this higher price desktop digital video editing would be much less widely accessible.

1.2 Let PV stand for percent vectorization divided by 100.

a. Plot Net speedup $= \dfrac{\text{PV}/10}{1 - \text{PV} + \dfrac{\text{PV}}{10}}$ for $0 \le \text{PV} \le 1$.

b. From the equation in (a), if Net speedup $= 2$ then the percent vectorization is $5/9$ or 56%.

c. Time in vector mode $= \dfrac{\text{PV}/10}{1 - \text{PV} + \dfrac{\text{PV}}{10}} = 1/9$ or 11%.

d. From the equation in (a), if Net speedup $= 10/2 = 5$ then PV $= 8/9$ or 89%.

e. The increased percent vectorization needed to match a hardware speedup of $10 \times 2 = 20$ applied to the original 70% vectorization is

$$\frac{1}{1 - \text{PV} + \dfrac{\text{PV}}{10}} = \frac{1}{1 - 70\% + \dfrac{70\%}{20}}$$

Solving shows that the vectorization must increase to 74%, not a large increase. Improving the compiler to increase vectorization another 4% may be easier and cheaper than improving the hardware by a factor of 2.

1.3 This question further explores the effects of Amdahl's Law, but the data given in the question is in a form that cannot be directly applied to the general speedup formula.

a. Because the information given does not allow direct application of Amdahl's Law we start from the definition of speedup:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Time}_{\text{unenhanced}}}{\text{Time}_{\text{enhanced}}}$$

The unenhanced time is the sum of the time that does not benefit from the 10 times faster speedup plus the time that does benefit, but before its reduction by the factor of 10. Thus,

$$\text{Time}_{\text{unenhanced}} = 50\%\ \text{Time}_{\text{enhanced}} + 10 \times 50\%\ \text{Time}_{\text{enhanced}} = 5.5\ \text{Time}_{\text{enhanced}}.$$

Substituting into the equation for Speedup yields

$$\text{Speedup}_{\text{overall}} = \frac{5.5\text{Time}_{\text{enhanced}}}{\text{Time}_{\text{enhanced}}} = 5.5.$$

b. Using Amdahl's Law, the given value of 10 for the enhancement factor, and the value for $\text{Speedup}_{\text{overall}}$ from part (a), we have

$$5.5 = \frac{\text{Fraction}_{\text{enhanced}}}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{10}}$$

Solving shows that the enhancement can be applied 91% of the original time.

1.4 a. $$\text{Speedup} = \frac{\text{Number of floating-point instructions DFT}}{\text{Number of floating-point instructions FFT}}$$

$$= \frac{n^2}{n \log_2 n}$$

Thus,

n	8	16	32	64	128	256	512	1024
Speedup	2.7	4.0	6.4	10.7	18.2	32.0	56.9	102.4

Also,

$$\lim_{n \to \infty} \text{Speedup} = \lim_{n \to \infty} \frac{n^2}{n \log_2 n} = \infty$$

b. Percent reduction $= 1 - \dfrac{\text{Number of floating-point instructions FFT}}{\text{Number of floating-point instructions DFT}}$

$$= 1 - \frac{1024 \times \log_2 1024}{1024^2}$$

$$= 99\%$$

c. Choosing to include a branch-target buffer in a processor means adding circuitry to the unenhanced design. This increases die size, testing time, and power consumption, making the enhanced processor more costly. Choosing to use an equivalent, asymptotically faster algorithm, such as the FFT, incurs no cost. Thus, the better algorithm will be universally adopted.

1.5 The examples cited in this exercise ask the reader to compare performance of two systems based on different descriptions: the first one uses speedup, the second one is based on the CPI factor of the expression for CPU time. The improvements described in the two examples apply to precisely the same subsets of the instruction set, and the factors of improvement for each subset are also exactly the same. The improvements are not described as having any effect on the other two factors in the CPU time formula, instruction count and clock cycle time. Thus, because the speedup in the first example is a ratio of CPU execution time, and changing CPI alone results in a ratio of CPU time equal to the ratio of the CPI change, the two examples set forth the same CPU improvement scenario.

1.6 The proposed formulation for $MIPS_B$ can be rewritten as

$$\frac{MIPS_A}{D_A} = \frac{MIPS_B}{D_B}$$

Examining the units of each factor, we have

$$\frac{\text{Computer } A \text{ instructions/second}}{\text{Dhrystone/second}} = \frac{\text{Computer } B \text{ instructions/second}}{\text{Dhrystone/second}}$$

The time units factor out, revealing that the formulation is founded upon the assumption that

Computer A instructions/Dhrystone = Computer B instructions/Dhrystone

Unless Computer A and Computer B have the same instruction set architecture and execute identically compiled Dhrystone executables, this assumption is likely false. If so, the formulation for $MIPS_B$ is also incorrect.

1.7 The exercise statement omits the value of the clock cycle time. We assume that the clock rate is the same for both processors and solve in terms of this unknown.

a. $$\text{IC}_{\text{RISC}} = \frac{\text{CPU time}_{\text{RISC}}}{\text{CPI}_{\text{RISC}} \times \text{CC}_{\text{RISC}}} = \frac{1.08 \text{ sec}}{10 \times \text{CC}} = 0.108 \times \text{CC instructions}$$

$$\text{IC}_{\text{emb}} = \frac{\text{CPU time}_{\text{emb}}}{\text{CPI}_{\text{emb}} \times \text{CC}_{\text{emb}}} = \frac{13.6 \text{ sec}}{6 \times \text{CC}} = 2.27 \times \text{CC instructions}$$

Relatively, there are 21 times more instructions executed by the embedded processor.

b. $$\text{MIPS}_{\text{RISC}} = \frac{\text{CC}_{\text{RISC}}}{\text{CPI}_{\text{RISC}} \times 10^6} = \frac{\text{CC}}{10 \times 10^6}$$

$$\text{MIPS}_{\text{emb}} = \frac{\text{CC}_{\text{emb}}}{\text{CPI}_{\text{emb}} \times 10^6} = \frac{\text{CC}}{6 \times 10^6}$$

The MIPS rating of the embedded processor will be a factor of 10/6 = 1.67 times higher than the rating of the RISC version.

c. The RISC processor performs the non-FP instructions plus 195,578 FP instructions. The embedded processor performs the same number of non-FP instructions as the RISC processor, but performs some larger number of instructions than 195,578 to compute the FP results using non-FP instructions only. The number of non-FP instructions is

$$\text{Number of non-FP instructions} = \text{IC}_{\text{RISC}} - 195578 = 0.108\ \text{CC} - 195578$$

Thus,

$$\begin{aligned}\text{Number instructions for FP}_{\text{emb}} &= \text{IC}_{\text{emb}} - \text{Number non-FP instructions} \\ &= 2.27\ \text{CC} - (0.108\ \text{CC} - 195578) \\ &= 2.16\ \text{CC} + 195578\end{aligned}$$

Finally,

$$\begin{aligned}\text{Average number instr. for FP in software}_{\text{emb}} &= \frac{\text{Number instr. for FP}_{\text{emb}}}{\text{Number FP instr.}} \\ &= \frac{2.16\ \text{CC} + 195578}{195578}\end{aligned}$$

1.8 Care in using consistent units and in expressing dies/wafer and good dies/wafer as integer values are important for this exercise.

a. The number of good dies must be an integer and is less than or equal to the number of dies per wafer, which must also be an integer. The result presented here assumes that the integer dies per wafer is modified by wafer and die yield to obtain the integer number of good dies.

Microprocessor	Dies/wafer	Good dies/wafer
Alpha 21264C	231	128
Powe3-II	157	71
Itanium	79	20
MIPS R14000	122	46
UltraSPARC III	118	44

b. The cost per good die is

Microprocessor	$/ good die
Alpha 21264C	$36.72
Powe3-II	$56.34
Itanium	$245.00
MIPS R14000	$80.43
UltraSPARC III	$118.18

c. The cost per good, tested, and packaged part is

Microprocessor	$ / good, tested, packaged die
Alpha 21264C	$64.77
Powe3-II	$78.67
Itanium	$268.33
MIPS R14000	$108.49
UltraSPARC III	$152.18

d. The largest processor die is the Itanium at 300 mm^2. Defect density has a substantial effect on cost, pointing out the value of carefully managing the wafer manufacturing process to maximize the number of defect-free die. The table below restates die cost assuming the baseline defect density from parts (a)–(c) and then for the lower and higher densities for this part.

Itanium	$ / good, tested, packaged die
defect density = 0.5	$268.33
defect density = 0.3	$171.82
defect density = 1.0	$635.83

e. For the Alpha 21264C, tested, packaged die costs for an assumed defect density of 0.8 per cm^2 and variation in parameter α from $\alpha = 4$ to $\alpha = 6$ are $77.53 and $78.59, respectively.

1.9 a. Various answers are possible. Assume a wafer cost of $5000 and $\alpha = 4$ in all cases. For a defect density of 0.6 /cm^2 and die area ranging from 0.5 to 4 cm^2, then die cost ranges from $4.93 to 118.56. Fitting a polynomial curve to the (die area, die cost) pairs shows that a quadratic model has an acceptable norm of the residuals value of 0.669. Fitting to a third degree polynomial yields a very small cubic term coefficient and a better norm of the residuals of 0.017, but the quadratic fit is good and the polynomial simpler, so that would be the preferred choice.

b. For the same assumptions as part (a) except for a defect density of 2.0 /cm^2, die costs range from \$7.58 to \$313.82. A quadratic curve fit now has a norm of the residuals that is a high 2.38. A cubic fit yields a relatively large cubic coefficient and a good residual of 0.052. A quartic polynomial has a very small fourth degree coefficient and only somewhat better residual. A cubic polynomial is the lowest good fit.

1.10 a. $$\text{MFLOPS} = \frac{\text{Number of floating-point operations in a program}}{\text{Execution time in seconds} \times 10^6}$$

The exercise statement gives 100×10^6 as the number of floating-point operations, and Figure 1.15 gives the execution times. Figure S.1 shows the times and MFLOPS rates for each computer and program.

	Computer A		Computer B		Computer C	
Program	**Time**	**MFLOPS**	**Time**	**MFLOPS**	**Time**	**MFLOPS**
P1	1	100	10	10	20	5
P2	1000	0.1	100	1	20	5

Figure S.1 MFLOPS achieved by three computers for two programs.

b. From the computed MFLOPS and the formulas for arithmetic, geometric, and harmonic means, we find the results in Figure S.2.

	Computer		
Mean	**A**	**B**	**C**
Arithmetic	50.1	5.5	5.0
Harmonic	0.2	1.8	5.0
Geometric (normalize to A)	1.0	1.0	1.6
Geometric (normalize to B)	1.0	1.0	1.6
Geometric (normalize to C)	0.6	0.6	1.0

Figure S.2 Means of the MFLOPS ratings.

c. The arithmetic mean of MFLOPS rates trends inversely with total execution time. The geometric means, regardless of which normalization is used, do not show each difference in total execution time. Harmonic mean tracks total execution time best.

1.11 For positive integers a and b

$$\text{Arithmetic mean} = \text{AM} = \frac{a+b}{2}$$

and

$$\text{Geometric mean} = \text{GM} = \sqrt{ab}\,.$$

Now

$$\text{AM} - \text{GM} = \frac{a+b}{2} - \sqrt{ab} = \frac{a - 2\sqrt{ab} + b}{2} = \frac{\left(\sqrt{a} - \sqrt{b}\right)^2}{2} \geq 0$$

because the quotient of a nonnegative real number at a positive real number is nonnegative. Thus, AM ≥ GM.

Now assume that AM = GM. Then,

$$\frac{a+b}{2} = \sqrt{ab}\,.$$

Algebraic manipulation yields $\sqrt{a} - \sqrt{b} = 0$, which for positive integers implies $a = b$. So AM = GM when $a = b$.

1.12 For positive integers r and s

$$\text{Arithmetic mean} = \text{AM} = \frac{r+s}{2}$$

and

$$\text{Harmonic mean} = \text{HM} = \frac{2}{\frac{1}{r} + \frac{1}{s}}\,.$$

Now

$$\text{AM} - \text{HM} = \frac{a+b}{2} - \frac{2}{\frac{1}{r} + \frac{1}{s}} = \frac{(r-s)^2}{2(r+s)} \geq 0$$

because the quotient of a nonnegative real number at a positive real number is nonnegative. Thus, AM ≥ HM.

Now assume that AM = HM. Then,

$$\frac{r+s}{2} = \frac{2}{\frac{1}{r} + \frac{1}{s}}\,.$$

Algebraic manipulation yields $(r–s)^2 = 0$, which for positive integers implies $r = s$. So AM = HM when $r = s$.

1.13 a. Let the data value sets be

$$\text{A} = \{10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 1\}$$

and

$$\text{B} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 10^7\}$$

Arithmetic mean (A) = 9×10^6

Median (A) = 10×10^6

Arithmetic mean (B) = 1×10^6

Median (B) = 1

Set A mean and median are within 10% in value, but set B mean and median are far apart. A large outlying value seriously distorts the arithmetic mean, while a small outlying value has a lesser effect.

b. Harmonic mean (A) = 10.0
Harmonic mean (B) = 1.1

In this case the set B harmonic mean is very close to the median, but set A harmonic mean is much smaller than the set A median. The harmonic mean is more affected by a small outlying value than a large one.

c. Which is closest depends on the nature of the outlying data point. Neither mean produces a statistic that is representative of the data values under all circumstances.

d. Let the new data sets be

$$C = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 2\}$$

and

$$D = \{10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 10^7, 5 \times 10^6\}$$

Then

Arithmetic mean (C) = 9.5×10^6
Harmonic mean (C) = 9.1×10^6
Median (C) = 10×10^6

and

Arithmetic mean (D) = 1.1
Harmonic mean (D) = 1.05
Median (D) = 1

In both cases, the means and medians are close. Summarizing a set of data values that has less disparity among the values by stating a statistic, such as mean or median, is intrinsically more meaningful.

1.14 a. For a set of n programs each taking Time_i on one machine, the equal-time weightings on that machine are

$$w_i = \frac{1}{\text{Time}_i \times \sum_{j=1}^{n} \left(\frac{1}{\text{Time}_j}\right)}$$

Applying this formula to the Reference Time data for the 14 benchmarks yields the weights shown in Figure S.3.

Benchmark	Weight
168.wupwise	0.0840
171.swim	0.0433
172.mgrid	0.0746
173.applu	0.0640
177.mesa	0.0960
178.galgel	0.0463
179.art	0.0517
183.equake	0.1033
187.facerec	0.0707
188.ammp	0.0611
189.lucas	0.0672
191.fma3d	0.0640
200.sixtrack	0.1221
301.apsi	0.0517

Figure S.3 Equal-time weightings for SPEC CFP2000 benchmarks.

b.
$$\text{Weighted arithmetic mean} = \sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i$$
$$= \sum_{i=1}^{n} \text{Weight}_i \times \frac{100 \times \text{SPEC Reference Time}_i}{\text{SPEC Base Ratio}_i}$$

The factor of 100 in the numerator adjusts for the fact that the SPEC Base Ratios are reported as percentages. Figure S.4 lists the weighted runtimes of all the machines and benchmarks. The last line of the table shows the weighted arithmetic mean.

	Weighted time for each benchmark (seconds)			
Benchmark	**Reference computer**	**Compaq**	**IBM**	**Intel**
168.wupwise	134.4	29.3	43.8	34.2
171.swim	134.4	12.5	59.2	33.1
172.mgrid	134.4	25.6	47.3	54.6
173.applu	134.4	34.8	43.2	55.1
177.mesa	134.4	26.8	49.2	25.1
178.galgel	134.4	30.2	35.4	45.5
179.art	134.4	10.9	14.5	35.4
183.equake	134.4	61.1	25.4	57.7
187.facerec	134.4	19.8	62.5	45.4
188.ammp	134.4	33.2	49.4	47.5
189.lucas	134.4	21.0	51.5	43.1
191.fma3d	134.4	28.5	44.1	47.6
200.sixtrack	134.4	49.2	65.5	79.5
301.apsi	134.4	30.2	46.0	38.9
Weighted arithmetic mean (seconds)	**1881**	**413**	**637**	**643**
SPECfp_base2000 (geometric mean as percent)	**100**	**500**	**313**	**304**

Figure S.4 Weighted runtimes. The table entries for each benchmark show the time in seconds for that benchmark on a given computer. The summation of benchmark times gives the weighted arithmetic mean execution time of the benchmark suite. Note that with equal weighting of the benchmarks the three computers studied are ranked Compaq, IBM, Intel from fastest (lowest time) to slowest, which is the same ranking seen in the SPECfp_base_2000 numbers, where the highest corresponds to fastest.

1.15 a. The first condition is that the measured time be accurate, precise, and exclusively for the program of interest. Execution time is measured, typically, using a clock that ignores what the computer is running. This might be a clock on the wall or a free-running timer chip in the computer with an output that can be read using a system call. If the computer can work on computational tasks other than the program of interest during the measurement interval, then it is important to remove this other time from the run duration of the program of interest. If we cannot account for this other time, then the performance result derived from the measurement will be inaccurate, and may be of little meaning.

If the program completes execution in an interval that is short compared to the resolution of the timer, then the run time may be over- or under-stated enough due to rounding to affect our understanding. This is a problem of insufficient measurement precision, also known as having too few significant digits in a measurement. When a more precise timer (for example, microsec-

onds instead of milliseconds) is not available, the traditional solution is to change the benchmark program input to yield a longer run time so that the available timer resolution is then sufficiently precise. The goal is for the run time to become long enough to require the desired number of significant digits to express so that rounding will have an insignificant effect.

The condition that has to do with the program itself is what if the program does not terminate. or does not terminate within the patience of the measurer? How long then is the execution time? How should run time be defined?

b. Throughput is a consistent and reliable measure of performance if a consistent, meaningful unit of work can be defined. Consider a web server that sends a single, fixed page in response to requests. Each request then triggers a computational task, transfer identical web page description language to each new requesting computer, that is essentially identical each time. Throughput in terms of pages served per unit time would then be inversely proportional to the time to perform what is essentially a fixed benchmark task, serve this page. This is the same concept involved in measuring the time to run a fixed SPEC benchmark with its given code and given input data set. So throughput of fixed tasks is directly comparable to running fixed benchmarks.

When the task performed changes each time, for example very different pages served for each new request, then the use of throughput becomes more difficult. If an aggregate of tasks with consistent character exists, then throughput measured over a time interval that encompasses the collection of tasks may be sufficiently consistent and reliable. It may be difficult to identify such a task collection or to restrict the processing performed to just that collection.

c. With overlapped work, single transaction time will understate the amount of work, measured in units of number of transactions completed, that the computer can perform per unit time. Throughput will not understate performance in this way.

1.16 a. Amdahl's Law can be generalized to handle multiple enhancements. If only one enhancement can be used at a time during program execution, then

$$\text{Speedup} = \left[1 - \sum_i \text{FE}_i + \sum_i \frac{\text{FE}_i}{\text{SE}_i}\right]^{-1}$$

where FE_i is the fraction of time that enhancement i can be used and SE_i is the speedup of enhancement i. For a single enhancement the equation reduces to the familiar form of Amdahl's Law.

With three enhancements we have

$$\text{Speedup} = \left[1 - (\text{FE}_1 + \text{FE}_2 + \text{FE}_3) + \frac{\text{FE}_1}{\text{SE}_1} + \frac{\text{FE}_2}{\text{SE}_2} + \frac{\text{FE}_3}{\text{SE}_3}\right]^{-1}$$

Substituting in the known quantities gives

$$10 = \left[1 - (0.25 + 0.25 + \mathrm{FE}_3) + \frac{0.25}{30} + \frac{0.25}{20} + \frac{\mathrm{FE}_3}{15}\right]^{-1}$$

Solving yields

$$\mathrm{FE}_3 = 0.45$$

Thus, the third enhancement must be usable 45% of the time.

b. Let T_e and TNE_e denote execution time with enhancements and the time during enhanced execution in which no enhancements are in use, respectively. Let $T_{original}$ and $FNE_{original}$ stand for execution time without enhancements and the fraction of that time that cannot be enhanced. Finally, let FNE_e represent the fraction of the reduced (enhanced) execution time for which no enhancement is in use. By definition,

$$\mathrm{FNE}_e = \frac{\mathrm{TNE}_e}{T_e}$$

Because the time spent executing code that cannot be enhanced is the same whether enhancements are in use or not, and by Amdahl's Law, we have

$$\frac{\mathrm{TNE}_e}{\mathrm{T}_e} = \frac{\mathrm{FNE}_{\mathrm{original}} \times \mathrm{T}_{\mathrm{original}}}{\mathrm{T}_{\mathrm{original}}/\mathrm{Speedup}}$$

Cancelling factors and substituting equivalent expressions for $FNE_{original}$ and Speedup yields

$$\frac{\mathrm{FNE}_{\mathrm{original}} \times \mathrm{T}_{\mathrm{original}}}{\mathrm{T}_{\mathrm{original}}/\mathrm{Speedup}} = \frac{1 - \sum_i \mathrm{FE}_i}{1 - \sum_i \mathrm{FE}_i + \sum_i \frac{\mathrm{FE}_i}{\mathrm{SE}_i}}$$

Substituting with known quantities,

$$\mathrm{FNE}_e = \frac{1 - (0.25 + 0.35 + 0.10)}{1 - (0.25 + 0.35 + 0.10) + \left(\frac{0.25}{30} + \frac{0.35}{20} + \frac{0.10}{15}\right)} = \frac{0.3}{0.3325} = 90\%$$

c. Let the speedup when implementing only enhancement i be $\mathrm{Speedup}_i$, and let $\mathrm{Speedup}_{ij}$ denote the speedup when employing enhancements i and j.

$$\mathrm{Speedup}_1 = \left(1 - 0.15 + \frac{0.15}{30}\right)^{-1} = 1.17$$

$$\mathrm{Speedup}_2 = \left(1 - 0.15 + \frac{0.15}{20}\right)^{-1} = 1.17$$

$$\mathrm{Speedup}_3 = \left(1 - 0.7 + \frac{0.7}{15}\right)^{-1} = 2.88$$

Thus, if only one enhancement can be implemented, enhancement 3 offers much greater speedup.

$$\text{Speedup}_{12} = \left[1 - (0.15 + 0.15) + \frac{0.15}{30} + \frac{0.15}{20}\right]^{-1} = 1.40$$

$$\text{Speedup}_{13} = \left[1 - (0.15 + 0.7) + \frac{0.15}{30} + \frac{0.7}{15}\right]^{-1} = 4.96$$

$$\text{Speedup}_{23} = \left[1 - (0.15 + 0.7) + \frac{0.15}{20} + \frac{0.7}{15}\right]^{-1} = 4.90$$

Thus, if only a pair of enhancements can be implemented, enhancements 1 and 3 offer the greatest speedup.

Selecting the fastest enhancement(s) may not yield the highest speedup. As Amdahl's Law states, an enhancement contributes to speedup only for the fraction of time that it can be used.

1.17 a. $\text{MIPS}_{\text{proc}} = 120 \times 10^6 = \dfrac{I + YF}{W}$

$\text{MIPS}_{\text{proc/co}} = 80 \times 10^6 = \dfrac{I + F}{B}$

b. $I = 120 \times 10^6 W - FY$

$= (120 \times 10^6)(4) - (8 \times 10^6)(50)$

$= 80 \times 10^6$ instructions

c. $B = \dfrac{80 \times 10^6 + 8 \times 10^6}{80 \times 10^6} = 1.1$ sec

d. $\text{MFLOPS}_{\text{proc/co}} = \dfrac{F}{B - \text{Time for integer instructions}}$

$= \dfrac{F}{B - I/\text{MIPS}_{\text{proc/co}}}$

$= \dfrac{8 \times 10^6}{1.1 - 80 \times 10^6 / 80 \times 10^6}$

$= 80$ MFLOPS

e. The time for the processor alone is $W = 4$ sec. The time for the processor/co-processor configuration is $B = 1.1$ sec. While its MIPS rating is lower, the faster execution time belongs to the processor/co-processor combination. Your colleague's evaluation is correct.

1.18 a. $$\text{MFLOPS}_{\text{native}} = \frac{\text{Number of floating-point operations}}{\text{Execution time in seconds} \times 10^6}$$
$$= \frac{199827008653}{287 \times 10^6}$$
$$= 696$$

Because one of the two measured values (time) is reported with only three significant digits, the answer should be stated to three significant digits precision.

b. There are four 171.swim operations that are not explicitly given normalized values: load, store, copy, and convert. Let's think through what normalized values to use for these instructions.

First, convert comprises only 0.006% of the FP operations. Thus, convert would have to correspond to about 1000 normalized FP operations to have any effect on MFLOPS reported with three significant digits. It seems unlikely that convert would be this much more time-consuming than exponentiation or a trig function. Any less and there is no effect. So let's apply an important principle—keep models simple—and model convert as one normalized FP operation.

Next, copy replicates a value, making it available at a second location. This same behavior can be produced by adding zero to a value and saving the result in a new location. So, reasonably, copy should have the same normalized FP count as add.

Finally, load and store interact with computer memory. They can be quick to the extent that the memory responds quickly to an access request, unlike divide, square root, exponentiation, and sin, which are computed using a series of approximation steps to reach an answer. Because load and store are very common, Amdahl's Law suggests making them fast. So assume a normalized FP value of 1 for load and store. Note that any increase would significantly affect the result.

With the above normalized FP operations model, we have

$$\text{MFLOPS}_{\text{normalized}} = \frac{\text{Normalized number of floating-point operations}}{\text{Execution time in seconds} \times 10^6}$$
$$= \frac{204111836401}{287}$$
$$= 711$$

1.19 No solution provided.

1.20 No solution provided.

1.21 a. No solution provided.

b. The steps of the word-processing workload and their nature are as follows.

1. Load the word-processing program and the document file. [Disk and memory system.]

2. Invoke the spelling checker. [User input via the user interface: typing, mouse movement, etc.]
3. Scan the document for words that may be misspelled. [CPU intensive.]
4. User examines the output from the spelling checker. [User think time.]
5. Request document printing. [User input via the user interface.]
6. Wait for the printing dialog box to appear. [CPU and operating system.]
7. User selects printing options. [User input via the user interface.]
8. System initiates interaction with printer. [Mainly O/S and I/O, buffering and handshaking.]
9. Printer creates output. [Peripheral device intensive.]

Processor improvements will increase the performance only of those steps that spend a significant fraction of their execution time using the CPU. This follows directly from Amdahl's law. For computer systems that are *identical* except for CPU clock speed, we can expect the following: little difference in performance of steps 1, 2, 5, 8, and 9; improvement in steps 3 and 6; and indirect, synergistic improvement in steps 4 and 7 as user think time decreases more than linearly as a result of faster system response. The key observation is that the steps where a faster CPU is a direct or indirect benefit (3, 4, 6, and 7) comprise a total of 7 seconds out of 90 seconds time for the entire workload. Even if steps 3 and 6 could be reduced in time to zero seconds by a faster CPU clock and the maximum user think time reduction (to 0.5 seconds) is achieved, the maximum speedup possible due to a faster CPU is only $\frac{90 \text{ seconds}}{90 \text{ seconds} - 6 \text{ seconds}} = 1.07$.

Clearly, a faster CPU is of very limited value. For this workload the focus of system improvement should be obtaining a faster printer, perhaps with a print spooler, followed by faster networking to the printer and faster disk I/O. These components affect a substantial portion of the total workload time.

c. No solution provided.

1.22 No solution provided.

1.23

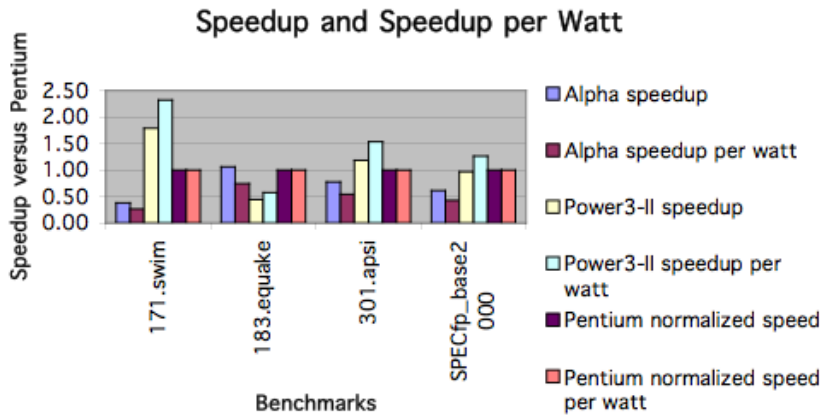

Figure S.5 Graph showing relative performance of three processors (normalized to Pentium). The speedup and speedup per watt are quite different.

1.24 Figure 1.10 shows the addition of three costs to that of the circuitry components, which determines system list price. System power and volume increases that are the unavoidable consequence of a CPU component power consumption increase can be identified in an analogous way. First, consider the effect of CPU power.

An additional watt of CPU power consumption requires an additional watt of power supply capacity. Because a power supply is not 100% efficient, the power input to the supply circuitry must increase by more than 1 watt and the waste energy of conversion, appearing as heat in the components of the supply, will increase. This input power increase of greater than 1 watt is modeled much as the direct costs increase shown in Figure 1.10.

At some level of power delivery to the system, the power supply components will become hotter than their rated maximum operating temperatures if only convection cooling is available. While several active cooling technologies are available, the least expensive is forced air. This requires addition of a power supply fan and the power to run it, with the typical small, rotating fan using about 1 watt of power. This additional power requirement can be modeled analogously to that gross margin of Figure 1.10.

Finally, with increasing CPU power consumption the chip will eventually become too hot without a substantial heat sink and, perhaps, a dedicated fan to assure high airflow for the CPU. (The fan may be designed to run only when the CPU temperature is particularly high, as is the case for many laptop computers). This final "power tax" of a CPU fan for the more power-hungry CPU is modeled as the average discount component of list price (see Figure 1.10).

System volume is affected by the need to house all system components and, for air cooling, to provide adequate paths for airflow. The volume model for CPU

power consumption increases follows the system cost (Figure 1.10) and system power consumption models. The starting point is the basic system electronics volume (motherboard) and the basic power supply.

To provide an additional watt to the CPU requires a higher capacity power supply, and all other things being equal, higher capacity supplies will use physically larger components, increasing volume. If power supply capacity increases sufficiently, then a cooling fan must be added to the supply along with (possibly) internal airflow paths to provide cooling. This increases power supply volume. Finally, a hot CPU chip may require a dedicated large heat sink and fan, further increasing system volume. A CPU heat sink can easily occupy 100 or more times the volume of the packaged CPU chip alone.

The bottom line is that each additional watt of CPU power consumption has a definite impact on system size due to larger and more numerous components and on system noise due to new and/or increased flows for forced air cooling. Generally these effects are viewed negatively in the marketplace. Volume is an important characteristic, as evidenced by the rapid replacement of CRT monitors with LCD displays in business environments where personnel workstations (cubicles) can be reduced in area saving office rent. Reduced computer noise is generally viewed favorably by users.

1.25 If the collection of performance values are viewed as a set of orthogonal vectors in hyperspace, then those vectors define a right hyperprism with one vertex at the origin. The geometric mean of those values is the length of the side of a hypercube having the same volume as the hyperprism.

The advantage of using total execution time is ease of computing the summary metric. The advantage of using weighted arithmetic mean of the execution times with weights relative to a comparison platform is that this takes into account the relative importance of the individual benchmarks in making up the aggregate workload. The advantage of using the geometric mean of the speed ratios is that any machine from a collection is equally valid as the normative platform and the result appears to allow for simple scaling to predict performance of new programs on the collection of machines.

The disadvantage of using total execution time is that the actual workload may not be well modeled. The disadvantage of using weighted arithmetic mean is that the results are affected by machine design details and program input size. The disadvantage of using geometric mean is that is does not track execution time, our gold standard for performance measurement.

1.26 Whether performance changes and what might be concluded will depend on the specific hardware and software platform chosen for the experiment.

What can be concluded about SPEC2000 is the following. Suppose that there are SPEC2000 results for a platform of interest to you. Your computational needs likely depend on programs, your own and/or those from a vendor, that are not compiled as aggressively as the benchmarks. Thus, the performance you enjoy from this computer is likely to be less than that reported in the SPEC2000 results. Further, the performance ranking of a variety of computers on your workload

may be quite different than that for the same set of machines on the SPEC2000 benchmarks.

1.27 A discussion answer can take many forms. Some points that should be addressed in discussing the disadvantages include: inability of this technique to address issues of user interface on user think time, the unrepresentativeness of the benchmark workload where each option is equally likely versus real-world use where some program options are quite rare. Some points that should be addressed in the discussion of penalized techniques include: the effect of testing each option once on caching in its many specific forms and compiling to take advantage of known likely paths of execution.

1.28 Generally, the sequence of topics within each chapter starts with the most widely applicable techniques and proceeds to more specialized ideas that provide performance benefit for a smaller fraction of the execution time. Chapters 3 through 7 and Appendix A exhibit this structure most clearly. There the authors point out in a number of specific instances that techniques providing speedup to large fractions of execution are ubiquitous or nearly ubiquitous. Starting with these topics brings the most significant information to the fore.

When the techniques are organized according to their Amdahl's Law speedup and then combined with their deployment costs, the result is a particularly clear view of the design rationale of technically successful computers. Performance and cost-performance information is also the source of insight into the technically-favored paths of future computer developments.

Finally, at the highest level this book is a quantitative approach to computer architecture. Quantitative description requires a metric. Here the universal metric is performance. This places the performance-measuring Amdahl's Law squarely in the role of an organizing principle.

Chapter 2 Solutions

2.1 a. Refer to Figure 2.2 for the instruction sequences.

Stack. This code contains three addresses, A, B, and C, which are all memory addresses. The code size is four opcodes, three memory addresses, and no register addresses for a total of 224 bits.

Accumulator. This code contains three addresses, A, B, and C, which are all memory addresses. The code size is three opcodes, three memory addresses, and no register addresses for a total of 216 bits.

Register (register-memory). There are three memory addresses and four register addresses in this code. Two registers are each referred to twice. The code size is three opcodes plus four register addresses plus three memory addresses for a total of 240 bits.

Register (load-store). This code contains the most addresses, nine, but the majority of them are the smaller-sized register addresses. There are six regis-

ter addresses and three memory addresses. With four instructions, the code size totals 260 bits.

b. Assume that each variable of A, B, C, D, and E is of size *n* bytes. Figures S.6 through S.9 show the code for each of the architectures.

Stack Code	Comment	Code size (bits)	Data size (bytes)
Push A	load from memory	72	n
Push B		72	n
Add	operands A and B destroyed	8	0
Pop C	save result to memory	72	n
Push E		72	n
Push A	overhead instruction; reloading A onto stack	72	n
Sub	operands A and E destroyed; order of operands on stack determines which is minuend and subtrahend	8	0
Pop D		72	n

Figure S.6 Stack architecture code. The total code size is 448 bits (56 bytes), and the total data moved to or from memory is 6*n* bytes. There is 1 overhead instruction and 1 overhead data operand movement.

Accumulator Code	Comment	Code size (bits)	Data size (bytes)
Load A		72	n
Add B	operands A and B destroyed	72	n
Store C		72	n
Load A	overhead instruction; reloading A	72	n
Sub E	operands A and E destroyed	72	n
Store D		72	n

Figure S.7 Accumulator architecture code. The total code size is 432 bits (54 bytes), and the total data moved to or from memory is 6*n* bytes. There is 1 overhead instruction and 1 overhead data operand movement.

Register-Memory Code	Comment	Code size (bits)	Data size (bytes)
Load R1,A		78	n
Add R1,B	operands A and B destroyed	78	n
Store C, R1		78	n
Load R1,A	overhead instruction; reloading A	78	n
Sub R1,E	operands A and E destroyed	78	n
Store D,R1		78	n

Figure S.8 Register-memory architecture code. The total code size is 468 bits (about 59 bytes), and the total data moved to or from memory is 6*n* bytes. There is 1 overhead instruction and 1 overhead data operand movement.

Load-Store Code	Comment	Code size (bits)	Data size (bytes)
Load R1,A		78	n
Load R2,B		78	n
Add R3,R2,R1	no operands destroyed	26	0
Store C,R3		78	n
Load R4,E		78	n
Sub R5,R1,R4	no operands destroyed	26	0
Store D,R5		78	n

Figure S.9 Load-store architecture code. The total code size is 442 bits (about 55 bytes), and the total data moved to or from memory is only 5*n* bytes. There are no overhead instructions and no overhead data operand movements.

2.2 **Stack advantage.** Stack instructions have the smallest encoding because no operand or result locations are named.

Stack disadvantages. Operands must be in the correct order on the stack or else it must be possible to "convert" the initial result. The order of subtraction can be changed by negating the result because A − B = − (B − A). However, there is no simple way to transform the result of "shift left A by B bit positions" to "shift left B by A bit positions." So, in general, operands must be in the correct order. An instruction to exchange the top two stack elements would be handy.

Accumulator advantage. Smaller instruction encoding because only one operand location is named.

Accumulator disadvantages. Operands must be in the correct order; that is, the correct operand must be in the accumulator. A swap instruction to exchange the accumulator contents with an operand location would be useful.

Load-store advantage. Because operands are in registers, operand order can always be as needed without any exchanges.

Load-store disadvantage. Larger instruction encoding size.

2.3 a.

43	4F	4D	50	55	54	45	52
C	O	M	P	U	T	E	R

b.

45	52	55	54	4D	50	43	4F
E	R	U	T	M	P	C	O

c. 4F4D, 5055, and 5455. Other misaligned 2-byte words would contain data from outside the given 64 bits.

d. 52 55 54 4D, 55 54 4D 50, and 54 4D 50 43. Other misaligned 4-byte words would contain data from outside the given 64 bits.

2.4 a. Accumulator architecture code:

```
Load B          ;Acc ← B
Add C           ;Acc ← Acc + C
Store A         ;Mem[A] ← Acc
Add C           ;Acc ← "A" + C
Store B         ;Mem[B] ← Acc
Negate          ;Acc ← − Acc
Add A           ;Acc ← "− B" + A
Store D         ;Mem[D] ← Acc
```

Memory-memory architecture code:

```
Add A, B, C   ;Mem[A] ← Mem[B] + Mem[C]
Add B, A, C   ;Mem[B] ← Mem[A] + Mem[C]
Sub D, A, B   ;Mem[D] ← Mem[A] − Mem[B]
```

Stack architecture code: (TOS is top of stack, NTTOS is the next to the top of stack, and * is the initial contents of TOS)

```
Push B          ;TOS ← Mem[B], NTTOS ← *
Push C          ;TOS ← Mem[C], NTTOS ← TOS
Add             ;TOS ← TOS + NTTOS, NTTOS ← *
Pop A           ;Mem[A] ← TOS, TOS ← *
Push A          ;TOS ← Mem[A], NTTOS ← *
Push C          ;TOS ← Mem[C], NTTOS ← TOS
Add             ;TOS ← TOS + NTTOS, NTTOS ← *
Pop B           ;Mem[B] ← TOS, TOS ← *
Push B          ;TOS ← Mem[B], NTTOS ← *
Push A          ;TOS ← Mem[A], NTTOS ← TOS
Sub             ;TOS ← TOS − NTTOS, NTTOS ← *
Pop D           ;Mem[D] ← TOS, TOS ← *
```

Load-store architecture code:

```
Load    R1,B          ;R1 ← Mem[B]
Load    R2,C          ;R2 ← Mem[C]
Add     R3,R1,R2      ;R3 ← R1 + R2 = B + C
Add     R1,R3,R2      ;R1 ← R3 + R2 = A + C
Sub     R4,R3,R1      ;R4 ← R3 − R1 = A − B
Store   A,R3          ;Mem[A] ← R3
Store   B,R1          ;Mem[B] ← R1
Store   D,R4          ;Mem[D] ← R4
```

2.5 The total of the integer average instruction frequencies in Figure 2.32 is 99%. The exercise states that we should assume the remaining 1% of instructions are ALU instructions that use only registers, and thus are 16 bits in size.

a. Multiple offset sizes are allowed, thus instructions containing offsets will be of variable size. Data reference instructions and branch-type instructions contain offsets; ALU instructions do not and are of fixed length. To find the average length of an instruction, the frequencies of the instruction types must be known.

The ALU instructions listed in Figure 2.32 are add (19%), sub (3%), compare (5%), cond move (1%), shift (2%), and (4%), or (9%), xor (3%), and the unlisted miscellaneous instructions (1%). The cond move instruction moves a value from one register to another and, thus, does not use an offset to form an effective address and is an ALU instruction. The total frequency of ALU instructions is 47%. These instructions are all a fixed size of 16 bits. The remaining instructions, 53%, are of variable size.

Assume that a load immediate (load imm) instruction uses the offset field to hold the immediate value, and so is of variable size depending on the magnitude of the immediate. The data reference instructions are of variable size and comprise load (26%), store (10%), and load imm (2%), for a total of 38%.

The branch-type instructions include cond branch (12%), jump (1%), call (1%), and return (1%), for a total of 15%.

For instructions using offsets, the data in Figure 2.42 provides frequency information for offsets of 0, 8, 16, and 24 bits. Figure 2.42 lists the cumulative fraction of references for a given offset magnitude. For example, to determine what fraction of branch-type instructions require an offset of 16 bits (one sign bit and 15 magnitude bits) find the 15-bit magnitude offset branch cumulative percentage (98.5%) and subtract the 7-bit cumulative percentage (85.2%) to yield the fraction (13.3%) of branches requiring more than 7 magnitude bits but less than or equal to 15 magnitude bits, which is the desired fraction. Note that the lowest magnitude at which the cumulative percentage is 100% defines an upper limit on the offset size. For data reference instructions a 15-bit magnitude captures all offset values, and so an offset of larger than 16 bits is not needed. Possible data-reference instruction sizes are 16, 24, and 32 bits. For branch-type instructions 19 bits of magnitude are necessary to handle all offset values, so offset sizes as large as 24 bits will be used. Possible branch-type instruction sizes are 16, 24, 32, and 40 bits.

Employing the above, the average lengths are as follows.

$$\begin{aligned}
\text{ALU length} &= 16 \text{ bits} \\
\text{Data-reference length} &= 16 \times 0.304 + 24 \times (0.669 - 0.304) + 32 \times (1.0 - 0.669) \\
&= 24.2 \text{ bits} \\
\text{Branch-type length} &= 16 \times 0.001 + 24 \times (0.852 - 0.001) + 32 \times (0.985 - 0.852) \\
&= 25.3 \text{ bits} \\
\text{Average length} &= 16 \times 0.47 + 24.2 \times 0.38 + 25.3 \times 0.15 \\
&= 20.5 \text{ bits}
\end{aligned}$$

b. With a 24-bit fixed instruction size, the data reference instructions requiring a 16-bit offset and the branch instructions needing either a 16-bit or 24-bit offset cannot be performed by a single instruction. Several alternative implementations are or may be possible.

First, a large-offset load instruction might be eliminated by a compiling optimization. If the value was loaded earlier it may be possible, by placing a higher priority on that value, to keep it in a register until the subsequent use. Or, if the value is the stored result of an earlier computation, it may be possible to recompute the value from operands while using no large-offset loads. Because it might not be possible to transform the program code to accomplish either of these schemes at lower cost than using the large-offset load, let's assume these techniques are not used.

For stores and branch-type instructions, compilation might be able to rearrange code location so that the store effective address or target instruction address is closer to the store or branch-type instruction, respectively, and thus require a smaller offset. Again, let's assume that this is not attempted.

Thus, code needing an offset greater than 8 signed magnitude bits and written with 24-bit fixed-size instructions will use additional instructions to change the program counter in multiple steps. If adding the available offset to the low-order PC register bits is all that is available, then to achieve an offset with a 15-bit magnitude requires adding up to 2^8 8-bit offsets (only 7 magnitude bits each) to the initial PC value. Clearly, this is beyond tedious and performance-robbing.

To avoid this performance penalty we assume that there are two new jump instructions to help with changing the PC. They are JUMP_Shift7 and JUMP_Shift14. These instructions add an 8-bit offset not at the low-order bit positions, but rather at a position shifted left by 7 and 14 bits, respectively, from the least significant bit. During the addition low order bit positions are filled with 7 or 14 zeros. With these two instructions and the original JUMP with unshifted offset, offsets of 8, 15, or 22 total bits including sign can be accomplished piece wise. First an 8-bit offset is added to the PC using JUMP. Next, another 8-bit offset is added to the PC using JUMP_Shift7. Finally, a third 8-bit offset is added to the PC using JUMP_Shift14. If the needed signed magnitude offset can be represented in 8 bits then the original instructions are sufficient. If up to 15 bits are needed for an offset, then a JUMP_Shift7 precedes the original instruction, moving the PC to within the range that can be reached by an 8-bit signed magnitude offset. To keep program instruction layout in memory more sequential a JUMP_Shift7 can be used to restore the PC to point of the next sequential instruction location following the first JUMP_Shift7. Similarly, if up to 22 offset bits are needed, then a JUMP_Shift7 followed by a JUMP_Shift14 changes the PC value to within the range of an 8-bit offset of the intended address. Analogously, a JUMP_Shift14 and a JUMP_Shift7 would follow the original instruction to return the PC to the next sequential instruction location.

From Figure 2.42 no data reference instruction requires more than a 15-bit offset, so JUMP_Shift14 is not needed. JUMP_Shift7 instruction pairs are needed for offsets greater than 7 magnitude bits but less than or equal to 15 bits, or 100% – 66.9% = 33.1% of data reference instructions. Similarly, JUMP_Shift14 and JUMP_Shift7 pairs are needed for 100% – 98.1% = 1.9% of branches, and JUMP_Shift7 pairs alone are needed for 98.1% – 85.2% = 12.9% of branches. Thus, the number of fetched instructions increases by a factor of

$$(2)(0.331)(0.38) + [(2)(0.129) + (4)(0.019)](0.15) = 30.2\%$$

The number of bytes fetched on a per-instruction basis for the byte-variable-sized instructions of part (a) is

$$(20.5 \text{ bits/instr})((1/8)\text{byte/bit}) = 2.56 \text{ bytes/instr.}$$

Because extra instructions must be added to the original code, we compare the number of bytes fetched for the fixed length 24-bit instructions on a per-original-instruction basis. The amount is

$$(1 + 0.302)(24 \text{ bits/instr})((1/8) \text{ bytes/bit}) = 3.91 \text{ bytes/original instr.}$$

c. With a 24-bit fixed offset size, the ALU instructions are 16 bits long and both data reference and branch instructions are 40 bits long. The number of instructions fetched is the same as for byte-variable-sized instructions, so the per-instruction bytes fetched is

$$(16 \text{ bits/instr})(0.47) + (40 \text{ bits/instr})(0.38 + 0.15) = 3.59 \text{ bytes/instr}.$$

This is less that the bytes per original instruction fetched using the limited 8-bit offset field instruction format. Having an adequately sized offset field to avoid frequent special handling of large offsets is valuable.

2.6 This exercise analyzes the trade-off between adding a new instruction that decreases the number of instructions executed at the cost of increasing the clock cycle by 5%.

a. Because the new design has a clock cycle time equal to 1.05 times the original clock cycle time, the new design must execute fewer instructions to achieve the same execution time. For the original design we have

$$\text{CPU Time}_{\text{old}} = \text{CPI}_{\text{old}} \times \text{CC}_{\text{old}} \times \text{IC}_{\text{old}}.$$

For the new design the equation is

$$\begin{aligned}\text{CPU Time}_{\text{new}} &= \text{CPI}_{\text{new}} \times \text{CC}_{\text{new}} \times \text{IC}_{\text{new}} \\ &= \text{CPI}_{\text{old}} \times (\text{CC}_{\text{old}} \times 1.05) \times (\text{IC}_{\text{old}} - R)\end{aligned}$$

where the CPI of the new design is the same as the original (per the exercise statement), the new clock cycle is 5% longer, and the new design executes R fewer instructions than the original design. To find out how many loads must be removed to match the performance of the original design set the above two equations equal and solve for R:

$$\begin{aligned} \text{CPI}_{\text{old}} \times \text{CC}_{\text{old}} \times \text{IC}_{\text{old}} &= \text{CPI}_{\text{old}} \times (\text{CC}_{\text{old}} \times 1.05) \times (\text{IC}_{\text{old}} - R) \\ R &= 0.048 \ \text{IC}_{\text{old}} \end{aligned}$$

Thus the instruction count must decrease by 4.8% overall to achieve the same performance, and this 4.8% is comprised entirely of loads. Figure 2.32 shows that 25.1% of the gcc instruction mix is loads so (4.8%/25.1%) = 19.0% of the loads must be replaced by the new register-memory instruction format for the performance of the old and new designs to be the same. If more than 19% of the loads can be replaced then performance of the new design is better.

b. Consider the code sequence

```
LOAD        R1,0(R1)
ADD         R1,R1,R1
```

The result written to R1 is MEM[0+R1] + MEM[0+R1]. However, the register-memory form of this code sequence is

```
ADD         R1,0(R1)
```

The result written to R1 in this case is R1 + MEM[0+R1] which is not the same unless the initial value in R1 just happens to be equal to the value at the memory location with address [0+R1]. In general the values will not be equal, and a compiler would not be able to use the new instruction form in this case.

2.7 No solution provided.

2.8 The point of this exercise is to highlight the value of compiler optimizations (see Exercise 2.9). In this exercise registers are not used to hold updated values; values are stored to memory when updated and subsequently reloaded. Because all the addresses of all the variables (including all array elements) can fit in 16 bits, we can use immediate instructions to construct addresses. Figure S.10 shows one possible translation of the given C code fragment.

```
ex2_8:      DADD        R1,R0,R0        ;R0 = 0, initialize i = 0
            SW          2000(R0),R1     ;store i

loop:       LD          R1,2000(R0)     ;get value of i
            DSLL        R2,R1,#3        ;R2 = word offset of B[i]
            DADDI       R3,R2,#5000     ;add base address of B to R2

            LD          R4,0(R3)        ;load B[i]
            LD          R5,1500(R0)     ;load C

            DADD        R6,R4,R5        ;B[i] + C

            LD          R1,2000(R0)     ;get value of i
            DSLL        R2,R1,#3        ;R2 = word offset of A[i]
            DADDI       R7,R2,#0        ;add base address of A to R2
            SD          0(R7),R6        ;A[i] ← B[i] + C

            LD          R1,2000(R0)     ;get value of i
            DADDI       R1,R1,#1        ;increment i
            SD          2000(R0),R1     ;store i

            LD          R1,2000(R0)     ;get value of i
            DADDI       R8,R1,#-101     ;is counter at 101?
            BNEZ        R8,loop         ;if not 101, repeat
```

Figure S.10 MIPS code to implement the C loop without using registers to hold updated values for future use or to pass values to a subsequent loop iteration.

The number of instructions executed dynamically is the number of initialization instructions plus the number of instructions in the loop times the number of iterations:

$$\text{Instructions executed} = 2 + (16 \times 101) = 1618$$

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

$$\text{Memory-data references executed} = 0 + (8 \times 101) = 808$$

Since MIPS instructions are 4 bytes in size, code size is the number of instructions times 4:

$$\text{Instruction bytes} = 4 \times 18 = 72$$

2.9 This exercise repeats Exercise 2.8 but asks for basic compiling optimizations to be applied. Specifically, values in registers should be reused whenever possible and loop invariant code should be moved outside of the loop.

Note that the value of the loop index variable is now stored only after the last iteration of the loop. Again, because the addresses of all variables (including all

array elements) can fit in 16 bits, we can use immediate instructions to construct addresses. Figure S.11 shows one possible translation of the given C code fragment

```
ex2_9:        DADDI    R1,R0,#0          ;R0 = 0, initialize i = 0
              DADDI    R3,R0,#0          ;base address of A

              DADDI    R4,R0,#5000       ;base address of B
              LD       R5,1500(R0)       ;load value of C
loop:         DSLL     R2,R1,#3          ;R2 = word offset for array
                                           elements
              DADDI    R6,R2,#5000       ;compute address of B[i]

              LD       R7,0(R6)          ;load B[i]
              DADD     R8,R7,R5          ;compute B[i] + C

              DADDI    R9,R2,#0          ;compute address of A[i]
              SD       0(R9),R8          ;A[i] ← B[i] + C

              DADDI    R1,R1,#1          ;increment i
              DADDI    R10,R1,#–101      ;is counter at 101?
              BNEZ     R10,loop          ;if counter not equal to 101,
                                           repeat
out_of_loop:  SD       2000(R0),R10      ;store final value of i
```

Figure S.11 Optimized MIPS code to implement the C loop.

The number of instructions executed dynamically is the number of initialization instructions plus the number of instructions in the loop times the number of iterations plus the cleanup instructions:

$$\text{Instructions executed} = 4 + (9 \times 101) + 1 = 914$$

This is a 43% reduction from the unoptimized code of Exercise 2.8.

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

$$\text{Memory-data references executed} = 1 + (2 \times 101) + 1 = 204$$

This is a whopping 75% reduction from the unoptimized code of Exercise 2.9, and because memory accesses are often slow (see Chapter 5) this change may result in a substantial reduction in execution time.

Since MIPS instructions are 4 bytes in size, code size is the number of instructions times 4:

$$\text{Instruction bytes} = 4 \times 14 = 56$$

2.10 The key to solving this exercise is realizing that for a load-store instruction set architecture every executed instruction requires one memory read for the pipeline to fetch that instruction and, further, each load and store instruction makes one additional memory read or write access, respectively, to move data between memory and a register. To use dynamic instruction mix data to analyze memory read and write accesses, aggregate the frequencies into three categories: all instructions (each fetch of an instruction is a read), loads (source of data reads), and stores (source of data writes). A memory access is any read or write, including those performed for instruction fetch; a data access is a read or write performed by a load or store instruction.

For the average instruction mix data in Figure 2.32, 26% of the instructions are loads and 10% are stores. The shown frequencies for all instructions total only to 99% because of rounding of rare instructions frequencies to 0%, but must total to 100%.

The exercise statement says to ignore the size of a datum when counting accesses so that access counting is straightforward: one datum requires one memory access. In reality, large data objects typically require multiple memory accesses.

To reduce the size of the equations for the desired memory access percentages, assume the following notation:

IC = number of dynamically executed instructions

f_F = frequency of instruction fetch = 100% by definition

f_L = frequency of load instructions = 26% from given information

f_S = frequency of store insrucitons = 10% from given information

A = number of memory accesses for a given event = 1

By defining the memory access categories as fetch, load, and store, each event generates the same number of memory accesses, simplifying the accounting and allowing the single value of A for all events. An equivalent, but more complicated formulation could use the three categories: non-load/store instruction (one memory read access), load instructions (two memory read accesses), and store instructions (one memory read access and one memory write access).

Now we have the following expressions:

$$\text{Percentage of all memory accesses for data} = \frac{\text{Number of data accesses}}{\text{Number of memory accesses}}$$
$$= \frac{(f_L + f_S) \times A \times \text{IC}}{(f_F + f_L + f_S) \times A \times \text{IC}}$$
$$= \frac{26 + 10}{100 + 26 + 10}$$
$$= 26.5$$

$$\text{Percentage of all data accesses that are reads} = \frac{\text{Number of read data accesses}}{\text{Number of data accesses}}$$
$$= \frac{f_L \times A \times \text{IC}}{(f_L + f_S) \times A \times \text{IC}}$$
$$= \frac{26}{26 + 10}$$
$$= 72.2$$

$$\text{Percentage of all memory accesses that are reads} = \frac{\text{Number of read accesses}}{\text{Number of memory accesses}}$$
$$= \frac{(f_F + f_L) \times A \times \text{IC}}{(f_F + f_L + f_S) \times A \times \text{IC}}$$
$$= \frac{100 + 26}{100 + 26 + 10}$$
$$= 92.6$$

2.11 The first challenge of this exercise is to obtain the instruction mix. The instruction frequencies in Figure 2.32 must add to 100, although gap and gcc add to 100.2 and 99.5 percent, respectively, because of rounding error. Because each each total must in reality be 100, we should not attempt to scale the per instruction average frequencies by the shown totals of 100.2 and 99.5. However, in computing the average frequencies to one significant digit to the right of the decimal point, we should be careful to use an unbiased rounding scheme so that the total of the averaged frequencies is kept as close to 100 as possible. One such scheme is called round to even, which makes the least significant digit always even. For example, 0.15 rounds to 0.2, but 0.25 also rounds to 0.2. For a summation of terms, round to even will not accumulate an error as would, for example, rounding up where 0.15 rounds to 0.2 and 0.25 rounds to 0.3.

For gap and gcc the average instruction frequencies are shown in Figure S.12.

Instruction	Average of gap and gcc
load	25.8
store	11.8
add	20.0
sub	2.0
mul	0.8
compare	4.4
load imm	3.6
cond branch	10.7
cond move	0.5
jump	0.8
call	1.1
return	1.1
shift	2.4
and	4.4
or	8.2
xor	2.0
other logical	0.2

Figure S.12 MIPS dynamic instruction mix average for gap and gcc.

The exercise statement gives CPI information in terms of four major instruction categories, with two subcategories for conditional branches. To compute the average CPI we need to aggregate the instruction frequencies in Figure S.12 to match these categories. This is the second challenge, because it is easy to mis-categorize instructions. The four main categories are ALU, load/store, conditional branch, and jumps. ALU instructions are any that take operands from the set of registers and return a result to that set of registers. Load/store instructions access memory. Conditional branch instructions must be able to set the program counter to a new value based on a condition. Jump-type instructions set the program counter to a new value no matter what.

With the above category definitions, the frequency of ALU instructions is the sum of the frequencies of the add, sub, mul, compare, load imm (remember, this instruction does not access memory, instead the value to load is encoded in a field within the instruction itself), cond move (implemented as an OR instruction between a controlling register and the register with the data to move to the destination register), shift, and, or, xor, and other logical for a total of 48.5%. The frequency of load/store instructions is the sum of the frequencies of load and store for a total of 37.6%. The frequency of conditional branches is 10.7%. Finally, the frequency of jumps is the sum of the frequencies of the jump-type instructions, namely jump, call, and return, for a total of 3.0%.

Now

$$\text{Effective CPI} = \sum_{\text{categories}} \text{Instruction category frequency} \times \text{Clock cycles for category}$$
$$= (0.485)(1.0) + (0.367)(1.4) + (0.107)((0.6)(2.0) + (1 - 0.6)(1.5)) + (0.03)(1.2)$$
$$= 1.24$$

2.12 This exercise provides an example of how a computer architect might go about analyzing the potential performance impact of a new architectural feature. It explores the effect of adding a new addressing mode that, in some cases, reduces the number of instructions required to perform an effective address calculation in support of a load or store instruction. The change allows for effective addresses that are the sum of two registers and a constant, as compared to the base instruction set capability of forming effective addresses as the sum of a single register and a constant. The first part of the exercise examines the change in program instruction count caused by the instruction set change. The second part evaluates the speedup achieved using the modification.

a. First, we must understand exactly how the proposed instruction set modification changes the instruction count. Look again at the code fragment in the exercise statement.

```
ADD         R1,R1,R2       ; R1 = R1 + R2
LW          Rd,100(R1)     ; load from 100 + R1
```

The ADD instruction is used to compute an intermediate value of the effective address for the LW, then the LW instruction adds the final amount of 100 to produce the intended address. For the new MIPS with the index addressing mode the code fragment would be

```
LW          Rd,100(R1,R2)  ; load from 100 + R1 + R2
```

In this code, the ADD has been folded into the load (or store) instruction. To make this possible some of the bits of the constant offset field of the original displacement load (or store) instruction must instead be used to encode the name of the second register. The remaining offset bits can encode a reduced range of offset values. This is one of the tradeoffs of the new addressing mode and is the reason that some ADD/LW sequences may not be able to use this new instruction form.

The exercise statement says that 10% of the displacement loads and stores in the original code can use the new form. From Figure 2.32, these loads and stores comprise 26% and 10% of the average instruction mix, respectively. Thus, assuming that MIPS executes $IC_{original}$ instructions, then the ratio of instructions executed by the enhanced MIPS to the original MIPS is

$$\frac{IC_{enhanced}}{IC_{original}} = \frac{IC_{original} - IC_{original}[(10\% \times 26\%) + (10\% \times 10\%)]}{IC_{original}}$$
$$= 0.964$$

The enhanced MIPS executes 3.6% fewer instructions than the original MIPS.

b. While the new addressing mode reduces the total instruction count, the work done by the new instruction requires addition of three operands to compute the effective address. It is thus reasonable that this might require a slower clock cycle to allow effective address computation to complete. The exercise statement says that the increase is 5%. To determine which version of MIPS is faster, go back to our gold standard of performance, CPU time and use that in the speedup equation. One unspecified parameter of the CPU time equation is the difference in CPI, if any, between the version of MIPS. In the absence of information about CPI or that can be used to compute relative CPI for the two machines, we assume that CPI is unchanged. This is not an unreasonable assumption because the fraction of affected instructions, 3.6%, is relatively small. However, for a more precise analysis of the speedup offered by the enhanced machine, CPI should be examined more closely.

Considering all of the above gives the following equation for speedup.

$$\begin{aligned}\text{Speedup} &= \frac{\text{CPI}_{\text{original}} \times \text{CC}_{\text{original}} \times \text{IC}_{\text{original}}}{\text{CPI}_{\text{enhanced}} \times \text{CC}_{\text{enhanced}} \times \text{IC}_{\text{enhanced}}} \\ &= \frac{\text{CPI}_{\text{original}} \times \text{CC}_{\text{original}} \times \text{IC}_{\text{original}}}{(\text{CPI}_{\text{original}})(1.05\ \text{CC}_{\text{original}})(0.964\ \text{IC}_{\text{original}})} \\ &= 0.988\end{aligned}$$

As it turns out, even though the instruction count is lower for the enhanced MIPS, the increase in clock cycle time hurts! The enhanced MIPS is actually 1.2% slower than the original MIPS, implying that this architectural feature is not one to include in a future version of the machine.

There is an important lesson here for computer designers. In the same way that squeezing part of an elastic balloon fails to reduce its total volume because of bulging elsewhere, so to is improving one aspect of the CPU time equation often accompanied by worsening of another aspect. Candidate machine enhancements must be evaluated with respect to time and not just for their effect on one factor of CPU time.

2.13 No solution provided.

2.14 a.

Pixel 1			Pixel 2		
R	G	B	R	G	B
E5	F1	D7	AA	C4	DE
+ 20	+ 20	+ 20	+ 20	+ 20	+ 20
05	11	F7	CA	E4	FE

0511F7 is not brighter and has a significant color shift because the red and green brightness are much less now than the blue brightness. CAE4FE is brighter.

b.

Pixel 1			Pixel 2		
R	G	B	R	G	B
E5	F1	D7	AA	C4	DE
$+_s$ 20	$+_s$ 20	$+_s$ 20	$+_s$ 20	$+_s$ 20	$+_s$ 20
FF	FF	F7	CA	E4	FE

Both results are now brighter.

2.15 Dedicated condition codes have the advantage of requiring no instruction encoding bits. The condition register that an instruction uses is implicit in the operation performed. This minimizes code size. The disadvantage is that the name, or location, for each condition is fixed, so there is a guaranteed dependence between two instructions that use the same condition code. This can limit compiler opportunities for instruction scheduling.

General-purpose condition bits provide name flexibility that the compiler can use to avoid many or most dependences on conditions. The disadvantage is that the instruction must explicitly name the condition register, using some number of instruction encoding bits and possibly increasing code size or limiting the size of another field within the instruction word format.

2.16 No solution provided.

2.17 No solution provided.

2.18 Take the code sequence one line at a time.

```
1. A = B + C                    ;The operands here are given, not computed
                                 by the code, so copy propagation will not
                                 transform this statement.
2. B = A + C                    ;Here A is a computed value, so transform the
                                 code by substituting A = B + C to get
     = B + C + C                ;Now no operand is computed
3. D = A - B                    ;Both operands are computed so substitute
                                 for both to get
     = (B + C) - (B + C + C)    ;Simplify algebraically to get
     = - C                      ;This is a given, not computed, operand
```

Copy propagation has increased the work in statement 2 from one addition to two. It has changed the work in statement 3 from subtraction to negation, possibly a savings. The above suggests that writing optimizing compilers means incorporating sophisticated trade-off analysis capability to control any optimizing steps, if the best results are to be achieved.

2.19 Reasons to increase the number of registers include:

1. Greater freedom to employ compilation techniques that consume registers, such as loop unrolling, common subexpression elimination, and avoiding name dependences.
2. More locations that can hold values to pass to subroutines.
3. Reduced need to store and re-load values.

Reasons not to increase the number of registers include:

1. More bits needed to represent a register name, thus increasing the overall size of an instruction or reducing the size of some other field(s) in the instruction.
2. More CPU state to save in the event of an exception.
3. Increased chip area and increased power consumption.

2.20 No solution provided.

2.21 No solution provided.

2.22 No solution provided.

2.23 No solution provided.

2.24 No solution provided.

2.25 Economic arguments for changing instruction set architecture that might be set forth in an answer include the following.

Desktop: If change provides better support for new functions, such as multimedia, then change may enable growth in market share for that segment.

Server: If the new ISA provides better performance, then the trend toward using open source software for servers reduces the cost of change because re-compilation is possible. If performance is significantly improved, there may be corresponding economic rewards.

Embedded: Change may enhance economically important characteristics for an embedded processor – power consumption and system cost. An example of the economic impact of a new instruction on this second characteristic would be when the new ISA allows more and different I/O or control capability in the base chip, reducing the need for support chips and, hence, system cost.

Economic arguments against changing ISA might include.

Desktop: Incompatibility with existing third-party software vendor application binary executables. When introduced, a new ISA may lack independent software vendor development effort forcing the hardware platform into a market niche (at least initially).

Server: Incompatibility with existing application binary executables. Re-compilation to comparable levels of optimization may be difficult for popular applications that have been tuned for many years on the old ISA.

Embedded: The need to obtain a set of software development tools for the new ISA adds to the cost of changing the hardware for OEMs that might use the new processor.

2.26 Pros and cons that might be presented include the following.

Pros of hardware translation.

1. Binary compatibility with legacy software is assured.
2. Changed underlying ISA is fully transparent to the end user.
3. If the base microprocessor is successful in the marketplace, then there is a revenue stream to fund the design effort for a steady stream of improvements.

Cons of hardware translation.

1. Increased chip area and power consumption.
2. Increased chip design complexity; increase chip testing complexity.

Pros of interpretation and on-the-fly compilation.

1. More flexible; support for multiple external ISAs by the same internal hardware ISA is possible.
2. Compilation of frequently used code segments offers performance greater than fixed hardware translation can achieve.
3. Power consumption can be less due to caching of compiled code segments that then do not consume as much power when subsequently executed if cache look-up is lower power than translation.
4. Because translation is done is software the die size can be smaller.

Cons of interpretation and on-the-fly compilation.

1. New technique; marketplace may not be comfortable with the technique.
2. No existing revenue steam to fund development effort; must succeed in the marketplace quickly or likely starve for cash flow.

Chapter 3 Solutions

3.1 a. The control dependences are the following, where $i \rightarrow j$ means statement j is control dependent on statement i.

Control dependence	Data references allow scheduling the dependent statement before the IF statement?
$1 \rightarrow 2$	yes
$1 \rightarrow 3$	no, could change the IF outcome
$1 \rightarrow 4$	yes
$1 \rightarrow 5$	yes
$1 \rightarrow 6$	no, could change the IF outcome

b. The data dependences are

$1 \rightarrow 3$, WAR dependence on a

$1 \rightarrow 6$, WAR dependence on c

$2 \rightarrow 3$, RAW dependence on d

$3 \rightarrow 7$, WAR dependence on b

$3 \rightarrow 7$, RAW dependence on a

$5 \rightarrow 6$, RAW dependence on f

$5 \rightarrow 7$, RAW dependence on f

The groups of statements that can be issued together are {1}, {2}, {3}, {4,5}, {6}, and {7}.

c. Only b and c are live after the code segment, so values a, d, e, and f are not needed subsequently. Values a and f are needed to compute b and a possible final value of c, so statements 1, 2, 3, 5, 6, and 7 are needed because they are part of the b and c chain of calculation. Statement 4 produces a value that does not contribute to the live values and is itself not live, thus this statement may be deleted.

d. The opportunity to issue statements 4 and 5 together is lost when statement 4 is deleted. Yet this was the only multiple-issue opportunity.

The available ILP is affected by the compiler technology. Compiler optimization may reduce available ILP. The performance improvement due to a hardware design option can be affected, both positively and negatively, by compiler actions.

3.2 a.

Code fragment	Data dependence?	Dynamic scheduling sufficient for out-of-order execution?
`DADDI R1,R1,#4` `LD R2,7(R1)`	True dependence on R1	No. Changing instruction order will break program semantics.
`DADD R3,R1,R2` `S.D R2,7(R1)`	None	Yes
`S.D R2,7(R1)` `S.D F2,200(R7)`	Output dependence may exist	Maybe. If the hardware computes the effective addresses early enough, then the store order may be exchanged.
`BEZ R1,place` `S.D R1,7(R1)`	None	No. Changing instruction order is speculative until the branch resolves.

3.3 a.

	Dependence type	Independent instruction	Dependent instruction	Storage location
1	data	`LD`	`DADD`	R1
2	data	`LD`	`DSUB`	R1
3	data	`LD`	`OR`	R1
4	name, anti dependence	`LD`	`XOR`	R2
5	data	`DADD`	`BNEZ`	R7
6	data	`DSUB`	`DADD`	R8
7	control	`BNEZ`	`DADD`	n/a
8	control	`BNEZ`	`XOR`	n/a

No instruction in the given code is labeled as the branch instruction target, so the target must be elsewhere in the program. The answer “not applicable” or n/a is fine for the storage location involved in a control dependence, as shown

above. However, another way of viewing the dependence is as involving the PC register. The value of the PC as set by the branch instruction passes control to one of two specific instructions and thus serves in the same way as the controlling operand of a conditional move instruction, for example. The difference is that the value of PC cannot be set arbitrarily by a program; it can only be modified by an offset provided by a branch instruction.

b. When the register file can be written in the first half of the clock cycle, then any dependent instruction in the ID stage will be able to read an operand from the WB stage during the second half of the clock cycle and there will be no pipeline hazard for that operand. For forwarding, the result from LD is available at the end of the clock cycle when that instruction is processed by the memory access stage, the fourth stage of five.

From the table in part (a), the dependences numbered 3, 5, and 6 will not become hazards because the needed operand will be written in the first half cycle of the clock cycle in which the value is needed.

Without forwarding hardware, dependence 2 will be a hazard because the new value for R1 will be only at the output of the MEM stage and not available to the ID stage that is processing the DSUB instruction. With forwarding this dependence would not be a hazard.

Dependence 1 is a hazard with or without forwarding because the ID stage needs the new value if R1 for the DADD instruction before the LW instruction completes its memory access in the MEM stage of the pipeline.

Dependences 7 and 8 will be a hazard in the MIPS pipeline that resolves branches in the MEM stage. The BNEZ instruction will not have resolved before both the IF stage will attempt to fetch both the following DADD and XOR instructions. The pipeline must stall until the BNEZ resolves.

In the MIPS pipeline register values are read early, at stage ID, before register values are written in the WB stage. Thus, two instructions with a name antidependence can never require the MIPS pipeline to stall to preserve correct access order. Thus, dependence 4 is not a hazard.

3.4 a. The specific answer depends on the finding for the fraction of die area that current processors devote to other than essential core execution logic and the reliability of that circuitry.

b. Each processor circuit in Figure 2.1 contains the same ALU. There are one or more registers with differing access paths – read and write ports in all instances, plus a direct linkage for pushing and popping in the case of stack registers and an addressing mechanism for selecting individual registers in the case of the register-memory and register-register architectures. There is also a direct path from memory to an ALU input for the register memory architecture. These structural differences are minor and, with the possible exception of the register-memory architecture, do not suggest any difference in achievable clock speeds. (The register-memory architecture might be slowed by the bus connecting memory to the ALU.) Clock speed potential

does not seem to be an explanation for the paucity of stack and accumulator designs.

c. In the following tables showing code sequences and dependence analysis the following abbreviations are used:

tos—top of stack

nttos—next to top of stack

acc—accumulator

Memory locations are denoted by the name of the variable stored, as taken from the given code sequence.

Stack	Accumulator	Load-store
1 Push A ; tos ← A	1 Load A ; acc ← A	1 Load R1,A
2 Push B ; tos ← B, nttos ← tos	2 Add B ; acc ← acc + B	2 Load R2,B
3 Add ; tos ← tos + nttos	3 Store C ; C ← acc	3 Add R3,R2,R1
4 Pop C ; C ← tos	4 Load D ; acc ← D *	4 Store C,R3
5 Push C ; tos ← C	5 Sub C ; acc ← acc – C **	5 Load R4,D
6 Push D ; tos ← D, nttos ← tos	6 Store E ; E ← acc	6 Sub R5,R4,R3
7 Sub ; tos ← tos – nttos		7 Store E,R5
8 Pop E ; E ← tos		

*,** These two instructions for the accumulator architecture may be replaced by the two-instruction sequence

```
Negate ; acc ← – acc
Add D ; acc ← acc + D
```

which takes advantage of the simplicity of generating an additive inverse to save one memory reference. Other type inverses may not be so readily computed, nor in this case is there any advantage gained with respect to dependences in the code.

The dependences in the three code sequences are listed in the following table. Instructions are identified by their number in the table above. All dependences are listed with the arrow pointing to the dependent instruction(s). When there is more than one dependent instruction, set notation is used to give the list. Dependences are identified as data, anti-, or output in type. The storage location involved is listed last.

The stack architecture has a total of 24 dependences and is least able to exploit ILP inherent in the high-level code. The accumulator architecture has a total of 12 dependences and is intermediate in ability. The load-store architecture has far fewer dependences at 6 total and is must more able to take advantage of ILP.

Stack dependences	Accumulator dependences	Load-store dependences
1 → 2, data, tos	1 → 2, data, acc	1 → 3, data, R1
1 → {2,3,5,6,7}, output, tos	1 → {2,4,5}, output, acc	2 → 3, data, R2
2 → 3, data, tos & nttos	2 → 3, data, acc	3 → 4, data, R3
2 → {3,5,6,7}, anti-, tos	2 → {4,5}, output, acc	3 → 6, data, R3
3 → 4, data, tos	3 → {4,5}, anti-, acc	5 → 6, data, R4
3 → {5,6,7,}, output, tos	4 → 5, data, acc	6 → 7, data, R5
4 → {5,6,7,}, anti-, tos	4 → 5, output, acc	
5 → 6, data, tos	5 → 6, data, acc	
5 → {6,7}, output, tos		
6 → 7, data, tos & nttos		
6 → 7, anti-, tos		
7 → 8, data, tos		

d. Stack—only one schedule is possible.

Accumulator—Could do Load B followed by Add A instead of the form in the solution code, but this is really the same schedule. Could change the Load D followed by Sub C in the solution code to Negate Acc followed by Add D but that is not a rescheduling of the instructions, it is a kind of code transformation. So, only one schedule is possible for the accumulator architecture.

Load-store—We can interchange Load R1,A and Load R2,B. We can move Load R4,D to any of four earlier positions. We can more Sub R5,R4,R3 earlier if Load R4,D is moved earlier, but not before Add R3,R1,R2. We can more Store R3,C to any of three later position. There are many possible schedules for the compiler to choose from.

e. Cost and clock speed are not that different among the three architectures. The significant difference among the factors considered is the much greater opportunity for avoiding name dependences provided by the load-store architecture. Compiler code scheduling is critical to achieving high performance from a pipeline implementation. Scheduling plus having an entire register file of data values available for single-clock-cycle access, versus only one or two fast-access values in the accumulator and stack architectures, yields lower execution times for the load-store machine.

3.5 a. In the following table, the column headings (1), (2), (3), and (4) correspond to the four items of information named in the exercise statement. N/A means "not applicable."

b. The base register address name is held in `rs`.

Proof: In the Issue step `rs` is stored in reservation station field `Vj`. In the Execute Load-Store step 1 phase field `A` holds the immediate value operand and is added to `Vj` to compute the effective address, which is stored back in field `A`.

Row	(1)	(2)	(3)	(4)
Issue FP Operation	yes	no	single issue	N/A
Issue Load or Store	yes	no	single issue	N/A
Issue Load only	yes	no	single issue	N/A
Issue Store only	yes	no	single issue	N/A
Execute FP Operation	yes	yes	N/A	structural hazard, not enough functional units
Execute Load-Store step 1	no	no	only 1 load queue only 1 store queue FIFO queues	N/A
Execute Load step 2	no	no	only 1 load queue FIFO queue	N/A
Write result FP Operation or Load	no	no	only 1 CDB	N/A
Write result Store	no	yes	N/A	number of memory ports and the protocol for computing effective addresses

c. The Address Unit performs: Execute Load-Store step 1, wait until `RS[r].Qj = 0 & r` is head of load-store queue, compute `RS[r].A ← RS[r].Vj + RS[r].A`. It may also manage the load-store buffer.

d. The entries for integer ALU operation instructions are identical to those for FP Operation in Issue, Execute, and Write result, with the caveat that `Regs[]` and `RegisterStat[]` refer to the integer register file, not the floating-point register file.

e. Change the Issue phase "Wait until" conditions to be

Station `r` empty & no pending (unresolved) branch

for instructions other than Load and Store, and to

Buffer `r` empty & no pending (unresolved) branch

for Load and Store instructions.

3.6 This problem examines the DAXPY loop on several Tomasulo configurations. For this problem, we assume that a queued instruction in a reservation station may execute in the same clock cycle that the previous instruction writes to the CDB. We also assume that a data dependant instruction begins to execute in the cycle after the data value is broadcasted on the CDB.

a. DAXPY loop on a single-issue Tomasulo MIPS pipeline:

Iteration Number	Instructions		Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D	F2, 0(R1)	1	2	3	First Issue
1	MUL.D	F4, F2, F0	2	4	19	Wait for F2
1	L.D	F6, 0(R2)	3	4	5	
1	ADD.D	F6, F4, F6	4	20	24	Wait for F4
1	S.D	F6, 0(R2)	5	25		Wait for F6
1	DADDUI	R1, R1, #8	6	7	8	
1	DADDUI	R2, R2, #8	7	8	9	
1	DSGTUI	R3, R1, #800	8	9	10	
1	BEQZ	R3, foo	9	11		Wait for R3
2	L.D	F2, 0(R1)	10	12	13	Wait for BEQZ
2	MUL.D	F4, F2, F0	11	19	34	FP MULT Busy
2	L.D	F6, 0(R2)	12	13	14	
2	ADD.D	F6, F4, F6	13	35	39	Wait for F4
2	S.D	F6, 0(R2)	14	40		Wait for F6
2	DADDUI	R1, R1, #8	15	16	17	
2	DADDUI	R2, R2, #8	16	17	18	
2	DSGTUI	R3, R1, #800	17	18	20	CDB Busy
2	BEQZ	R3, foo	18	20		Wait for R3
3	L.D	F2, 0(R1)	19	21	22	Wait for BEQZ
3	MUL.D	F4, F2, F0	20	34	49	FP MULT Busy
3	L.D	F6, 0(R2)	21	22	23	22
3	ADD.D	F6, F4, F6	22	50	54	Wait for F4
3	S.D	F6, 0(R2)	23	55		Wait for F6
3	DADDUI	R1, R1, #8	24	25	26	
3	DADDUI	R2, R2, #8	25	26	27	
3	DSGTUI	R3, R1, #800	26	27	28	
3	BEQZ	R3, foo	27	29		Wait for R3

b. DAXPY loop on a two-issue Tomasulo machine with a fully pipelined FPU:

Iteration Number	Instructions		Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D	F2, 0(R1)	1	2	3	First Issue
1	MUL.D	F4, F2, F0	1	4	19	Wait for F2
1	L.D	F6, 0(R2)	2	3	4	
1	ADD.D	F6, F4, F6	2	20	24	Wait for F4
1	S.D	F6, 0(R2)	3	25		Wait for F6
1	DADDUI	R1, R1, #8	3	4	5	
1	DADDUI	R2, R2, #8	4	5	6	
1	DSGTUI	R3, R1, #800	4	6	7	INT Busy
1	BEQZ	R3, foo	5	7		INT Busy
2	L.D	F2, 0(R1)	6	8	9	Wait for BEQZ
2	MUL.D	F4, F2, F0	6	19	34	FP MULT Busy
2	L.D	F6, 0(R2)	7	9	10	INT Busy
2	ADD.D	F6, F4, F6	7	35	39	Wait for F4
2	S.D	F6, 0(R2)	8	40		Wait for F6
2	DADDUI	R1, R1, #8	8	10	11	INT Busy
2	DADDUI	R2, R2, #8	9	11	12	INT Busy
2	DSGTUI	R3, R1, #800	9	12	13	INT Busy
2	BEQZ	R3, foo	10	14		Wait for R3
3	L.D	F2, 0(R1)	11	15	16	Wait for BEQZ
3	MUL.D	F4, F2, F0	11	34	49	FP MULT Busy
3	L.D	F6, 0(R2)	12	16	17	INT Busy
3	ADD.D	F6, F4, F6	12	50	54	Wait for F4
3	S.D	F6, 0(R2)	14	55		INT RS Full
3	DADDUI	R1, R1, #8	15	17	18	INT RS Full&Busy
3	DADDUI	R2, R2, #8	16	18	19	INT RS Full&Busy
3	DSGTUI	R3, R1, #800	20	21	22	INT RS Full
3	BEQZ	R3, foo	21	22		INT RS Full

c. DAXPY loop on a Tomasulo machine with speculation.

Iteration Number	Instructions		Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D	F2, 0(R1)	1	2	3	First Issue
1	MUL.D	F4, F2, F0	2	4	19	Wait for F2
1	L.D	F6, 0(R2)	3	4	5	
1	ADD.D	F6, F4, F6	4	20	24	Wait for F4
1	S.D	F6, 0(R2)	5	25		Wait for F6
1	DADDUI	R1, R1, #8	6	7	8	
1	DADDUI	R2, R2, #8	7	8	9	
1	DSGTUI	R3, R1, #800	8	9	10	
1	BEQZ	R3, foo	9	11		
2	L.D	F2, 0(R1)	10	11	12	
2	MUL.D	F4, F2, F0	11	19	34	FP MULT Busy
2	L.D	F6, 0(R2)	12	13	14	
2	ADD.D	F6, F4, F6	13	35	39	Wait for F4
2	S.D	F6, 0(R2)	14	40		Wait for F6
2	DADDUI	R1, R1, #8	15	16	17	
2	DADDUI	R2, R2, #8	16	17	18	
2	DSGTUI	R3, R1, #800	17	18	19	
2	BEQZ	R3, foo	18	20		Wait for R3
3	L.D	F2, 0(R1)	19	20	21	
3	MUL.D	F4, F2, F0	20	34	49	
3	L.D	F6, 0(R2)	21	22	23	
3	ADD.D	F6, F4, F6	22	50	54	Wait for F4
3	S.D	F6, 0(R2)	23	55		Wait for F6
3	DADDUI	R1, R1, #8	24	25	26	
3	DADDUI	R2, R2, #8	25	26	27	
3	DSGTUI	R3, R1, #800	26	27	28	
3	BEQZ	R3, foo	27	29		Wait for R3

3.7 a. There are many code sequences that stall a Tomasulo-based CPU. Such code sequences contain two or more instructions that attempt to write their results in the same cycle. One such sequence is the following.

```
frob:  DMUL
       AND
       OR
       LD
```

b. The code characteristic is to have $n + 1$ instructions using distinct function units that all complete in the same clock cycle. Because a machine with n CDBs cannot write more than n results per clock cycle, one of the instructions must stall. To have $n + 1$ instructions complete simultaneously requires an appropriate set of execution initiation times in the function units relative to function unit latency and probably some operand sharing.

3.8 Consider indexing with the high-order address bits, which is the maximally different possibility. With today's 32- and 64-bit address spaces it is rare for the instructions of a program to occupy any more than a tiny fraction of the available memory locations. Typically, then, the high-order n branch instruction address bits are all or nearly all the same throughout the program. Thus, only a very few of the 2^n branch prediction buffer locations would actually be addressed, wasting most of the predictor storage and sacrificing the majority of the performance enhancement potential.

Clearly, the goal is to use all of the available buffer locations so that as many predictors can be maintained as possible. Using n bits from closer to the low-order positions supports this goal. Further, buffer performance is best if each location is on average equally active in providing predictions. The most equal use of locations occurs when they are indexed using the most varied bit patterns, which is the case when the low-order address bits are the index.

3.9 Consider two branches, B1 and B2, that are executed alternately. In the following tables columns labeled P show the value of a 1-bit predictor shared by B1 and B2. Columns labeled B1 and B2 show the actions of the branches. Time increases to the right. T stands for taken, NT for not taken. The predictor is initialized to NT.

a.

	P	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**
	NT	T	T	NT	NT	NT	NT	T	T	T	T	NT	NT	NT	NT	T
Correct prediction?	—	no	—	no	—	yes	—	no	—	yes	—	no	—	yes	—	no

Here, B1 and B2 each alternate taken/not taken. If they each had a 1-bit predictor, each would always be mispredicted. Because a single predictor is shared here, prediction accuracy improves from 0% to 50%.

b.

	P	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**	**P**	**B1**	**P**	**B2**
	NT	T	T	NT	NT	T	T	NT	NT	T	T	NT	NT	T	T	NT
Correct prediction?	—	no	—	no	—	no	—	no	—	no	—	no	—	no	—	no

Here, B1 is always taken, B2 is always not taken, and they are interleaved as in part (a). If each had a 1-bit predictor, each would be correctly predicted after an initial transient. Because of predictor sharing here, accuracy is 0%.

c. If a predictor is being shared by a set of branch instructions, then over the course of program execution set membership will likely change. When a new branch enters the set or an old one leaves the set, the branch action history represented by the state of the predictor is unlikely to predict new set behavior as well as it did old set behavior, which had some time to affect predictor state. The transient intervals following set changes likely will reduce long-term prediction accuracy.

3.10 The table below shows the answer. The prediction used for b1 and b2 is indicated by the boldface font in the table. If the correlation bit, i.e. the previous branch outcome, has the value "Not Taken" then the left prediction is used. If the previous branch outcome is "taken" then the right prediction is used. Because the predictors used are 1-bit, each time there is a misprediction the value of the predictor is changed. Mispredictions are noted in the "b1 action" and "b2 action" columns. For this set of *d*, initial preditor, and correlation values there is a total of 5 mispredictions, worse performance than that shown in Figure 3.13.

d = ?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
1	NT/**NT**	T (miss)	NT/T	NT/**NT**	NT	NT/NT
2	**NT**/T	T (miss)	T/T	NT/**NT**	T (miss)	NT/T
1	T/**T**	T	T/T	NT/**T**	NT (miss)	NT/NT
2	**T**/T	T	T/T	NT/**NT**	T (miss)	NT/T

3.11 a. The simple code fragment has only one branch instruction, so there is never any sharing of the predictor with another branch. Increasing the buffer size only allows more branch history to control predictor selection. At some buffer size there will be separate entries for each branch in the program and there will be enough history to either predict the branch perfectly or reach the fundamental limit of the information contained in the taken/not taken sequence for a given branch. The sequence of decisions made by a perfectly predictable branch can be expressed as a finite amount of information bits regardless of how many times the branch executes. The sequence of decision made by a completely non-predictable branch communicates one bit of information each time the branch executes. The misprediction rate is a measure of the amount of information communicated by the sequence of all branch decisions. That amount of information is a property of the program itself; no branch prediction scheme can reduce misprediction below this information-theoretic amount.

b. Increasing the non-correlating predictor from 4096 to unlimited entries did result in fewer mispredictions. The reduction in misprediction due to increased buffer size alone could only happen if there were predictor sharing among the 4096-entries that was eliminated by the infinite buffer. Thus, these three codes must contain more than 4096 branch instructions.

c. Yes, somewhat. The amount of predictor sharing for these seven benchmarks did not decrease by an amount sufficient to change the misprediction rate by more than the limit of the accuracy of the measurement. This means that there was relatively little sharing with 4096 buffer entries. We can conclude that these benchmarks cannot contain significantly more than 4096 branch instructions or there would be significant sharing with the smaller buffer size.

d. For whatever entry sharing exists an optimizing compiler could take into account the relative low-order address bits of the branches in the code it generates and then seek to adjust the sequence of instructions to reduce the occurrence of matching low-order address bits among branches.

3.12 No solution provided.

3.13 No solution provided.

3.14 To compare the performance of systems with and without a branch-target buffer (BTB) for conditional branches, we determine the speedup of the CPIs for the two designs, CPI_{BTB} and $CPI_{no\ BTB}$. (We might also approach this exercise by figuring out the speedup of the pipeline speedups with and without the BTB—the resulting speedup equation, when simplified, is identical to the equation for the speedup of the CPIs.) The speedup is given by

$$\text{Speedup} = \frac{CPI_{no\ BTB}}{CPI_{BTB}} = \frac{CPI_{base} + Stalls_{no\ BTB}}{CPI_{base} + Stalls_{BTB}} \tag{S.1}$$

From the exercise statement CPI_{base} is 1.0, which, by definition, accounts for everything *except* conditional branches. To complete the solution, we must find the number of stall cycles that are caused by unconditional branches in the machines with and without BTBs. To find the stall terms, $Stalls_{BTB}$ and $Stalls_{no\ BTB}$, we begin with the following expression:

$$\text{Stalls} = \sum_{s \in \text{Stall}} \text{Frequency}_s \times \text{Penalty}_s \tag{S.2}$$

which sums over all stall cases related to branch folding the product of the frequency of the stall case and the penalty.

The value for $Stalls_{no\ BTB}$ follows simply from the exercise statement and Equation B.2:

$$Stalls_{no\ BTB} = 15\% \times 2 = 0.30 \tag{S.3}$$

As the system without a BTB has a fixed two-cycle branch penalty, the stall contribution from branches is simply the product of the frequency of branches and the number of penalty cycles. Computing $Stalls_{BTB}$ is a bit more involved.

To find $Stalls_{BTB}$ we must consider each possible BTB outcome from a conditional branch. There are three cases: first, the branch can miss the BTB; second, the branch can hit the BTB and be correctly predicted; and finally, the branch can hit the BTB and be incorrectly predicted. Figure S.13 summarizes these observations. The frequencies and penalties can be found from the exercise statement and the discussion of BTBs in Section 3.4 of the text. For example, if the BTB hits

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

Figure S.13 A summary of the behavior of a BTB for conditional branches.

and correctly predicts a branch, there is no penalty (because the BTB returns the next PC in time to be used for the fetch of the instruction following the branch). This case occurs with a frequency per instruction given by the frequency of branches (15%) multiplied by the frequency of branches that are in the BTB (90%) multiplied by the frequency of branches that are in the BTB and are predicted correctly (90%).

From Equation B.2 and Figure S.13, we can compute $\text{Stalls}_{\text{BTB}}$:

$$\text{Stalls}_{\text{BTB}} = (1.5\% \times 3) + (12.1\% \times 0) + (1.3\% \times 4) = 0.097$$

This result along with Equation B.3 can be plugged into Equation B.1 to arrive at

$$\text{Speedup} = \frac{\text{CPI}_{\text{base}} + \text{Stalls}_{\text{no BTB}}}{\text{CPI}_{\text{base}} + \text{Stalls}_{\text{BTB}}} = \frac{1.0 + 0.30}{1.0 + 0.097} = 1.2$$

Therefore, adding a BTB for conditional branches makes the DLX pipeline about 20% faster.

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

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

3.16 Note: See part (f) for the sustained CPI figures for each exercise. We assume that reservation stations have enough entries as to not cause conflicts.

a. The DADDIU had to wait for the ALU.

b. Assume that there are two integer functional units.

Iteration Number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First Issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for F0
1	S.D	F4,0(R1)	2	3	9		Wait for F4
1	DADDIU	R1,R1,#-8	2	3		4	
1	BNE	R1,R2,Loop	3	5			Wait for R1
2	L.D	F0,0(R1)	4	6	7	8	BNE Wait
2	ADD.D	F4,F0,F2	4	9		12	Wait for F0
2	S.D	F4,0(R1)	5	6	13		Wait for F4
2	DADDIU	R1,R1,#-8	5	7		9	INT/CDB Busy
2	BNE	R1,R2,Loop	6	10			Wait for R1
3	L.D	F0,0(R1)	7	11	12	13	BNE Wait
3	ADD.D	F4,F0,F2	7	14		17	Wait for F0
3	S.D	F4,0(R1)	8	11	18		Wait for F4
3	DADDIU	R1,R1,#-8	8	12		13	INT Busy
3	BNE	R1,R2,Loop	9	14			Wait for R1

c. Assume that three instructions may issue simultaneously (but the BNE still issues separately).

Iteration Number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First Issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for F0
1	S.D	F4,0(R1)	1	3	9		INT Busy
1	DADDIU	R1,R1,#-8	2	4		5	INT Busy
1	BNE	R1,R2,Loop	3	6			Wait for R1
2	L.D	F0,0(R1)	4	7	8	9	BNE Wait
2	ADD.D	F4,F0,F2	4	10		13	Wait for F0
2	S.D	F4,0(R1)	4	8	14		INT Busy
2	DADDIU	R1,R1,#-8	5	9		10	INT Busy
2	BNE	R1,R2,Loop	6	11			Wait for R1
3	L.D	F0,0(R1)	7	12	13	14	BNE Wait
3	ADD.D	F4,F0,F2	7	15		18	Wait for F0
3	S.D	F4,0(R1)	7	13	19		INT Busy
3	DADDIU	R1,R1,#-8	8	14		15	INT Busy
3	BNE	R1,R2,Loop	9	16			Wait for R1

d. Assume branches are speculated and issue with another instruction. Assume that branch prediction is perfect.

Iteration Number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First Issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for F0
1	S.D	F4,0(R1)	2	3	9		Wait for F4
1	DADDIU	R1,R1,#-8	2	4		5	INT Busy
1	BNE	R1,R2,Loop	3	6			Wait for R1
2	L.D	F0,0(R1)	3	5	6	7	INT Busy
2	ADD.D	F4,F0,F2	4	8		11	Wait for F0
2	S.D	F4,0(R1)	4	6	12		INT Busy
2	DADDIU	R1,R1,#-8	5	7		8	INT Busy
2	BNE	R1,R2,Loop	5	9			Wait for R1
3	L.D	F0,0(R1)	6	7	8	9	
3	ADD.D	F4,F0,F2	6	10		13	Wait for F0
3	S.D	F4,0(R1)	7	8	14		Wait for F4
3	DADDIU	R1,R1,#-8	7	9		10	INT Busy
3	BNE	R1,R2,Loop	8	11			Wait for R1

e. Assume branches are speculated and issue with another instruction. Assume that branch prediction is perfect.

Iteration Number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First Issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for F0
1	S.D	F4,0(R1)	2	3	9		Wait for F4
1	DADDIU	R1,R1,#-8	2	3		4	
1	BNE	R1,R2,Loop	3	5			Wait for R1
2	L.D	F0,0(R1)	3	4	5	6	
2	ADD.D	F4,F0,F2	4	7		10	Wait for F0
2	S.D	F4,0(R1)	4	5	11		Wait for F4
2	DADDIU	R1,R1,#-8	5	6		7	
2	BNE	R1,R2,Loop	5	8			Wait for R1
3	L.D	F0,0(R1)	6	7	8	9	
3	ADD.D	F4,F0,F2	6	10		14	Wait for F0
3	S.D	F4,0(R1)	7	8	15		Wait for F4
3	DADDIU	R1,R1,#-8	7	8		9	
3	BNE	R1,R2,Loop	8	10			Wait for R1

f. The various exercises in this problem show how the number of integer units, issue width, and speculation effect the performance of the machine. When there is multiple issue, there must be enough functional units to handle the additional instructions. However, data dependencies limit the usefulness of multiple issue. In the these exercises there are a number of independent instructions that can be executed together. One should increase the number of integer units as the issue bandwidth increases to address these instructions. There were numerous wasted cycles due to branches. With speculation, the loads were able to start their execute phases earlier. However, this increased competition for the integer unit.

In all of the exercises, the sustained CPIs were 3/5. Five instructions issued in three clock cycles. This issue rate was sustained over three loops. However, this assumes that there are no issue stalls due to full reservation stations. If there are only a couple of slots in the integer reservation station, then we would see issue stalls especially in the exercises that saw numerous occurrences of busy integer units as noted in the comments.

3.17 a. The same data dependences—true, anti-, and output—must be checked for, but without the issue restriction we would expect dependences to be much more common within a fetched pair of instructions.

b. Remember that the given pipeline is a very simple five-stage design with no structural hazards and with instructions taking one clock cycle in each stage. Register names are 5 bits. Checking only for data hazards means just checking instructions i and $i + 1$ in ID for dependence on instructions in EX (cannot forward ALU results in time) and Load instructions in MEM (also cannot forward memory read data in time). ID must check each operand of each instruction in ID with the destination registers of both instructions in EX and any load instructions in MEM. ID must also check the operands of instruction $i + 1$ against the result of instruction i because forwarding cannot prevent a data hazard if there is a dependence. Figure S.14 shows the comparisons needed.

The column and row labels in Figure S.14 are the names that must be brought to comparators in ID. There are nine 5-bit names for a total of 45 bits. The X symbols in Figure S.14 denote the comparisons that must be performed. The count is 18.

	Results				
Operands	**ID_i**	**EX_1**	**EX_2**	**MEM_1**	**MEM_2**
ID_{i+1} Op 1	X	X	X	X	X
ID_{i+1} Op 2	X	X	X	X	X
ID_i Op 1		X	X	X	X
ID_i Op 2		X	X	X	X

Figure S.14 Comparisons required to check for data hazards in two-issue pipeline. ID_x is an instruction in the ID stage; ID_x Op 1 and ID_x Op 2 denote the first and second operands of ID_x. EX_i and MEM_i stand for instructions in the EX and MEM stages. An X means that a comparison is performed.

If issue width is doubled, then there are up to four instructions at each stage. Figure S.15 shows the required comparisons.

Doubling the issue width requires comparison with 19 5-bit names for 95 bits, as shown in Figure S.15. The total comparisons to be performed are 76.

	Results										
Operands	**ID_{i+2}**	**ID_{i+1}**	**ID_i**	**EX_1**	**EX_2**	**EX_3**	**EX_4**	**MEM_1**	**MEM_2**	**MEM_3**	**MEM_4**
ID_{i+3} Op 1	X	X	X	X	X	X	X	X	X	X	X
ID_{i+3} Op 2	X	X	X	X	X	X	X	X	X	X	X
ID_{i+2} Op 1		X	X	X	X	X	X	X	X	X	X
ID_{i+2} Op 2		X	X	X	X	X	X	X	X	X	X
ID_{i+1} Op 1			X	X	X	X	X	X	X	X	X
ID_{i+1} Op 2			X	X	X	X	X	X	X	X	X
ID_i Op 1				X	X	X	X	X	X	X	X
ID_i Op 2				X	X	X	X	X	X	X	X

Figure S.15 Comparisons required to check for data hazards in four-issue pipeline. The notation used is the same as in Figure S.14.

The growth in data hazard checking work in ID is more than linear in issue width.

c. Number of compares $= 2n * n + 2(n-1) + 2(n-2) + \ldots + 2(1)$

$$= 4n^2 + 2\sum_{i=1}^{n-1} i$$

$$= 4n^2 + 2\left[\frac{n(n-1)}{2}\right]$$

$$= 5n^2 - n$$

3.18 The following two code schedules show how the code given to us in the problem would execute on a dynamically scheduled processor (top) and a speculative processor (bottom).

Iteration Number	Instruction		Issues at	Executes	Memory access at	Write CDB at	Comment
1	LD	R2,0(R1)	1	2	3	4	
1	DADDI	R2,R2,#1	2	5	6	7	Wait for R2
1	SD	R2,0(R1)	3	4	8		Wait for R2
1	DADDI	R1,R1,#4	4	6		7	ALU busy
1	BNEZ	R2,LOOP	5	8			Wait for R2
2	LD	R2,0(R1)	6	9	11	12	Branch stall
2	DADDI	R2,R2,#1	7	13		14	Wait for R2
2	SD	R2,0(R1)	8	10	15		LD/SD busy
2	DADDI	R1,R1,#4	9	10		11	
2	BNEZ	R2,LOOP	10	15			Wait for R2
3	LD	R2,0(R1)	11	16	17	18	Branch stall
3	DADDI	R2,R2,#1	12	19		20	Wait for R2
3	SD	R2,0(R1)	13	17	21		LD/SD busy
3	DADDI	R1,R1,#4	14	16		17	
3	BNEZ	R2,LOOP	15	21			Wait for R2

Iteration Number	Instruction		Issues at	Executes	Memory access at	Write CDB at	Commit
1	LD	R2,0(R1)	1	2	3	4	5
1	DADDI	R2,R2,#1	2	5	6	7	8
1	SD	R2,0(R1)	3	4	8		9
1	DADDI	R1,R1,#4	4	6		7	10
1	BNEZ	R2,LOOP	5	8			11
2	LD	R2,0(R1)	6	8	9	10	12
2	DADDI	R2,R2,#1	7	11		12	13
2	SD	R2,0(R1)	8	9	13		14
2	DADDI	R1,R1,#4	9	10		11	15
2	BNEZ	R2,LOOP	10	9			16
3	LD	R2,0(R1)	11	12	13	14	17
3	DADDI	R2,R2,#1	12	15		16	18
3	SD	R2,0(R1)	13	14	17		19
3	DADDI	R1,R1,#4	14	16		17	20
3	BNEZ	R2,LOOP	15	18			21

3.19 Fundamentally, the benefit of speculation must exceed its cost. Formally,

Average speculation benefit (clocks/instr.) × Frequency of correct speculation >
Average speculation cost (clocks/instr.) × (1 – Frequency of correct speculation)

For a given use of speculation the cost, if incorrect, likely does not equal the benefit if correct, thus maintaining two distinct terms in the model equation is appropriate.

3.20 When speculation is correct, it allows an instruction that should execute to execute earlier by reducing or eliminating stalls that would occur if execution were delayed until the instruction was no longer speculative. Early execution of a required instruction has no effect on instruction count or clock cycle. The reduction in stall cycles improves CPI.

When speculation is incorrect, instructions that are not on the path of execution are executed and their results ignored. There is no effect on clock cycle time, but the dynamic instruction count increases. The mix of instructions executed may change and lead to a minor effect on CPI, but the majority of the increase in CPU time will be due to the cycles spent on incorrectly speculated instructions, which is best modeled as an increase in IC.

3.21

	ROB Fields			Committed?
Entry	Instruction	Destination	Value	Yes/no
0	ADD.D	F0	F8 + F8	Yes
1	MUL.D	F2	–	No
2	SUB.D	F4	–	No
0	DADDI	R10	R12 + R12	No
1				
2				

There is no value entry for MUL.D because its 10-cycle latency means it has not yet completed execution. There is no value entry for SUB.D because it is dependent on MUL.D. ROB entry 0 initially held ADD.D but has been overwritten by ADDI; entries 1 and 2 hold their initial data.

3.22 a. The following example shows a segment of code were issued instructions read their operand values directly from the ROB. This example assumes that the divide executes immediately and that the divide takes significantly long to perform than the other FP operations. The register values of F2, F3, F4, and F6 are present in the register file for immediate use by the instructions. While the divide executes, the add and multiply issue. The multiply starts immediately when the divide finishes since the add finishes before the divide.

```
DIV.D  F1, F2, F3
ADD.D  F5, F6, F4
MULT.D F9, F1, F5
```

If the add moves to before the divide (as shown in the code below), then the ROB entry associated with the add retires before the divide finishes. The multiply, already in the reservation station, would have to read the value of F5 from the register file. This datapath does not exist in the reservation station architecture. Unless the rules of commit change, reservation stations require value storage.

```
ADD.D  F5, F6, F4
DIV.D  F1, F2, F3
MULT.D F9, F1, F5
```

b. In order to remove the value fields from the reservation stations, a rule must be added to the commit phase. The ROB entry allocated to an instruction must not be permitted to commit until all issued instructions that are data dependant on that particular allocated space have start to execute. In the example above where the add is first, the add instruction may not commit until the multiply starts to execute. The ROB may start to fill with these lingering instructions. Once filled, issue must stall.

3.23

History Buffer					
Entry	**Busy**	**Instruction**	**State**	**Destination**	**Old Value**
1	no	`L.DF0, 0(R1)`	Commit	`F0`	`X1`
2	no	`MUL.D F4, F0, F2`	Commit	`F4`	`X2`
3	yes	`S.D F4, 0(R1)`	Write Result	*Pending in memory pipeline*	
4	yes	`DADDI R1, R1, #-8`	Write Result	`R1`	`X3`
5	yes	`BNE R1, R2 Loop`	Write Result		
6	yes	`L.D F0, 0(R1)`	Write Result	`F0`	`X4`
7	yes	`MUL.D F4, F0, F2`	Write Result	`F4`	`X5`
8	yes	`S.D F4, 0(R1)`	Write Result	*Pending in memory pipeline*	
9	yes	`DADDI R1, R1, #-8`	Write Result	`R1`	`X6`
10	yes	`BNE R1, R2 Loop`	Write Result		

As mentioned in the problem, the history buffer contains the older values as opposed to the newer values found in the reorder buffer. For example, the L.D instruction in entry 1 has written to the register file. Before doing so, the old value of F0 was read and placed into the history buffer (value denoted by X1). When the loop repeats, X4 contains the value of the F0 as produced by the first L.D instruction. Thus if the BNE in entry 5 mispredicts, X1 is loaded back into F0, X2 is loaded into F4, and X3 is loaded back into R1. The pending store in entry 8 is simply erased from the write buffer if that is the method used to write results into memory.

Several changes have to be made to Figure 3.32. On instruction issue, a slot in the history buffer has to be created. If no slots are available, then issue must stall. If

the instruction modifies the register file, then, on issue, the value of the destination register must be read from the register file and inserted into the proper history buffer entry. Also, the tail is incremented to reflect this addition to the buffer. On completion of instruction's execution phase, the result is written into the register file. If it is a store, it is written into the write buffer (if the machine supports it). On a mispredict, values from the history buffer are written back into the register file from the head to the tail of the buffer. The head of the buffer is incremented when no preceding instructions can cause an exception or a misprediction.

For more information, please see the paper by James Smith and Andrew Pleszkun, entitled "Implementing Precise Interrupts in Pipelined Processors." A reprint of this article is found in *Readings in Computer Architecture* by Hill, Jouppi, and Sohi.

3.24 a. If the average instruction memory access time is $h + mp$ then attempting to fetch one instruction at a time will yield an average fetch rate of 1 instruction per $h + mp$ clock cycles. To achieve an average of c instruction completions per clock requires a matching fetch rate in the steady state, $f = c$, of $f = \frac{k}{h + mp}$ or $k = c(h + mp)$ fetch attempts per clock cycle.

b. The minimum value of k occurs for the minimum values of h, m, and p. This value is $k = 4(1 + (0.01)(10)) = 4.4$ fetches per clock cycle to support 4 completions per clock cycle. The maximum value for k occurs for the maximum values of h, m, and p. This value is $k = 4(2 + (0.1)(100)) = 48$ fetches per clock cycle to support 4 completions per clock cycle in this slow access, high miss rate, high miss penalty environment.

3.25 No solution provided.

3.26 No solution provided.

3.27 No solution provided.

3.28 The essential idea is to look for a needed value on the CDB as well as in the register file at the time of issue. One solution is for the Issue step to examine the register file for operands and to monitor the CDB register address during the first half of the clock cycle. The CDB register address will show which register is being written during that clock cycle (if any) by the CDB. For a register file that can be written in the first half of the clock cycle, the issue stage can identify the location of all available operands during the first half cycle from information available from the register file and the CDB. Then in the second half cycle the Issue step can copy the available values from the register file, even a new value transferred by the CDB in this clock cycle, to the reservation station chose for the current instruction. After this cycle any additional needed operand(s) can be found by monitoring the CDB.

3.29 No solution provided.

3.30 Because all entries in the BTB must be searched on every clock cycle, increasing BTB size quickly increases power consumption. Also, BTB access circuitry may be on the critical path in the fetch stage, determining the minimum clock cycle time.

Consider the effect of other pipeline structures on power consumption and clock cycle time. For the ALU a relatively small fraction of its circuitry is used for any give ALU instruction, so power consumption does not increase as rapidly for a more complex ALU as for a larger BTB. For a register file, only the locations accessed by the instructions currently in the decode or write back steps of processing will be active in a given clock cycle. Further, registers access is fast, with write and read within the same clock cycle readily achieved. While larger register files have more register name decode circuitry time delay, this is a small fraction of the typical clock cycle and not likely to set the minimum clock cycle time. Again, power consumption and clock cycle time effects as a function of register file size growth are probably minor.

To improve the benefits of the BTB while keeping power and clock cycle effects within goals, the general approach is to keep the most valuable entries possible in the BTB of an acceptable size. Poorly predicted branches and a long BTB update process may reduce pipeline performance. Instead of updating the entry for a poorly predicted branch a better strategy might be to mark the entry invalid and hope that another program branch that is more frequently taken will take its place among the active BTB entries.

The number of BTB entries searched simultaneously might be reduced to thus reduce power consumption. With a banked design of even and odd BTB entries, fetch at an even PC value would trigger a search of only entries in the even BTB bank, and similarly on fetch at an odd PC value (odd modulo the instruction alignment amount) would search only the odd BTB bank entries. For a given power budget and BTB access time, this scheme would allow essentially twice as many BTB entries as a unified design.

There are doubtless other design possibilities.

3.31 No solution provided.

Chapter 4 Solutions

4.1 a. Figure S.16 shows pipeline structure.

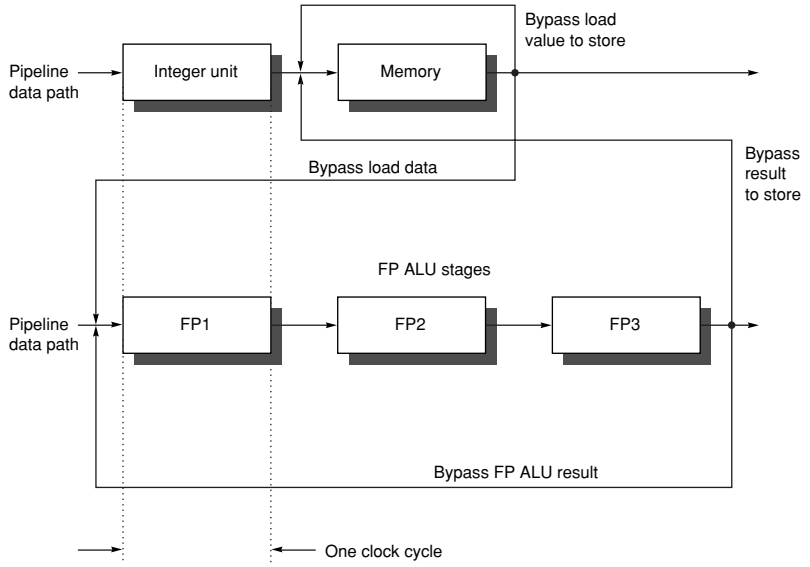

Figure S.16 Pipeline structure implied by Figure 4.1 and full forwarding. An equally valid structure would show the FP ALU as a single stage taking three clock cycles to perform its work.

b. Figure S.17 shows the latency of each unit.

Functional unit	Row of Figure 4.1 that determines the clock cycles needed for this functional unit		
Integer Unit	FP ALU op	Store double	2
	or		
	Load double	FP ALU op	1
Memory	Load double	FP ALU op	1
	or		
	Load double	Store double	0
FP ALU	FP ALU op	Another FP ALU op	3

Figure S.17 Information determining latency of each functional unit.

To derive the answer to part (a) from Figure 4.1 and an assumption that all results can be forwarded, start with the FP ALU to Another FP ALU latency. Given forwarding, a latency of i between instructions using the same FU implies that there are i stages in the FU, if fully pipelined.

Next, if a load can supply a data value to a store with zero latency, then the Memory stage must take only one cycle. Bypassing a load result (a data value read from memory) to the FP ALU with a latency of one means the output of the Memory stage must fall at the end of the clock cycle following the first FP ALU clock cycle. This positions the Memory stage in time with respect to the FP ALU.

Finally, the Integer Unit precedes the Memory stage and computes effective addresses for it. If a load (an integer operation) can supply an FP ALU op with only one cycle of latency, then the EA calculation must start and finish in one cycle. By similar reasoning, if an FP ALU op can supply a store with a two-cycle latency, then the store EA must have been computed in one cycle in the Integer Unit.

4.2 a.

Code fragment	Compiler-found dependence?	Can compiler schedule code non-speculatively?
`DADDI R1,R1,#4` `LD R2,7(R1)`	True dependence on R1	Yes. Change 7(R1) to 11(R1) and then exchange instruction order.
`DADD R3,R1,R2` `S.D R2,7(R1)`	None	Yes
`S.D R2,7(R1)` `S.D F2,200(R7)`	Output data dependence	Maybe. Depends on compiler ability to disambiguate memory addresses.
`BEZ R1,place` `S.D R1,7(R1)`	Control dependence	No. Changing instruction order is speculative.

4.3 a. Unrolled iterations cost 3 cycles each (for the L.D, ADD.D, and S.D instructions) plus two more cycles for loop control (the DADDUI and BNE instructions) to finish up. The not-unrolled, or remainder iterations, cost 6 cycles each, if scheduled (see the Example on page 305). On average, with k iterations in the unrolled loop, there are $\left\lfloor \frac{k}{2} \right\rfloor$ remainder iterations. Thus, the total time to compute n elements (i.e., n original iterations) is

$$3(n - n \bmod k) + 2\left\lfloor \frac{n}{k} \right\rfloor + 6(n \bmod k) \geq 3n + 2\left(\frac{n}{k} - 1\right) + 3(n \bmod k)$$

Minimizing total time means minimizing the above sum as a function of k. The first term, $3n$, is independent of k, so we must minimize

$$2\left(\frac{n}{k} - 1\right) + 3(n \bmod k)\,.$$

Consider now the range of possible amounts of unrolling. The minimum value of k is 2 but this is not an interesting value because it is insufficient to allow scheduling to remove all the stall cycles (see the Example starting on page 305). Three iterations is the minimum unrolling to eliminate stall cycles, so let that be the minimum value for k. The theoretically maximum possible value for k is $\left\lfloor \frac{n}{2} \right\rfloor$ because with any more unrolling even two executions of the unrolled loop would result it performing too many total iterations.

To find the optimal value for k, take the derivative with respect to k and set it equal to zero. Solve for k and check whether that value is a global minima or maxima. For the form of the expressions to be minimized there should not be multiple local minima or maxima. Doing this yields

$$\begin{aligned} 0 &= \frac{d}{dk}\left[2\left(\frac{n}{k}-1\right)+3(n \bmod k)\right] \\ &\approx \frac{d}{dk}\left[\frac{2n}{k}+3\frac{k}{2}\right] \\ &= \frac{3}{2}-\frac{2n}{k^2} \end{aligned}$$

So an estimate for the optimal value of k is $k = \sqrt{\frac{4n}{3}}$. Testing to see which of the two extrema values or this optimum is actually the minimum time value, we have total time for $k = 3$ is

$$\begin{aligned} 3(n - n \bmod 3) + 2\left\lfloor \frac{n}{3} \right\rfloor + 6(n \bmod 3) &\approx 3n - 3\left(\frac{3}{2}\right) + 2\left(\frac{n}{3}-1\right) + 6\left(\frac{3}{2}\right) \\ &= \frac{11n}{3}+\frac{5}{2} \end{aligned}$$

Total time for $k = \left\lfloor \frac{n}{2} \right\rfloor$ is

$$\begin{aligned} 3\left(n - n \bmod \left\lfloor \frac{n}{2} \right\rfloor\right) + 2\left\lfloor \frac{n}{\left\lfloor \frac{n}{2} \right\rfloor} \right\rfloor + 6\left(n \bmod \left\lfloor \frac{n}{2} \right\rfloor\right) &\approx 3n - 3\left(\frac{n}{4}\right) + 2(2) + 6\left(\frac{n}{4}\right) \\ &= \frac{15n}{4}+4 \end{aligned}$$

This time is greater than that for $k = 3$ for positive values of n, so $k = \left\lfloor \frac{n}{2} \right\rfloor$ cannot be the minimum we seek. The total time for $k = \sqrt{\frac{4n}{3}}$ is

$$3\left(n - n \bmod \sqrt{\frac{4n}{3}}\right) + 2\left\lfloor \frac{n}{\sqrt{\frac{4n}{3}}} \right\rfloor + 6\left(n \bmod \sqrt{\frac{4n}{3}}\right)$$

$$\approx 3n - 3\left(\frac{1}{2}\right)\left(\sqrt{\frac{4n}{3}}\right) + 2\left(\frac{1}{\sqrt{\frac{4}{3}}}\right)(\sqrt{n}) + 6\left(\frac{1}{2}\right)\left(\sqrt{\frac{4n}{3}}\right)$$

$$= 3n + \left(3\sqrt{\frac{1}{3}} + \frac{1}{\sqrt{\frac{1}{3}}}\right)\sqrt{n}$$

Comparing $\frac{11n}{3} + \frac{5}{2}$ to $3n + \left(3\sqrt{\frac{1}{3}} + \frac{1}{\sqrt{\frac{1}{3}}}\right)\sqrt{n}$ in the limit as n grows shows the time for $k = 3$ is greater (the larger coefficient on n dominates the $\sqrt{n}$ term), so $k = \sqrt{\frac{4n}{3}}$ is a global minimum and is the theoretically optimum number of unrollings.

b. Examine the unrolled loop code in the Example starting on page 306. The most obvious change is the number of floating-point registers used. The other, more subtle, changes are considered in part (c). The original loop uses two integer registers, one floating-point register to hold a constant, and two more FP registers to perform the computation. Each additional unrolled iteration consumes two more FP registers. With 32 floating-point registers available in MIPS (see Section 2.12, page 130), the limit on unrolling is 15 iterations, which uses 31 FP registers.

   To see if it is possible to make less intensive use of the FP registers and so increase the maximum unrolling possible, look closely at the code in the Examples. One FP register is used to hold the initial array element value loaded, and a second FP register is used to hold the loop iteration computation result, which is then stored. The initial array element value is not needed for any other iteration, so there is no benefit to using a second FP register to hold the result. If the basic computational iteration were changed to be

```
L.D     Fx,0(R1)
ADD.D   Fx,Fx,F2
S.D     Fx,0(R1)
```

   which overwrites the loaded value with the computation result, then each additional unrolling would consume only one additional FP register. With this transformation the unrolling limit is 31 times, using a total of 32 FP registers.

Performance with 15-times unrolling is $15 \times 3 + 2 = 47$ clock cycles per loop iteration, or $47/15 = 3.13$ clock cycles per result. For 31-times unrolling the respective times are $31 \times 3 + 2 = 95$ clock cycles per iteration and $95/31 = 3.06$ clock cycles per result. Unrolling 31 times improves performance by 2%.

Note that further unrolling is possible if registers are re-used within the unrolled loop body. Once a value is stored, the register that held it can be used for a new array element value. Depending on the loop body calculation, a unrolling with register re-use may still allow code scheduling with equal success at reducing stall cycles as when registers are not re-used. However, given the small improvement gained by increasing unrolling from 15 times to 31 times, a transformation re-using registers is probably of little value.

c. While is may be obvious that register file size is the immediate limiting factor in loop unrolling, expanding our search exposes addition aspects of the architecture.

Close examination of the unrolled loop code in the Example starting on page 306 reveals that, in addition to the number of registers used, the code transformation changes (1) the number of effective addresses computed using a given base register value, (2) the magnitude of the loop count and base address register update value, and (3) the code size. Because register files, address offset magnitudes, immediate operand magnitudes, and memory addresses are all of finite size, each limits the number of times that the loop can be unrolled. Which one of these architectural resources is the most significant limiting factor depends on the program.

The displacement addressing used by the L.D and S.D instructions uses a 16-bit field with sign-magnitude format. Thus, with byte addressing, offset magnitudes of up to 32 Kbytes are possible. This corresponds to 4 K double words and a limit on the number of unrollings to 4 K. If the FP register file contained more than 4 K registers, then load and store addressing would limit the amount of unrolling possible.

Next, let's examine loop control. The number of times the loop can be unrolled must be small enough so that the immediate operand field of the DADDUI instruction can hold a value with large enough magnitude so that the correct loop iteration count update and correct update of the base address value in R1 can be accomplished. MIPS has a 16-bit immediate field for the DADDUI instruction. This allows for an offset of 64 K and, thus, with byte-addressing and 8-byte double-word values in the x[i] array, we can step the base address by up to 8 K array elements. So, the unrolling limit from the DADDUI immediate field size is 8 K iterations. (If you are concerned that DADDUI is a double-word add *unsigned* immediate operation and yet the added offset amount must be a negative value, i.e., *signed*, there are two resolutions. First, if program correctness allows, simply reverse the order in which the calculation proceeds through the array elements so that the loop counter and base address register updates can be unsigned (positive) amounts

to add. Alternatively, imagine that the MIPS hardware can actually perform a double-word subtract unsigned immediate, DSUBUI, operation and that the code DADDUI R1,R1,# – CONSTANT is a macro expression for DSUBUI.)

Note that the limit on unrollings that could be achieved with a DADDI instruction (16-bit signed-magnitude immediate value) is 4 K which matches the limit due to displacement addressing. An aspect of good instruction set design is matching the various immediate fields in size and format. This makes the instructions with immediates matched in capability and eliminates the need for a compiler to identify the limiting instruction in code transformations depending on immediate value compensation.

Finally, consider the issue of code size. With a 32-bit address space and byte addressing, the memory space can hold up to 2^{30} instruction words. With each new iteration in the unrolled loop adding just three instructions, we could unroll nearly $2^{30}/3 \approx 3.58 \times 10^8$ times! Of course, that many unrollings would leave no place for the data. In any event, code size places only the most lax of constrains on unrolling.

4.4 To use a single loop, the number of unrolled iterations must divide the number of original loop body iterations. If the original data structures can be padded with extra elements then the number of iterations can be increased until it is divisible by the desired unrolling factor.

The restrictions on using this method are that the addition of padding elements must not change the values in the original results locations and there must be space for the padding elements. A compiler can analyze code structure to determine if this transformation would yield correct results.

4.5 There are six dependences in the C loop presented in the exercise:

1. Anti dependence from `S1` to `S1` on `a`.
2. True dependence from `S1` to `S2` on `a`.
3. Loop-carried true dependence from `S4` to `S1` on `b`.
4. Loop-carried true dependence from `S4` to `S3` on `b`.
5. Loop-carried true dependence from `S3` to `S3` on `b`.
6. Loop-carried output dependence from `S3` to `S3` on `a`.

For a loop to be parallel, each iteration must be independent of all others, which is not the case in the code used for this exercise.

Because dependences 3, 4, and 5 are "true" dependences, they can not be removed through renaming or other such techniques. In addition, as these dependences are loop-carried, they imply that iterations of the loop are not independent. These factors together imply the loop can not be made parallel as the loop is written. By "rewriting" the loop it may be possible to find a loop that is functionally equivalent to the original loop that can be made parallel.

4.6 There are five dependences in the C loop presented in the exercise:

1. Data dependence from S1 to S2 on a[i].
2. Anti-dependence from S1 to S2 on b[i].

3. Loop-carried output dependence from S3 to S1 on a[].
4. Loop-carried data dependence from S3 to S2 on a[].
5. Loop-carried data dependence from S3 to S3 on a[].

To parallelize the loop we simply "break" all of the loop-carried dependences. The loop-carried output dependence can be removed by the compiler through renaming. The loop-carried data dependences can be removed only by transforming the structure of the code. The fact that they both involve the statement that is the source of the output dependence and that the data dependence is on the same data item suggests a close look at the original code. We can see that S3 does no actual useful work. All values computed by S3 are overwritten by S1. Even if the result of S3 is renamed to eliminate the output dependence, the results of S3 are still not intended to be saved. Thus, statement S3 can be removed from the original code.

```
for (i = 1; i < 100; i=i+1) {
   a[i] = b[i] + c[i];   /* S1 */
   b[i] = a[i] + d[i];   /* S2 */
}
```

This transformed loop is functionally equivalent to the original and eliminates all the loop-carried dependences. This is a parallel loop.

4.7 a. The given code has a loop-carried dependence from iteration i to $i + 1$ and high latency dependences within and between loop bodies. Simply unrolling twice yields

```
foo:   L.D      F0,0(R1)
       L.D      F4,0(R2)
       L.D      F6,#-8(R1)
       MUL.D    F0,F0,F4      ;1 from L.D F4,0(R2)
       L.D      F8,#-8(R2)
       DADDUI   R1,R1,#-16
       MUL.D    F6,F6,F8      ;1 from L.D F8,-8(R2)
       ADD.D    F2,F0,F2      ;3 from MUL.D F0,F0,F4
       DADDUI   R2,R2,#-16
       stall
       BNEZ     R1,foo
       ADD.D    F2,F6,F2      ;in slot, 3 from ADD.D F2,F0,F2
```

The dependence chain from one ADD.D to the next ADD.D forces the stall. Unrolling further will only aggravate the problem. We must break the dependence chain to schedule without stalls.

A solution takes advantage of the commutativity and associativity of dot product to compute two running sums in the loop, one for even elements and one for odd elements, and combines the two partial sums outside the loop body to get the final result. The code for this solution is

```
foo:  L.D     F0,0(R1)
      L.D     F6,-8(R1)
      L.D     F4,0(R2)
      L.D     F8,-8(R2)
      MUL.D   F0,F0,F4      ;1 from L.D F4,0(R2)
      MUL.D   F6,F6,F8      ;1 from L.D F8,-8(R2)
      DADDIU  R1,R1,-16
      DADDIU  R2,R2,-16
      ADD.D   F2,F0,F2      ;3 from MUL.D F0,F0,F4
      BNEZ    R1,foo
      ADD.D   F10,F6,F10    ;3 from MUL.D F6,F6,F8
                            ;and fill branch delay slot
bar:  ADD.D   F2,F2,F10     ;combine even and odd
                            ;elements
```

The code shown assumes the loop executes a nonzero, even number of times. The loop itself is stall free, but there are three stalls when the loop exits. The loop body takes 11 clock cycles.

b. Figure S.18 shows the schedule of execution for the transferred code.

	Integer instruction		FP instruction	Clock cycle
foo:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F4,0(R2)		3
	L.D	F8,-8(R2)		4
	DADDUI	R1,R1,#-16	MUL.D F0,F0,F4	5
	DADDUI	R2,R1,#-16	MUL.D F6,F6,F8	6
	stall			7
	stall			8
	BNEZ	R1,foo	ADD.D F2,F0,F2	9
			ADD.D F10,F6,F10	10
				...
bar:			ADD.D F2,F2,F10	14

Figure S.18 The unrolled and scheduled code as it would look on a superscalar MIPS.

The loop body now takes 10 cycles instead of 11, a very limited return from the superscalar hardware.

4.8 a. We assume that the loop is executed an even, non-zero number of times to simplify the unrolling transformation by not having to include a second, remainder loop to handle the iterations not done by the unrolled loop body.

When unrolling a loop with no loop-carried dependences there a three steps to take. First, copy all the statements in the original loop as many times as desired, and place these iteration bodies after the original loop statements. Second, rename all the registers in the copies instructions so that they are different from the original statements (this can b e done by adding a fixed value to each register number assuming there are enough registers available). Third, interleave all of the statements by putting the *i*th in each copied iteration body after the *i*th instruction in the original loop sequence. These steps yield an instruction schedule that does not violate data dependences.

Then remove loop overhead instructions and schedule other instructions as necessary to cover pipeline latencies.Instructions that use or update index calculations will have to be updated based on reordering of instructions or elimination of intermediate index updates.

The given loop requires unrolling twice (two original iterations per unrolled loop body) to schedule without delays. In the code below, the comments indicate the amount of latency and the instruction from which the latency is measured (e.g., ">1 from L.D F0,0(R1)" means that the instruction must follow the specified load by more than one clock cycle).

```
loop:
        L.D     F0,0(R1)
        L.D     F8,–8(R1)
        MUL.D   F0,F0,F2        ;>1 from L.D F0,0(R1)
        MUL.D   F8,F8,F2        ;>1 from L.D F8,–8(R1)
        L.D     F4,0(R2)
        L.D     F12,–8(R2)
        ADD.D   F0,F0,F4        ;>3 from MUL.D F0,F0,F2
        ADD.D   F8,F8,F12       ;>3 from MUL.D F8,F8,F2
        DSUBUI  R1,R1,#16
        DSUBUI  R2,R2,#16
        S.D     +16(R2),F0      ;>2 from ADD.D F0,F0,F4
        BNEZ    R1,loop
        S.D     +8(R2),F8       ;>2 from ADD.D F8,F8,F1
                                ; and fills the branch delay slot
```

Figure S.19 Code unrolled with two original iterations in the body and scheduled.

This code has 13 instructions and takes 13 clock cycles to produce two results for a time of 6.5 clocks per element.

b. The loop must be unrolled four times. Figure S.20 shows the schedule.

	Integer instruction		FP instruction		Clock cycle
loop:	L.D	F0.0(R1)			1
	L.D	F8,–8(R1)			2
	L.D	F16,–16(R1)	MUL.D	F0,F0,F2	3
	L.D	F24,–24(R1)	MUL.D	F8,F8,F2	4
	L.D	F4,0(R2)	MUL.D	F16,F16,F2	5
	L.D	F12,–8(R2)	MUL.D	F24,F24,F2	6
	L.D	F20,–16(R2)	ADD.D	F0,F0,F4	7
	L.D	F28,–24(R2)	ADD.D	F8,F8,F12	8
	DSUBUI	R1,R1,#32	ADD.D	F16,F16,F20	9
	S.D	0(R2),F0	ADD.D	F24,F24,F28	10
	S.D	–8(R2),F8			11
	S.D	–16(R2),F16			12
	DSUBUI	R2,R2,#32			13
	BNEZ	R1,loop			14
	S.D	+8(R2),F24			15

Figure S.20 The unrolled and scheduled code for the dual issue processor of Figure 4.2.

The code takes 15 cycles and 30 issue slots for the dual-issue processor, with 23 slots used and 7 slots unused. In 15 clock cycles the code produces four results for a time of 3.75 clocks per result. The dual-issue processor is 73% faster than the single issue processor with code unrolled twice.

4.9 a. Figure S.21 compares the unscheduled and scheduled code for the single-issue pipeline.

Unscheduled code			Scheduled code			Clock cycle
bar:	L.D	F2,0(R1)	bar:	L.D	F2,0(R1)	1
	stall			L.D	F6,0(R2)	2
	MUL.D	F4,F2,F0		MUL.D	F4,F2,F0	3
	L.D	F6,0(R2)		DADDUI	R1,R1,#8	4
	stall			DADDUI	R2,R2,#8	5
	stall			DSGTUI	R3,R1,#800	6
	ADD.D	F6,F4,F6		ADD.D	F6,F4,F6	7
	stall			*stall*		8
	stall			BEQZ	R3,bar	9
	S.D	0(R2),F6		S.D	–8(R2)	10
	DADDUI	R1,R1,#8				11
	DADDUI	R2,R2,#8				12
	DSGTUI	R3,R1,#800				13
	stall					14
	BEQZ	R3,bar				15
	stall					16

Figure S.21 Unscheduled and scheduled code for the single issue pipeline.

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

Note that several other code schedules with a total time of 10 clocks are possible.

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

Scheduled Code			Clock cycle
bar:	L.D	F2,0(R1)	1
	L.D	F8,8(R1)	2
	L.D	F6,0(R2)	3
	L.D	F12,8(R2)	4
	MUL.D	F4,F2,F0	5
	MUL.D	F10,F8,F0	6
	DADDUI	R1,R1,#16	7
	DSGTUI	R3,R1,#800	8
	ADD.D	F6,F4,F6	9
	ADD.D	F12,F10,F12	10
	DADDUI	R2,R2,#16	11
	S.D	0(R2),F6	12
	BEQZ	R3,bar	13
	S.D	8(R2),F12	14

Figure S.22 Code unrolled twice and scheduled without stalls.

This code produces two results per 14 clock cycles, for 7 clock per element. The major reduction in time per element is because unrolling twice eliminates four instructions compared to not unrolling (the loop overhead instructions DADDUI, DADDUI, DSGTUI, and BNEZ of one of the iterations). This immediately saves four clock cycles compared to the original loop. Further, with unrolling the loop body is more suited to scheduling and allows the stall cycle present in the scheduled original loop to be eliminated, also. Without the dynamic instruction count reduction and the better schedule, this unrolled version of the loop would have the same performance as the scheduled by not unrolled loop.

c. Figures S.23 and S.24 show the VLIW code schedules for four and ten times unrolling. To unroll ten times, register usage must be more efficient than in the given code, each iteration uses only two registers instead of four, so that the 32 FP registers are sufficient.

The four-times unrolled loop takes 11 cycles to compute 4 elements, or 2.75 clocks per element. The schedule executes 24 instructions in 55 slots, filling about 44%. The total code size is 11 VLIW instructions, or the equivalent space of 55 MIPS instructions, a substantial increase from the original loop. This code uses 16 registers.

The 10-times unrolled code takes 15 cycles to compute 10 elements, or 1.5 clocks per element. The schedule executes 54 instructions in 75 slots, filling about 72%. The total code size is 15 VLIW instructions, or the equivalent space of 75 MIPS instructions, again a substantial increase from the original loop. This code uses 20 registers.

Memory reference 1	Memory reference 2	FP operation 1		FP operation 2		Integer operation/branch	
L.D F2,0(R1)	L.D F8,8(R1)						
L.D F14,16(R1)	L.D F20,24(R1)						
L.D F6,0(R2)	L.D F12,8(R2)	MUL.D	F4,F2,F0	MUL.D	F10,F8,F0		
L.D F18,16(R2)	L.D F24,24(R2)	MUL.D	F16,F14,F0	MUL.D	F22,F20,F0		
						DADDUI	R1,R1,#32
						DADDUI	R2,R2,#32
		ADD.D	F6,F4,F6	ADD.D	F12,F10,F12		
		ADD.D	F18,F16,F18	ADD.D	F24,F22,F24		
						DSGTUI	R3,R1,#800
S.D –32(R2),F6	S.D –24(R2),F12						
S.D –16(R2),F18	S.D –8(R2),F24					BEQZ	R3,bar

Figure S.23 VLIW instructions that form the loop body when unrolling the original loop four times. This code occupies 11 VLIW instructions and takes 11 cycles assuming no branch delay; normally a branch delay would also need to be scheduled. The issue rate is 24 operations in 11 clock cycles, or 2.18 operations per cycle. The efficiency, the percentage of available instruction slots that an operation, is 24/55 or about 44%. This VLIW code sequence uses 16 FP registers, although fewer could be used to allow more unrolling.

Memory reference 1	Memory reference 2	FP operation 1		FP operation 2		Integer operation/branch	
L.D F1,0(R1)	L.D F3,8(R1)						
L.D F5,16(R1)	L.D F7,24(R1)						
L.D F9,32(R2)	L.D F11,40(R2)	MUL.D	F1,F1,F0	MUL.D	F3,F3,F0		
L.D F13,48(R2)	L.D F15,56(R2)	MUL.D	F5,F5,F0	MUL.D	F7,F7,F0		
L.D F17,64(R1)	L.D F19,72(R1)	MUL.D	F9,F9,F0	MUL.D	F11,F11,F0		
L.D F2,40(R1)	L.D F4,24(R1)	MUL.D	F13,F13,F0	MUL.D	F15,F15,F0		
L.D F6,48(R2)	L.D F8,8(R2)	MUL.D	F17,F17,F0	MUL.D	F19,F19,F0		
L.D F10,156(R2)	L.D F12,24(R2)	ADD.D	F2,F1,F2	ADD.D	F4,F3,F4		
L.D F14,64(R2)	L.D F16,8(R2)	ADD.D	F6,F5,F6	ADD.D	F8,F7,F8		
L.D F18,72(R2)	L.D F20,24(R2)	ADD.D	F10,F9,F10	ADD.D	F12,F11,F12		
S.D 0(R2),F2	S.D +8(R2),F4	ADD.D	F14,F13,F14	ADD.D	F16,F15,F16	DADDUI	R1,R1,#80
S.D +16(R2),F6	S.D +24(R2),F8	ADD.D	F18,F17,F18	ADD.D	F20,F19,F20	DADDUI	R2,R2,#80
S.D –48(R2),F10	S.D –40(R2),F12					DSGTUI	R3,R1,#800
S.D –32(R2),F14	S.D –24(R2),F16						
S.D –16(R2),F18	S.D –8(R2),F20					BEQZ	R3,bar

Figure S.24 VLIW instructions that form the loop body when unrolling the original loop ten times. This code occupies 15 VLIW instructions and takes 15 cycles assuming no branch delay; normally a branch delay would also need to be scheduled. The issue rate is 54 operations in 15 clock cycles, or 3.6 operations per cycle. The efficiency, the percentage of available instruction slots that an operation, is 54/75 or 72%. This VLIW code sequence uses only 20 FP registers because the code has been re-written.

d. VLIW compilers develop an instruction schedule that takes into account the hardware latencies of the intended processor. For a different VLIW processor with different latencies it is likely that a given VLIW instruction sequence will fail to execute correctly. This is because some value will not be ready at the clock cycle demanded by the original executable, and so an old register value will be used instead, yielding an incorrect result.

e. The software pipelined code is shown in Figure S.25. The time per element is 9 clock cycles.

Software pipelined code			Clock cycle
bar:	S.D	F0(R2),F6	1
	ADD.D	F6,F4,F6	2
	L.D	F6,0(R2)	3
	MUL.D	F4,F2,F0	4
	L.D	F2,0(R1)	5
	DADDUI	R1,R1,#8	6
	DSGTUI	R3,R1,#800	7
	BEQZ	R3,bar	8
	DADDUI	R2,R2,#8	9

Figure S.25 Software pipelined version of the code. This code takes 9 cycles including the branch delay slot. This is only marginally better than just scheduling the original loop, showing that there was enough ILP in the original loop to hide most of the dependent instruction latency.

4.10 a. The complete software pipelined loop is as follows. Stall cycles are indicated.

```
   L.D F0,0(R1)
   stall
   ADD.D F4,F0,F2
   L.D F0,-8(R1)
   DSUBUI R1,R1,#16
loop:
   S.D 16(R1),F4
   ADD.D F4,F0,F2
   L.D F0,0(R1)
   DSUBUI R1,R1,#8
   stall
   BNEZ R1,loop
   stall
out_of_the_loop:
   S.D 16(R1),F4
   ADD.D F4,F0,F2
   S.D 8(R1),F4
   stall
   stall
   S.D 8(R1),F4
```

An expression for the total time for the code includes a count of the stall cycles and instructions within and without the loop.

$$\begin{aligned}\text{Total cycles} &= 3 + 2(n-2) + 5(n-2) + 7\\ &= 7n - 4\end{aligned}$$

b. A solution is

```
   load to initialize R1 and F2
   SUBUI R1,R1,#8
   L.D F0,8(R1)     ; Load M(n)
   BEZ R1,one_iteration_only
   ADD.D F4,F0,F2   ; Add to M(n)
   SUBUI R1,R1,#8
   L.D F0,8(R1)     ; Load M(n-1)
   BEZ R1, two-iterations_only
loop:
   S.D 16(R1),F4    ; Store M(i)
   DSUBUI R1,R1,#8
   ADD.D F4,F0,F2   ; add to M(i-1)
   BNEZ R1,loop
   L.D F0,8(R1)     ; load M(i-2)
out_of_the_loop:
   S.D 16(R1),F4    ; store M(2)
two_iterations_only:
   ADD.D F4,F0,F2   ; add to M(1)
one_iteration_only:
   stall
   stall
   S.D 8(R1),F4     ; store M(1)
```

4.11 The startup, loop body, and cleanup code for the original software pipelined code on page 330.

```
      L.D     F0, 0(R1)
      ADD.D   F4, F0, F2
      DADDI   R1, R1, #-8
      L.D     F0, 0(R1)
      DADDI   R1, R1, #-8
LOOP:S.D      F4, 16(R1)
      ADD.D   F4, F0, F2
      L.D     F0, 0(R1)
      DADDI   R1, R1, #-8
      BNE     R1, R2, LOOP
      S.D     F4, 8(R1)
      ADD.D   F4, F0, F2
      S.D     F0, 0(R1)
```

The new startup, loop body, and cleanup code for the software pipelined code unrolled twice.

```
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        L.D     F1, -8(R1)
        ADD.D   F5, F1, F2
        DADDI   R1, R1, #-16
        L.D     F0, 0(R1)
        L.D     F1, -8(R1)
        DADDI   R1, R1, #-16
LOOP:   S.D     F4, 32(R1)
        S.D     F5, 24(R1)
        ADD.D   F4, F0, F2
        L.D     F0, 0(R1)
        ADD.D   F5, F1, F2
        L.D     F1, -8(R1)
        DADDI   R1, R1, #-16
        BNE     R1, R2, LOOP
        S.D     F4, 8(R1)
        S.D     F5, 0(R1)
        ADD.D   F4, F0, F2
        ADD.D   F5, F1, F2
        S.D     F0, 0(R1)
        S.D     F1, -8(R1)
```

4.12 Normalizing the loop leads to the modified C code, shown in Figure S.26. The greatest common divisor (GCD) test indicates the potential for dependence within an array indexed by the functions $ai + b$ and $cj + d$ if the following condition holds:

$$(d - b) \bmod \gcd(c, a) = 0$$

Applying the GCD test with, in this case, $a = 2$, $b = 0$, $c = 100$, and $d = 1$ allows us to determine if there is a dependence in the loop. Thus, gcd (2, 100) = 2 and $d - b = 1$. Because one is a factor of two, the GCD test indicates that there is a dependence in the code. In reality, there is not a dependence in the code since the loop loads its values from `a[101]`, `a[201]`, . . . , `a[5001]` and assigns these values to `a[2]`, `a[4]`, . . . , `a[100]`.

```
for(i = 1; i <= 50; i++) {
    a[2*i] = a[(100*i)+1];
}
```

Figure S.26 Normalized loop.

4.13 When the loop is normalized so the index starts at 1 and increments by 1 on every iteration, the code becomes

```
for (i=1;i<=n/4;i+=1)
   a[2i] = a[2i] + a[2i + n/2];
```

This code performs the same number of iterations with the same index values for the a[] array as the original code. Note that n/2 is an integer because the original code uses i + n/2 as an array index, which must be integer. The value n/4 may not be integer, but the loop iteration count will be correct whether n/4 happens to be integer or not.

A loop-carried true dependence exists in the loop if the result in array element a[2i] is ever read to be an operand as array element a[2i] or a[2i + n/2] in any subsequent iteration, i.e., for some subsequent value of i. Clearly, there is no need to perform the GCD test between a[2i] the result and a[2i] the operand because they are used only in the same iteration so no dependence exists. However, if the GCD test is performed thee values are a = 2, b = 0, c = 2, and d = 0. The GCD(2,2) = 2, which does divide 0 – 0 = 0, so in this case the test reports a possible dependence when one does not exist.

Between a[2i] and a[2i + n/2] the possibility of index value overlap is less clear and the GCD test may be more helpful. The values are a = 2, b = 0, c = 2, and d = n/2. Again, the GCD(2,2) = 2 and this may divide d – b = n/2 – 0 = n/2, so the test reports a possible dependence if n/2 happens to be even.

Looking closer at the range of possible array index values shows that for i=1 to i<=n/4 the range of the expression a[2i] is from a[2] to a[n/2] and the range of the expression a[2i + n/2] is from a[2 + n/2] to a[n]. These two ranges do not overlap, so a loop carried true dependence is actually impossible for these array references, and for the loop as a whole.

If the loop index i is allowed to be any integer, then an index of the form $a \times i + b$ defines an infinite set of uniformly-spaced integer values, as does the index $c \times i + d$. The mathematical workings of the GCD test examine whether there is an intersection between these two sets *anywhere*. This is the source of the conservatism of the GCD test: it reports dependence if there is an intersection without regard to whether that intersection is within the range of loop index values.

4.14 For a data dependence to exist the indices used to access the array must be the same. We can write this as

$$ai + b = cj + d$$

for some values of i and j. Manipulating the above equation to match the form of the GCD test yields

$$ai - cj = d - b$$

The values of a and c can be expressed as

$$a = g\prod_x a_x$$

$$c = g\prod_y c_y$$

where g is the greatest common divisor of a and c and a_x and c_y represent the remaining, integer, prime factors of a and c. Substituting this formulation of a and c gives

$$g\left(\prod_x a_x\right)i - g\left(\prod_y c_y\right)j = d - b$$

After manipulating this equation to depict the actual GCD divisibility test we have

$$\left(\prod_x a_x\right)i - \left(\prod_y c_y\right)j = \frac{d-b}{g}$$

Analyzing the lefthand side of the equation, each factor is integer, the product of two integers is integer and the difference of two integers is integer, so the left-hand side must be integer. Thus the righthand side must also be integer, meaning that g divides $d - b$. We set the initial conditions to force a dependence, so this then confirms the GCD test rule that g divides $d - b$ when there is a possible dependence.

Keep in mind that a positive outcome for the GCD test is a necessary but not sufficient condition for a true dependence. Although the test may indicate a dependence between $ai + b$ and $cj + d$, the range over which i and j iterate also comes into play. For example, the GCD test may indicate dependence even though $ai + b$ is always between 0 and 100 and $cj + d$ is always between 1000 and 1100.

4.15 As the loop executes from i=1 to i<=n the index array x[i] provides a sequence of of values used to name elements from arrays a[] and b[]. There can be no loop-carried dependence if the elements of a[] are named only once. This condition is true when there are no duplicate values in the elements of x[].

4.16 a. Recompile the code given in the problem using conditional load instructions.

```
LD      R1, 0(R3)
DADDI   R4, R1, #4
LWZ     R4, 0(R2), R1
SD      0(R3), R4
```

The original code requires 4 instructions on the not taken path and 5 on the taken path. This new code requires only 4 instructions and uses one more register. The two stalls caused by the branch are removed by removing the branch. They are replaced by a single stall in the DADDI instruction.

b. Recompile the code given in the problem using boosting.

```
      LD       R1, 0(R3)
      BNEZ     R1, L1
      LD+      R1, 0(R2)
      J        L2
L1:   DADDI    R1, R1, #4
L2:   SD       0(R3), R1
```

The boosted load would be executed between the first load and the BNEZ, then according to the proper branch outcome, the new value of R1 (from the load) would be committed to R1. This would shorten the not taken path by effectively removing one stall from the BNEZ instruction; however the taken path remains unchanged. No additional registers are consumed in this code.

c. Recompile the code given in the problem using conditional move instructions.

```
      LD       R1, 0(R3)
      LD       R5, 0(R2)
      DADDI    R4, R1, #4
      CMVZ     R4, R5, R1
      SD       0(R3), R4
```

The conditional move allows the load of B to be moved between the first load and the DADDI. This effectively removes the memory stall of DADDI. This compilation requires two more registers over the original code. However, this produces one additional instruction in the original code when the branch is not taken.

4.17 The branch is written to skip the LW instructions if R10 = 0, so let's assume the branch guards against a memory access violation. A violation would terminate the program, so if LW R8,0(R10) is moved before the branch, the effective address must not be zero. The load can be guarded by conditional move instructions if there are two unused registers available. One of the registers must contain a safe address for the load. The code is

```
DADDI    R29,R0,#1000     ;initialize R29 to a safe address
LW       R1,40(R2)
MOV      R30,R8           ;save R8 in unused R30
CMOVNZ   R29,R10,R10      ;R29 is unused and contains a safe
                          ;address. R29 ← R10 if R10
                          ;contains a safe address ≠ 0
LW       R8,0(R29)        ;speculative load
CMOVZ    R8,R30,R10       ;if R10 = 0 load is incorrectly
                          ;speculated so restore R8
BEQZ     R10,L
LW       R9,0(R8)
```

Both loads after the branch can be speculated using only conditional moves. One more unused register is needed. The code is

```
ADDI    R29,R0,#1000
LW      R1,40(R2)
MOV     R30,R8
MOV     R31,R9          ;save R9 in unused R31
CMOVNZ  R29,R10,R10
LW      R8,0(R29)
LW      R9,0(R8)        ;now this load is speculated
CMOVZ   R8,R30,R10
CMOVZ   R9,R31,R10      ;restore R9 if needed
```

There is no branch instruction at all, but there is a significant conditional instruction overhead.

4.18 a. The code becomes

```
        cmp.eq pT,pF = R13,R14
(pT)    DADDI R2,R2,#1
(pT)    SD 0(R7),R2
(pF)    MUL.D F0,F0,F2
(pF)    ADD.D F0,F0,F4
(pF)    S.D 0(R8),F0
```

b. In the given code of part (a) the data dependences are DSUB --> BNEZ on R1, DADDI --> SD on R2, MUL.D --> ADD.D on F0 (the instruction should be ADD.D F0,F0,F4), and ADD.D --> S.D on F0. There is one name dependence, an output dependence, MUL.D --> ADD.D on F0. Finally, there are control dependences from BNEZ to all subsequent instructions.

The data dependences in the predicated code of part (a) are cmp.eq to all subsequent instructions, DADDI --> SD on R2, MUL.D --> ADD.D on F0, and ADD.D --> S.D on F0. There is again one output dependence, MUL.D --> ADD.D on F0.

Because the predicated code has no control dependences, the potential for exploiting ILP is much greater. Also, all the instructions can be fetched from sequential memory locations. This tends to improve cache performance and thus effective memory access time.

4.19 a. The predicated code is

```
       CMP.GT pT,pF = a,b
(pT)   ADDI X,R0,#1
(pF)   CMP.LT pJ,pK = c,d
(pJ)   ADDI X,R0,#2
(pK)   ADDI X,R0,#3
```

b. Stops are shown as ;; symbols after an instruction that is followed by a stop.

```
CMP.GT pT,pF = a,b              ;;
(pT)       ADDI X,R0,#1
(pF)       CMP.LT pJ,pK = c,d   ;;
(pJ)       ADDI X,R0,#2
(pK)       ADDI X,R0,#3
```

The first stop enforces completion of the comparison setting the pT and pF predicate bits before those bits are used. The second stop enforces completion of the second compare before predicates pJ and pK are used.

c. The possible template sequences from Figure 4.12 all begin with template 11 which supports a bundle containing the first compare, first ADDI, and the second compare. The second bundle may be any of templates 8, 9, 14, 15, 24, and 25 because each of these supports the remaining two ADDI instructions plus one more instruction of some kind. With no subsequent instruction specified because the exercise deals just with this code fragment, any of the listed templates is a possibility.

4.20 Predicated instructions maintain a straight-line program execution path. They are useful for collapsing relatively balanced, parallel, rejoining execution paths by removing the conditional branch and transforming its function into new data dependences.

In general a fork with non-rejoining execution paths, established by a branch, will be too unbalanced for predicated instructions to offer any performance improvement. There is no point in using predication if performance will be worse than retaining the branch.

4.21 There may be a true dependence on R1 from the ADD to the SUB, depending on the truth values of predicate registers p1 and p2. The cases are

Compiler determines that	Dependence?	Template to use
p1 = p2	yes	A stop is needed to preserve the dependence, so use template 2 or, if a stop is also needed between the SUB and the following instruction, then use template 3
p1 ≠ p2	no	Use template 0, or if a stop is needed between the SUB and the following instruction, then use template 1
p1 ? p2	maybe	Use template 2 or 3; the compiler must produce code that will be correct in case there is a dependence

4.22 This exercise has a flaw in its formulation. Because the if statement has both then and else parts, a different form of parallel compare than described in the exercise statement is needed for a good solution. So, the literal solution is rather ugly. However, because the then and else blocks both update the same location there is an unintended way to solve the exercise using a single predicate register.

The exercise can be corrected by deleting the else block from the given code; delete "else { A=A+2}." With the else block deleted, the first three bundles of the literal solution form a fine answer. The exercise can also be corrected by describing a different type of parallel compare instruction that has the ability to directly form predicates for arbitrary then and else blocks.

The following presents the literal (straightforward) solution, the unintended solution that depends on the special then/else relationship, a corrected exercise statement presenting the different type of parallel compare, and a solution to the corrected exercise statement.

Straightforward (brute force) solution (Figure S.27):

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/ cycle)
1: M M I	`CMP.EQ p1,p2=R0,R0`			1
1: M M I	`CMP.NE.AND p1=RX,#1`	`CMP.NE.AND p1=RY,#1`	`CMP.NE.AND p1=RZ,#1`	2
1: M M I	`(p1) ADDI RA,RA,#1`			3
1: M M I	`MOVE R1,p1`			4
1: M M I	`CMP.EQ p2,p3=R1,R0`			5
1: M M I	`(p1) ADDI RA,RA,#2`			6

Figure S.27 The literal solution for the exercise. This solution begins by initializing the p1 and p2 predicate registers to true and false, respectively, in the first bundle. Blank entries in this and other bundles are encoded as no-ops. This bundle must complete before the second bundle is executed, thus a stop is necessary and template 1 rather than 0 is chosen. The second bundle performs a parallel compare on the if condition boolean values, X, Y, and Z. For a true boolean value among X, Y, and Z, the CMP.EQ.AND instruction will find the register equal to 1 and thus make no change to predicate p1. For any false boolean value among X, Y, and Z the compare will set p1 to false. If more than one of the three compares in the second bundle finds a false boolean value, the update to p1 will be identical in all cases and so can be performed correctly in parallel This bundle must also complete before the next bundle executes, so a stop is chosen after slot 3. The third bundle holds the then block instruction and executes it if the value of p1 has remained true after the parallel compare. If p1 is false the else block should be performed instead, and this instruction will execute as a no-op. This solution runs into difficulty now because a predicate register value to control execution of the else statement A=A+2 does not yet exist. The complement of the value of p1 is needed, and the fourth bundle begins a brute force sequence to compute it. First p1 is copied into a register that can be an operand of a compare instruction. Then this value is compared in the fifth bundle to R0, which always contains false, in a way that will make p2 the complement of p1. The final bundle, which must come after the second compare, conditionally executes the elseblock instruction. This solution can handle arbitrary then and else blocks, but does not use the appropriate form of parallel compare instruction. If there were no else block in the program code, then the form of parallel compare described would be appropriate, and the first three bundles would form a fine solution.

Unintended solution (Figure S.28):

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/ cycle)
1: M M I	`CMP.EQ p1,p2=R0,R0`			1
1: M M I	`CMP.NE.AND p1=RX,#1`	`CMP.NE.AND p1=RY,#1`	`CMP.NE.AND p1=RZ,#1`	2
1: M M I	`ADDI RA,RA,#2`			3
1: M M I	`(p1) ADDI RA,RA,#1`			4

Figure S.28 The unintended solution. The problem with this solution is that it takes advantage of the fact that the then and else statements update the same location and does not generalize to the case where the then statement(s) and the else statement(s) do not update the same location(s). What is needed is a parallel compare form that sets two predicate registers, which have been initialized to complementary values, to their opposite values in the event that one or more of the ANDed boolean values, X, Y, or Z, is false. Such a parallel compare does exist in the Itanium instruction set. Examining this solution in detail, he first bundle initializes the p1 and p2 predicate registers to true and false, respectively. Blank entries in this and other bundles are encoded as no-ops. This bundle must complete before the second bundle is executed, thus a stop is necessary and template 1 rather than 0 is chosen. The second bundle performs a parallel compare on the if condition boolean values, X, Y, and Z. For a true boolean value among X, Y, and Z, the CMP.EQ.AND instruction will find the register equal to 1 and thus make no change to predicate p1. For any false boolean value among X, Y, and Z the compare will set p1 to false. If more than one of the three compares in the second bundle finds a false boolean value, the update to p1 will be identical in all cases and so can be performed correctly in parallel This bundle must also complete before the next bundle executes, so a stop is chosen after slot 3. The third bundle holds the else block instruction and always performs it. The final bundle, which must come after the third bundle, conditionally executes the then block instruction, overwriting the work of the else block instruction when appropriate.

Corrected exercise statement:

A compound conditional joins more than two values by Boolean operators, for example, X&&Y&&Z. If predicate-generating compare instructions can be ganged together to simultaneously update a pair of predicate registers, then compound conditionals can be computed more rapidly. Consider a parallel computation of an && -only compound conditional (a conjunction term). If the pair of predicate registers were initialized to true and false, then simultaneous writes by only those compare instructions determining that their comparison should complement both predicate registers is readily supportable in hardware. There will be no contention from trying to set the predicates simultaneously to conflicting value pairs. A parallel not-equal compare to update a complementary predicate register pair might be written

```
(qp) CMP.NE.AND.ORCM pT,pF=Rx,#0
```

where CMP means compare; .NE denotes the relationship "not equal" which is one of many possible relationships including equal, greater than, less than, and so forth; .AND.ORCM indicates the type of parallel comparison; pT and pF are the complementary predicate registers; and Rx and #0 are the values to be compared. The immediate value #0 denotes false. The meaning of .AND.ORCM is that the writes to complement both predicate registers pT and pF will occur only if the

comparison of Rx and #0 is false, that is Rx is *not* equal to #0. If the comparison is true, i.e., Rx is not equal to #0, then the predicate registers pT and pF are untouched.

Initialize a pair of predicate registers to true and false. The use the .AND.ORCM parallel form of comparison to transform the following code into a single block of predicated instructions. Form the code into as few bundles as possible, as in Figure 4.13 on page 355.

if (X && Y && Z) then { A=A+1; }

else { B=B+2; }

Solution to corrected version of exercise (see Figure S.29):

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
1: M M I	`CMP.EQ p1,p2=R0,R0`			1
1: M M I	`CMP.EQ.AND.ORCM p1,p2=RX,#1`	`CMP.EQ.AND.ORCM p1,p2=RY,#1`	`CMP.EQ.AND.ORCM p1,p2=RZ,#1`	2
1: M M I	`(p1) ADDI RA,RA,#1`	`(p2) ADDI RB,RB,#2`		3

Figure S.29 Solution to the corrected exercise statement. The first bundle initializes the p1 and p2 predicate registers to true and false, respectively. Blank entries in this and other bundles are encoded as no-ops. This bundle must must complete before the second bundle is executed, thus a stop is necessary and template 1 rather than 0 is chosen. The second bundle performs a parallel compare on the if condition boolean values, X, Y, and Z. For a true boolean value among X, Y and Z, the CMP.EQ.AND.ORCM instruction will find the register equal to 1 and thus make no change to the predicates p1 and p2. For any false boolean value among X, Y, and Z the compare will complement both p1 and p2. If more than one of the three compares in the second bundle finds a false boolean value, the update to p1 and p2 will be identical in all cases and so can be performed correctly in parallel This bundle must also complete before the next bundle executes, so a stop is chosen after slot 3. The third bundle holds the then block and else block instructions for this case where then and else blocks are small. Because p1 and p2 remain complementary after the parallel .AND.ORCM type compares, the ADDI instructions in the third bundle are mutually exclusive and so can be done simultaneously. For larger then and else blocks more bundles of instructions may be needed, each instruction will be predicated by p1 or p2 depending on whether it is from the then or else block, respectively. This solution performs the computation in only three cycles.

4.23 The branch in the given code checks if R1 contains a null pointer. The code then loads F2 by dereferencing the pointer and then used the loaded value. Normally, the compiler cannot schedule the load before the branch because of the potential for a memory access fault.

a. The speculative load instruction defers the hardware response to a memory access fault should one occur. In combination with the speculation check instruction this allows the load to be moved above the branch. Because the load may have long latency it should be moved as early in the program as possible, in this case to the position of first instruction in the basic block. Use of the load has not been speculated, so if the speculation check finds no deferred exceptions, computation can proceed.

```
sL.D     F2,0(R1)
instr. 1
instr. 2
. . .
BEQZ     R1,null
SPECCK   F2      ;check for exception deferred by sL.D on F2
ADD.D    F4,F0,F2
. . .
null:    . . .
```

b. With a speculation check instruction that can branch to recover code, instructions dependent on the load can also be speculated. Now if the load fails because of an exception for high latency such as a TLB miss or page fault, rather than one that is a fatal error, such as a memory protection access violation, the speculated use instruction(s) may take as an operand an incorrect value from the register that is the target of the delayed load. The speculation check instruction can distinguish these types of exceptions, terminating the program in the event of a protection violation and branching to recovery code for the case of a page fault or TLB miss which will yield correct load behavior given sufficient time.

```
      sL.D   F2,0(R1)
      instr. 1
      instr. 2
      . . .
      ADD.D  F4,F0,F2    ;ADD.D speculated, far from load for
                          latency
      BEQZ   R1,null
      SPECCK F2,recover  ;check for exception deferred by sL.D
                          on F2
                         ;and branch to "recover" if exception
back:  . . .
       . . .             ;next instruction following back
       . . .             ;intervening instructions
recover: L.D   F2,0(R1)    ;
         ADD.D F4,F0,F2
         JUMP  back        ;return to original path
         . . .
         . . .
null:. . .
```

c. In part (a) the speculation check is ensuring that fatal exceptions, such as a memory access protection violation are only taken if the load would actually be executed. In part (b) a high-latency exception, such as a page fault, will trigger the recovery code.

4.24 For the processor of Figure 4.1 the integer load latency is 1, and the floating-point load latency is either 1 or 0 depending on the destination. The processor of Figure 4.15 has the same integer load latency, but has a much greater floating-point load latency of 9.

The value of an ALAT is to allow loads to easily be moved earlier than preceding stores. This is essential if loads are to be scheduled much earlier. However, to avoid causing data hazard stalls, a load need only be scheduled earlier than its dependent instructions by an amount equal to the load latency.

Thus, an ALAT is more beneficial to the processor of Figure 4.15 because it will be needed more often to hide the long floating-point latency of that processor.

4.25 Speculation and predication are techniques to aggressively exploit more ILP. The effect is to put more instructions through the CPU pipeline per unit time, including some instructions that will not commit or will be nullified. This additional CPU work will increase the power and energy demands for a given computation or workload. In the embedded computer marketplace, higher power and energy consumption is often unwelcome, as compared to the server or desktop arena. Further, in the embedded market processing speed may be quite adequate without aggressive techniques. Aggressive techniques will be least valued in the cost-sensitive, low-performance, and low-power portions of the computing marketplace.

4.26 The comparison is listed as follows.

Technique: Loop unrolling.
Best-suited branch characteristics: limited to loop-forming branches.
Suitable program structures: loops.
Needed hardware support: ample register file(s).
Complexity of compiler support: simple.
Effect on code size: increases roughly proportional to unrolling factor.
Effect on fetching: helps fetch efficiency by reducing the number of branches.
Other characteristic(s): helps instruction scheduling by exposing ILP.

Technique: Trace scheduling.
Best-suited branch characteristics: loop-forming and highly predictable branches.
Suitable program structures: any with a highly likely path of execution.
Needed hardware support: none, but targeted for VLIW hardware.
Complexity of compiler support: significant.
Effect on code size: increases code size because of recovery code.
Effect on fetching: helps by finding highly likely execution path and placing those instructions sequentially in memory.
Other characteristic(s): tries to expose large amounts of ILP.

Technique: Superblocks.
Best-suited branch characteristics: loop-forming.
Suitable program structures: loops.
Needed hardware support: none.

Complexity of compiler support: less than trace scheduling because there is only one entry point for a superblock, but a trace may have many entry points.
Effect on code size: increases more than trace scheduling.
Effect on fetching: helps by reducing branches.
Other characteristic(s): forms large blocks that help compiler with instruction scheduling.

Technique: Predication.
Best-suited branch characteristics: unpredictable branches, with short, relatively balanced alternate paths of execution that rejoin.
Suitable program structures: short alternative paths, case statements.
Needed hardware support: ISA modification to include predicated instructions; simultaneous or multiway comparison is useful for performing case statements.
Complexity of compiler support: ranges from simple to complex.
Effect on code size: little.
Effect on fetching: helps fetch efficiency by eliminating branch instructions.
Other characteristic(s): a powerful technique to combine with speculation.

4.27 No solution provided.

4.28 The straightforward way to handle predicated instructions is to treat them just as normal instructions with an additional operand, the predicate register. The issue step proceeds as in the basic Tomasulo design, but the reservation station now has an additional field to hold the predicate register value. In the execute phase, the reservation station again monitors the CDB for results, including the predicate value, that are needed as operands. When all operands, including the predicate value are available and the logic value of the predicate indicates that the instruction should execute, then the reservation station signals the function unit that it is ready. The function unit processes the instruction, puts the result on the CDB when able, and the write-result step completes normally.

If the predicate value indicates that the instruction should execute as a no-op, then processor state must not change and any dependent instructions must receive the existing value in the result register for the predicated instruction. Executing as a no-op can be done in several ways. The reservation station might simply purge its contents without ever signalling the function unit that it is ready, or the function unit might have a mode of operation that does not put the result on the CDB. The more difficult task is to transfer the existing value in the result register to any dependent instructions already in reservation stations that will be looking only to the CDB for their needed operand. The simplest way to forward this operand is to enhance the hardware to be able to put a register value on the CBD with appropriate tagging so that the reservation stations that need the value will recognize it as the one for which they are waiting. Ideally, as soon as the predicate register value is known to require no-op behavior, the needed register value and tagging could be ready for the CDB. Dependent instructions should get their correct operand as soon as possible.

For this straightforward method to handle predicated instructions, the performance potential is to preserve the gains of predicated instructions and combine

that with the potential of dynamic instruction scheduling. The additional hardware support needed is the new capabilities in the reservation station and to place register values on the CDB. This approach does not involve speculation. Finally, dependences on the predicate behave as data dependences.

By not requiring the predicate value to be known before a reservation station signals its function unit that execution may proceed, we can combine predicated instructions with Tomasulo's algorithm and speculation.

The Issue step proceeds in much the same way. If there is an empty reservation station and ROB slot then the instruction is issued. Again, the reservation station has fields for the operands, including the predicate register. Control entries are updated as for conventional Tomasulo. Depending on how speculation is handled at the function units, the ROB slot may need to be marked to record that this slot is for a predicated instruction.

Speculation begins in the Execute step when the reservation station signals its function unit to begin execution when all operands except the predicate are available. The function unit can handle the speculation in two ways, one mildly speculative, the other more aggressive. First, if the function unit holds its result from the CDB until the predicate value is available, then the speculation is modest and the mechanism for putting a existing register value on the CDB is all that is necessary to complete execution and write result to the ROB. A more aggressive approach is for the function unit to put the possible result on the CDB and for the Write Result step to update the ROB. In this case, the ROB entries for predicated instructions must be marked during the Issue step because such an instruction must not commit until its predicate value is known.

For the speculation where the function unit waits to receive the predicate value, the Commit step for the predicated instruction is the same as for any non-branch, non-predicated instruction. For the more aggressive speculation, if the predicated instruction should execute normally, then the Commit step is also the same as for any non-branch, non-predicated instruction. However, for a predicated instruction that should execute as a no-op, the Commit step now has an additional sequence of action to take. First, the result register for the predicated instruction should remain as it is, holding the value that the program would have produced through in-order execution up to the point of reaching the predicated instruction, rather than being updated with the discarded result of the predicated instruction. Second, the instructions following the predicated instruction executed as a no-op may be dependent on it, so when the predicate value is finally known, the correct operand value must be delivered to these dependent instructions. The straightforward way to do this is to reissue program instructions starting with the instruction following the predication. Note that this is similar to the handling of an incorrectly speculated branch instruction, in that ROB entries following the predicated instruction are treated differently. But, instead of being flushed, these instructions are re-executed.

For this speculative method to handle predicated instructions, the performance potential is to preserve the gains of predicated instructions and combine that with

the potential of dynamic instruction scheduling with speculation. The additional hardware support needed is the new capability in the reservation station and ROB. This approach does involve speculation. Finally, dependences on the predicate behave in much the same way as control dependences, triggering re-issue of dependent instructions.

4.29 No solution provided.

4.30 No solution provided.

Chapter 5 Solutions

5.1 This exercise uses differences in cache organizations to illustrate how benchmarks can present a skewed perspective of system performance. Because system performance is heavily influenced by the memory hierarchy (if you do not believe this, take a look at Figure 5.2 in the text again!), it is possible to develop code that runs poorly on a particular cache organization. This exercise should drive home not only an appreciation for the influence of cache organization on performance, but also an appreciation of how difficult it is for a single program to provide a reasonable summary of *general* system performance.

a. Consider the MIPS code blurb shown in Figure S.30. We make two assumptions in this code: First, the value of `r0` is zero; second, locations `f00` and `bar` both map onto the same set in both caches. For example, `foo` and `bar` could be `0x00000000` and `0x80000000` (these addresses are in hexadecimal), respectively, since both addresses reside in set zero of either cache.

   On Cache A, this code only causes two compulsory misses to load the two instructions into the cache. After that, all accesses generated by the code hit the cache. For Cache B, all the accesses miss the cache because a direct-mapped cache can only store one block in each set, yet the program has two active blocks that map to the same set. The cache will "thrash" because when it generates an access to `foo`, the block containing `bar` is resident in the cache, and when it generates an access to `bar`, the block containing `foo` is resident in the cache.

   This is a good example of where a victim cache [Jouppi 1990] could eliminate the performance benefit of the associative cache. Keep in mind that in this example the information that Cache B misses on is always recently resident.

```
foo:    beqz    r0,bar    ;branch iff r0 == 0
        .
        .
        .
bar:    beqz    r0,foo    ;branch iff r0 == 0
```

Figure S.30 MIPS code that performs better on Cache A.

b. Consider the MIPS code blurb shown in Figure S.31. We make two assumptions: first, locations `baz` and `qux` and the location pointed to by `0(r1)` map to different sets within the caches and are all initially resident; second, `r0` is zero (as it always is for MIPS).

   This code illustrates the main thrust of a program that makes a system with Cache B outperform a system with Cache A, that is, one that repeatedly writes to a location that is resident in both caches. Each time the `sw` executes on Cache A, the data stored are written to memory because Cache A is write through. For Cache B, the `sw` always finds the appropriate block in the cache (as we assume the data at location `0(r1)` are resident in the cache) and updates only the cache block, as the cache is write back; the block is not written to main memory until it is replaced.

```
baz:    sw      0(r1),r0    ;store r0 to memory
qux:    beqz    r0,baz      ;branch iff r0 == 0
```

Figure S.31 MIPS code that performs better on Cache B.

c. With all accesses hits, Cache A allows the processor to maintain CPI = 1. Cache B misses each access at a cost of 100 ns, or 200 clock cycles. Thus Cache B allows its processor to achieve CPI = 200. Cache A offers a speedup of 200 over Cache B.

d. In the steady state, Cache B hits on every write and, so, maintains CPI = 1. Cache A writes to memory on each store, consuming an extra 100 ns each time. Cache B allows the processor to complete one iteration in 2 clocks. With Cache A the processor needs 202 clocks per iteration. Cache B offers a speedup of 101 over Cache A.

5.2 While specific answers are not provided for this problem, the following gives you a general how-to guide to conduct and analyze the results from running the code.

a. The program varies two key parameters to explore the memory hierarchy. They are the stride of which an array is accessed and the size of the array. The program contains a triply nested loop in which it controls the portion of the array to access, the stride size, and finally the innermost loop repeats the experiment to remove any transient timing fluctuations. A set of timing values corresponding to a particular size of array will be called a *run.* The data from this program should be plotted using the access times for the y axis and the actual stride size for the x axis. Note, there is a difference between the element/array stride and the actual stride. The actual stride is the element stride multiple by the size of the element.

b. Each level of cache will produce a horizontal cluster of timing results. In Figure 5.59, we can see that there are two levels of cache. Note the lower and upper timing clusters before the abrupt timing change. If there was another

level of cache, we would see yet another cluster with an increased access time.

c. In order to measure the size of the L1 cache and its block size, we are looking for two abrupt changes in access timings across various sized runs. If the entire array fits into the L1 cache, every access hits in L1. When it does not fit in L1, the processor will have to fetch the value from L2 (if it exists). If the array is too large for L1, we will have no L1 hits since we access elements in the array cyclically. The exceptions here are if our stride is small enough that we are accessing elements within the block or if our stride is large enough that we are only accessing a very small fraction of the array. Thus, the largest size array that belongs to the lowest access time cluster, as defined in part b., yields the size of the cache. The block size of L1 is found when the stride is small enough that we are hitting multiple elements within the same block. For an example, see the three data points in the lower left hand side of Figure 5.59 before a slight increase in access times.

d. The same technique that was used in part c. can be applied again to find the size of the L2 cache. However finding the block size of the L2 cache becomes difficult. If your stride is small enough, a block from L2 will be loaded into L1 and you will have L1 hits. You want to find the stride that will hit only once in any given L2 block. Once this starts to happen, the access timing in the L2 cluster, as defined in part b., will level off. The stride at which this happens is the size of your L2 block size.

e. The maximum number of blocks that map to a given cache slot (index) is, by definition, the cache's associativity. If we consistently access more blocks that can fit into a particular slot, we will constantly generate misses into the next higher/slower level of the memory hierarchy. A straightforward technique to target a particular slot is to use a stride the size of the cache. To determine the number of entries mapping into this slot, take the size of the array used in the run and divide by the actual stride. There will be an abrupt timing change when we start to surpass the associativity of the cache. Use the stride that resulted in the smaller timing value to compute the associativity.

f. A TLB entry maps a page in virtual memory to a page in physical memory. Therefore, a TLB miss occurs when the processor is trying to access data within an unmapped page. We will look at how TLB misses yield the page size and the number of entries in the TLB in part g.

g. To find the number of TLB entries, we must find out how much memory it maps. For an example, let us assume a processor with a 512 entry fully associative TLB on a system with an 8 KB page size. This TLB will map up to 4 MBs of memory. Now let us imagine running the program in this problem on such a machine. The 4 MB and 8 MB cache size runs will give us the desired information. The 4 MB run will produce TLB hits (compulsory misses vanish after first loop iteration). However, the 8 MB run will not produce as many TLB hits (we will have capacity misses). The 8 MB run will only hit in the

TLB when we are striding across multiple elements within a page. The difference between access times between the 4 MB and 8 MB runs will reach its maximum when the stride hits the page size. Once the stride in the 8 MB run has passed the page size limit, it will experience a TLB hit on every access since we are effectively accessing 4 MBs instead of 8 MBs. This means that this access time will match the access timing of the 4 MB on a stride of the page size. To conclude, the stride that peaks the difference is the page size and lower of the two sizes (4 MBs in this case) to produce this peak is the TLB mapping capacity. The number of TLB entries is the mapping capacity divided by the page size.

h. The L1 miss penalty includes an access to the L1 cache and ultimately an access to the L2 cache. Depending on the particular definition of a miss penalty, the first term may or may not be present in the calculation. With it present, the L1 miss penalty is the time it takes to access L2. The second timing cluster, as defined in part b., is the time it takes to access L2 having accessed L1. The same methodology may be applied to figure the L2 miss penalty to the next higher level cache or main memory.

i. We must adjust the array size before we can measure the page fault penalty to disk. This will require setting CACHE_MAX to a value higher than the amount of physical memory on your system. In order to work with the rest of the program, as it stands, this new value must be a power of two. With a small stride, the fault penalty will amortize over multiple hits within a page. The access timings will slowly increase and flatten out as the code only touches one element per page. The timing value when it flattens out is the page fault penalty. This value includes a TLB miss penalty. As mentioned in the problem, the penalty should be in the milliseconds since it is accessing a disk.

j. You may have to increase CACHE_MAX in order to see TLB misses. For example, let us consider the UltraSparc III. It has a 512 entry TLB. Each entry maps an 8 KB page yielding a TLB mapping capacity of 4 MBs. When looking at the results from an 8 MB run, you will notice several phases in access timing. With a short stride, the TLB miss will amortize over several hits. As the stride increases, the miss will not be amortized as much. It will eventually flatten out. The timing value when it flattens out is the TLB miss penalty. If your L2 cache holds less than the TLB can map, a TLB miss penalty will likely include a L2 miss as well. If your L2 cache holds more than the TLB can map, then a TLB miss penalty could be less than a L2 miss penalty. This is possible with the UltraSparc III, since it can ship with an 8 MB L2 cache.

k. Various particularities will have a direct impact on answering the previous questions. On some processors, the TLB may aggregate contiguous pages into a single entry. If so, you will measure a larger page size. Further, the TLB may or may not be fully associative. For example, the data TLB on the UltraSPARC III has a 512 entry two way set associativity structure, i.e., 256 slots with two ways. Also, be aware of associativities that are not a power of two. As mentioned before, you will need to use a stride that is not a power to two to measure such associativities.

5.3 In answering this problem, we must keep in mind that several factors will contribute to noise in the data seen in Figure 5.59. These include (a) interference in a unified cache, (b) interference between the parameters that determine the results of the benchmark, (i.e., for larger strides, page faults can distort cache-block size determination, and (c) strides that produce pathological behavior. Additionally, we need to be aware of the usual concerns when running a benchmark, such as other programs running on the machine.

Note: When answering this question, keep in mind that we might expect a stride of 64KB to fit in a cache of size 64KB, but it actually results in accessing two pieces of data that span a size of 64KB plus the size of the data. In this case, the size is 64KB plus 4 bytes.

a. How big is the cache?

The L1 data cache is 64KB. The evidence for this is the jump in run times between cache sizes 64KB and 128KB. The reason that the jump becomes more pronounced at a stride of 32 bytes is the loss of the implicit prefetching that occurs when the stride is less than the cache line size (see the solution for part b).

b. What is the block size in the cache?

For simulated cache sizes that fit in the L1 data cache (64KB), the strides of 4, 8, and 16 bytes all result in the same execution time. At a stride of 32 bytes, the execution time increases and stays the same over several stride-size increases. Therefore, the block size must be 32 bytes. Also, note that there is only one jump in the execution time for these cache sizes and small strides. If there were other jumps, this might suggest that cache subblocking is taking place.

A quick check of the specification of the Sun UltraSPARC III processor—the processor currently used in the Sun Blade 1000—verifies that the L1 caches have cache lines of 32 bytes and do not support subblocking. Additionally, the external L2 cache can have different block sizes from 64 bytes to 512 bytes, but in all cases, the processor supports subblocking at a granularity of 64 bytes. The fact that we see only one jump in the execution times confirms this subblocking support. Finally, the UltraSPARC III reference also states that the L1 cache is 4-way set-associative, and that there is a 2KB, 4-way set associative write-back cache for coalescing stores to the external cache.

c. What is the miss penalty for the L1 cache? L2 cache? Additional memory hierarchy level(s)?

The simulations report the average access time, where an access is a read followed by a write to the same memory location. Writes are usually buffered, so this solution ignores write times. The average time difference between all hits and all misses in the L1 data cache (strides of 4KB and 8KB in Figure 5.59) is about 58 – 26 = 32 ns. No other information can be precisely determined using only the data in Figure 5.59.

d. What might explain the access time behavior shown for the very large strides?

The jump to higher run times between strides of 4KB and 8KB results from a jump in conflict misses in the 32KB, 4-way associative L1 data cache. This behavior is worst at a stride of 8KB, or multiples of 8KB, since all accesses land in the same cache line. Depending on the number of lines that are conflicting, we may get some hits in the cache. If there are five or more, accessed in a round-robin fashion, we will see no hits.

For each of the large cache sizes, the execution time falls off for the last three data points. To understand this data, note that the program makes the same number of total accesses, regardless of cache size. The last data point represents a stride of one-half of the simulated cache size, resulting in two active addresses (yes, only two). The second-to-the-last data point represents a stride of one-fourth of the simulated cache size, or four active addresses. These four still fit in the *real* cache, but the simulation results in twice as many compulsory misses, so the execution time is slightly longer. The third-to-the-last data point represents a stride of one-eighth of the simulated cache size, or eight addresses. This does not fit in the cache, so we see conflict misses. For the longest-running strides, all accesses are either compulsory or conflict misses. One possible explanation for the slight drop in running time for the stride of one-eighth of the cache is the servicing of some reads by the write-back cache.

5.4 A useful tool for solving this type of problem is to extract all of the available information from the problem description. It is possible that not all of the information will be necessary to solve the problem, but having it in summary form makes it easier to think about. Here is a summary:

- Processor information
  - In-order execution
  - 1.1 GHz (0.909 ns equivalent)
  - CPI of 0.7 (excludes memory accesses)
- Instruction mix
  - 75% non-memory-access instructions
  - 20% loads
  - 5% stores
- Memory system
  - Split L1 with no hit penalty, (i.e., the access time is the time it takes to execute the load/store instruction)

- 128-bit, 266 MHz bus (3.75 ns equivalent) between the L1 and L2 caches
  - L1 I-cache
    - 32 KB, direct mapped
    - 2% miss rate
    - 32-byte blocks (requires 2 bus cycles to fill)
    - Miss penalty is 17 ns + 2 cycles = 24.5 ns
  - L1 D-cache
    - 32 KB, direct mapped, write-through (no write-allocate)
    - 5% miss rate
    - 95% of all writes do not stall because of a write buffer
    - 16-byte blocks (requires 1 bus cycle to fill)
    - Miss penalty is 17 ns + 1 cycle = 20.75 ns
    - Miss penalty on write-through is 17 ns
  - L2 (unified) cache, associativity not given—assume full
    - 512 KB, write-back (write-allocate)
    - 80% hit rate
    - 50% of replaced blocks are dirty (must go to main memory)
    - access time is 15 ns (66.7 MHz equivalent)
    - 64-byte blocks (requires 2 bus cycles to fill)
    - Miss penalty is 60 ns + 7.52 ns = 67.52 ns
  - Memory
    - 128 bits (16 bytes) wide
    - first access takes 60 ns (16.7 MHz equivalent)
    - subsequent accesses take 1 cycle on 133 MHz, 128-bit bus (7.52 ns equivalent)

Note that the processor in this description is similar to the Sun UltraSPARC III described in Section 5.15 of the text. Looking at the description of that processor can be useful in understanding the issues involved in the solution of this problem.

The problem statement says the L1 cache "imposes no penalty on hits." The problem does not say anything about the execution time taken by a memory-access instruction. That is, each memory access instruction must pass through the processor pipeline, incurring some execution latency. The solution will assume this latency is zero

Since the problem provides bus information, it is possible to include the effects of bus contention in the solution. The solution provided here assumes that the bus is able to support the required traffic. The instructor may wish to grant extra credit to students who include bus contention in their calculations. Also, note that this

solution assumes that cache miss penalties are cumulative. That is, there is no overlap between the miss penalty for the L1 cache and the L2 cache when a memory access must go to memory.

Finally, we must consider how data is delivered when a request is satisfied using multiple bus transactions. It is possible to deliver the data with the critical (word or other portion) first. This solution assumes delivery in address order.

a. What is the average memory access time for instruction accesses?

This is the average time to fetch an instruction. Assuming no bus contention, the average memory access time for instruction accesses depends on the hit/miss rates of the two caches. Keep in mind that the instruction miss rate normally never falls far below 1 / instructions per cache line, since only the first instruction of each line or branch targets falling into another line cause a miss. For this reason, architects consider L1 instruction cache miss rates of 10% to be *very* high, although data miss rates are often larger than this. Finally, note that 50% of all blocks replaced in the L2 cache are dirty. This is the origin of the 1.5 factor in the term for memory accesses.

$$\begin{aligned} \textit{avg. fetch time} &= \sum_{L1,\,L2,\,Memory} (\textit{portion of accesses} \times \textit{hit penalty}) \\ &= (1 - 0.02) \times 0\ ns + 0.02 \times 24.5\ ns + 0.02 \times (1 - 0.80) \times 1.5 \times 67.5\ ns \\ &= 0.895\ ns \times 1.1\ \textit{clock cycles / ns} \\ &= 0.985\ \textit{clock cycles} \end{aligned}$$

b. What is the average memory access time for data reads?

We compute this the same way as the time for instruction reads, with the same caveats.

$$\begin{aligned} \textit{avg. access time} &= \sum_{L1,\,L2,\,Memory} (\textit{portion of accesses} \times \textit{hit penalty}) \\ &= (1 - 0.02) \times 0\ ns + 0.02 \times 20.75\ ns + 0.02 \times (1 - 0.80) \times 1.5 \times 67.5\ ns \\ &= 0.820\ ns \times 1.1\ \textit{clock cycles / ns} \\ &= 0.902\ \textit{clock cycles} \end{aligned}$$

c. What is the average memory access time for data writes?

This computation is similar to those above, except that we must include the effect of the write buffer on misses in the L1 data cache. This is the origin of the 1-0.95 term in the calculation of the traffic going to the L2 cache, and consequently the main memory.

$$\begin{aligned} \textit{avg. access time} &= \sum_{L1,\,L2,\,Memory} (\textit{portion of accesses} \times \textit{hit penalty}) \\ &= (1 - 0.05) \times 0\ ns + 0.05 \times (1 - 0.95) + 17\ ns + \\ &\quad 0.05 \times (1 - 0.95) \times (1 - 0.80) \times 1.5 \times 67.5\ ns \\ &= 0.931\ ns \times 1.1\ \textit{clock cycles / ns} \\ &= 1.02\ \textit{clock cycles} \end{aligned}$$

d. What is the overall CPI, including memory accesses?

The overall CPI is sum of the CPIs for each type of instruction, scaled by their portion of the total instruction mix.

$$\begin{aligned} overall\_CPI &= instruction\ fetch\ time + \sum_{computation,\ D\ reads,\ D\ writes} (portion\ of\ instructions \times CPI) \\ &= 0.985\ CPI + 0.75 \times 0.700\ CPI + 0.20 \times 0.902\ CPI + 0.05 \times 1.02\ CPI \\ &= 1.74\ \text{CPI} \end{aligned}$$

e. You are considering replacing the 1.1 GHz CPU with one that runs at 2.1 GHz, but is otherwise identical. How much faster does the system run with a faster processor?

The system specifications remain the same as detailed above, except for the following:

- Processor information
  - 2.1 GHz (0.476 ns equivalent)

Doing the same calculations as above, but with 0.476 ns substituted for 0.909 ns, we get

$$\begin{aligned} avg.\ fetch\ time &= 0.895\ ns \times 2.1\ clock\ cycles/ns = 1.88\ clock\ cycles \\ avg.\ access\ time_{data\ reads} &= 0.820\ ns \times 2.1\ clock\ cycles/ns = 1.72\ clock\ cycles \\ avg.\ access\ time_{data\ writes} &= 0.931\ ns \times 2.1\ clock\ cycles/ns = 1.96\ clock\ cycles \\ overall\_CPI &= 1.88\ CPI + 0.75 \times 0.700\ CPI + 0.20 \times 1.72\ CPI + 0.05 \times 1.96\ CPI \\ &= 2.85\ CPI \end{aligned}$$

f. If you want to make your system run faster, which part of the memory system would you improve? Graph the change in overall system performance, holding all parameters fixed except the one that you are improving. Based on these graphs, how could you best improve overall system performance with minimal cost?

I am not supplying a solution to this, as there are so many ways to answer it. One issue is how you measure the cost of changing each system parameter. Since there is no way to know the real cost, the student should make an argument to modify the parameters that show the best improvement per unit change.

5.5 This problem explores the difference between the misses per reference and misses per instruction. The last paragraph on page 396 discusses the most important issues. I suggest the instructor can point out this paragraph to any student who does not understand this problem.

a. What is different about writing misses per instruction as miss rate times the factor memory accesses per instruction?

Accesses per instruction represents an average rate commonly measured over an entire benchmark or set of benchmarks. However, memory access and

instruction commit counts can be taken from collected data for any segment of any program's execution. Essentially, average accesses per instruction may be the only measure available, but because it is an average, it may not correspond well with the portion of a benchmark, or a certain benchmark in a suite of benchmarks that is of interest. In this case, getting exact counts may result in a significant increase in the accuracy of calculated values.

b. Convert the formula for misses per instruction on page 396 into one that uses only miss rate, references per instruction fetched, and fraction of fetched instructions that commit. Why rely upon these factors rather than those in the formula on page 396?

To make this clearer, I will substitute "instruction(s) committed" for "instruction(s)."

From page 396 in the text:

$$\frac{Misses}{Instruction\ committed} = \frac{Miss\ rate \times Memory\ accesses}{Instructions\ committed}$$

$$= \frac{Miss\ rate \times Memory\ accesses}{Instructions\ fetched \times \frac{\%\ of\ instructions\ committed}{100}}$$

$$= Miss\ rate \times \frac{Memory\ accesses}{Instruction\ fetched} \times \frac{Instructions\ fetched}{Instructions\ committed}$$

c. Rewrite the formula in part (b) to correct this deficiency.

The measurement "memory accesses per instruction fetched" is an average over all fetched instructions. It would be more accurate, as mentioned in the answer to part (a), to measure the exact access and fetch counts. Suppose we are interested in the portion of α of a benchmark.

$$\frac{Misses}{Instruction\ committed} = Miss\ rate \times \frac{Memory\ accesses_{\alpha}}{Instructions\ fetched_{\alpha}} \times \frac{100}{\%\ of\ instructions\ that\ commit_{\alpha}}$$

$$= Miss\ rate \times \frac{Memory\ accesses_{\alpha}}{Instructions\ fetched_{\alpha}} \times \frac{Instructions\ fetched_{\alpha}}{Instructions\ committed_{\alpha}}$$

$$= Miss\ rate \times \frac{Memory\ accesses_{\alpha}}{Instructions\ committed_{\alpha}}$$

d. When simulating using SimpleScalar, does miss rate change when switching between in-order and out-of-order execution?

SimpleScalar 2.0 reports the miss rate as the ratio of all memory accesses that missed in the cache over all memory accesses attempted. Since speculative accesses are included in this measure, all non-trivial benchmarks should give different miss-rate results. Whether these miss rates are higher or lower than those reported by the non-speculative simulator are depends on whether or not the incorrectly speculated memory accesses tend to pollute the cache.

5.6 The merging write buffer links the CPU to the write-back L2 cache. Two CPU writes cannot merge if they are to different sets in L2. So, for each new entry into the buffer a quick check on only those address bits that determine the L2 set number need be performed at first. If there is no match in this "screening" test, then the new entry is not merged. If there is a set number match, then all address bits can be checked for a definitive result.

As the associativity of L2 increases, the rate of false positive matches from the simplified check will increase, reducing performance.

5.7 With a separate victim cache and write buffer, evicting a dirty block results in the block being copied to the write buffer and not the victim cache. Eviction of a clean block results in the block being moved to the victim cache. On a cache miss, the victim cache and the write buffer has to be searched for the needed block. Both evictions and cache misses require a multiplexer to move data to the proper location in the cache. The advantage of a unified victim cache/buffer would be the resulting simplification of searching one structure instead of two. Another advantage is removing the possibility of duplicate entries. After a dirty block is written to the write buffer, it may be needed again. If the block remains clean until its eviction, then it would be moved to the victim cache. Given a large enough write buffer, the block may still be there. The disadvantage is that the unified structure now has to track which entries are dirty. Further, it has to search for entries that need to be written into memory.

5.8 Original Code:

```
For(i=0; i<3; i=i+1)
   For(j=0; j<100; j=j+1)
      A[i][j] = B[j][0] * B[j+1][0];
```

New prefetch version to reduce misses and extraneous prefetches:

```
Prefetch(A[0][0])...Prefetch(A[0][4]); /*Skipping odd indices*/
Prefetch(A[1][0])...Prefetch(A[1][4]); /*Skipping odd indices*/
Prefetch(A[2][0])...Prefetch(A[2][4]); /*Skipping odd indices*/
Prefetch(B[0][0])...Prefetch(B[6][0]);

For(j=0; j<93; j=j+2) {
   Prefetch(B[j+7][0]);
   Prefetch(B[j+8][0]);
   Prefetch(A[0][j+6]); /*(j+7) is the next element in the block*/
   A[0][j] = B[j][0] * B[j+1][0];
   A[0][j+1] = B[j+1][0] * B[j+2][0];}

For(j=94; j<100; j=j+1)
   A[0][j] = B[j][0] * B[j+1][0];

For(i=1; i<3; i=i+1) {
   For(j=0; j<93; j=j+2) {
      Prefetch(A[i][j+6]);
      A[i][j] = B[j][0] * B[j+1][0];
      A[i][j+1] = B[j+1][0] * B[j+2][0];}
For(j=94; j<100; j=j+1)
   A[0][j] = B[j][0] * B[j+1][0];}
```

This new prefetch code has no cache misses or extraneous prefetches. We remove the initial misses by adding prefetches in the beginning of the code sequence. Note that for each prefetch of A we fetch two needed elements—this is why some loops have been unrolled twice. This removes extraneous prefetches for A. Also we do not have to prefetch exactly 7 elements ahead since other instructions add the needed latency.

The only overhead from the original loops is caused by the added prefetches. We have 16 initial prefetches. The first loop with prefetches executes 47 times, resulting in 141 prefetches. The second loop body with prefetches executes 94 (47×2) times, resulting in 94 prefetches. The total number of prefetches is 251. The original code required 2100 cycles to complete assuming no misses (p. 442). However with misses it completed in 27,200 cycles. This new code requires 2351 (2100 + 251) cycles to execute yielding a speedup of 11.6.

5.9 No solution provided.

5.10 No solution provided.

5.11 No solution provided.

5.12 The interaction of the cache storage organization and replacement policy with the specifics of program memory access patterns yields cache behavior that can only be called complex.

a. Looking at the surface of the three C's cache miss model, a fully associative cache *should* have fewer non-compulsory misses (capacity plus conflict) than an equal size direct mapped cache because conflict misses are in addition to capacity misses and occur only in set associative or direct mapped caches. Capacity misses are defined as those misses in a fully associative cache that occur when a block is retrieved any time(s) after its initial compulsory miss.

   The genesis of the opportunity for a small direct mapped cache to outperform an equally sized fully associative cache can be found in the question hiding within this definition of capacity misses and left begging for an answer. If fully associative cache capacity misses are caused by blocks being discarded before their final use, why are these blocks discarded and must otherwise equivalent set associative or direct mapped caches discard the same blocks at the same times during program execution?

   Blocks are discarded, or replaced, based on the decision of the replacement policy. This replacement decision is very important to cache performance. If the block chosen for replacement is not referenced in the future by the program, then no capacity miss or conflict miss can occur in the future. This is the ideal case. If the block chosen for replacement will be used again in the very near future or is used frequently, as compared to other candidate blocks for replacement, then the replacement choice is a poor one and cache performance will be generally worse than ideal.Because fully associative, set associative, and direct mapped caches have different block *placement* constraints, the block re-*placement* policy for one cache type cannot consider the same blocks for replacement as are considered by the same policy on another organizational type. To see this more clearly, consider an example.

Let a program loop access three distinct addresses, A, B, and C, and then repeat the sequence from A. The reference stream for this program at this point would look like this: ABCABCABCA . . . . To simplify the discussion we assume that the direct mapped and fully associative caches each can hold two blocks and that addresses A and C are from different cache block frames in memory but map to the same location in the direct mapped cache, while address B maps to the other location in the cache. If the replacement policy for the fully associative cache is LRU, then every reference generated by the loop is a miss. If the replacement policy for the direct mapped cache is LRU (a degenerate form to be sure, because with only one block in each set whatever blocks are in a direct mapped cache are all always "least recently used"), accesses to A or C will always miss, but we will always hit on B (ignoring its compulsory miss).

The replacement policy of a fully associative cache can cast its eye on all the blocks in the cache, and in our example, makes the worst possible choice for replacement from all the blocks every time. For the direct mapped cache this choice is also always the worst possible, but is limited to two of the three blocks by cache structure. The result is that the direct mapped cache performs better.

b. The three C's model considers the organization of the storage within a cache (fully associative, set associative, or direct mapped), but it does not address how the cache is managed when an access misses. If a block is to be allocated on a miss (read or write) and there is no currently invalid (empty) location in the cache available to hold the allocated block, then some valid block must be replaced to make room.

We can make four observations. First, what looks like a capacity miss may be just as reasonably viewed instead as a replacement policy error if a different replacement policy would prevent that miss. (The fully associative misses in the example from the solution to part (a) would vanish if the cache could hold three blocks instead of just two.) Second, conflict misses are very similar to capacity misses. If the capacity miss definition is changed to focus on the cache set, "If the cache set cannot all the blocks mapping to that set during execution of a program, set *capacity* misses will occur" then conflict misses are capacity misses for the narrower portion of the cache. Third, the storage organization of set associative or direct mapped caches limits the scope of blocks considered by the replacement policy. Finally, limiting the scope of a replacement policy to a set can change its effectiveness for the better (see part (a)), so the non-compulsory misses experienced by set associative and direct mapped caches are not necessarily a superset of the non-compulsory misses of a fully associative cache as assumed by the three C's model. Thus, the three C's model does not provide a clean distinction between capacity and conflict misses. This means non-compulsory misses in the three C's model cannot in general be definitively classified.

So, replacement policy seems to fit into the capacity and conflict miss realm of the three C's model. However, replacement policy effects blur the distinction between these two miss categories. Only if there is one optimal replacement policy can replacement error misses be readily separated from capacity misses.

c. For the example in the answer to part (a), if the fully associative cache simply never replaced after loading the first two references of the loop, it would hit on two out of every three loop references and beat the performance of the direct mapped cache.

5.13 a. Allocating space in the cache for an instruction that is used infrequently or just once means that the time taken to bring its block into the cache is invested to little benefit or no benefit, respectively. If the replaced block is heavily used, then in addition to allocation cost there will be a miss penalty that would not have been incurred if there were a no-allocate decision.

b. Such a cache does not fit into the model as stated. More sophisticated allocation strategies could be readily added by expanding Question 3 from "Which block should be replaced on a cache miss?" to "Which cache block, if any, should be replaced on a read miss or on a write miss?"

5.14 This problem discusses the issues affecting the choice of whether or not to include way prediction on the RS10000.

For information about the MIPS R10000 processor, see the web page *http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=hdwr&db=bks&docnumber=007-2490-001*. Of particular interest are Section 3.2, page 39: Secondary Cache Interface Signals and Section 5.4, page 64: Secondary Cache Way Prediction Table. The instructor may wish to direct students to this web page in order to help them better understand the issues in this problem, before they attempt to answer it.

a. How can way prediction reduce the number of pins needed on the R10K package to read L2 tags and data, and what is the impact on performance compared to a package with a full complement of pins to interface to the L2 cache?

I assume that a full complement of pins to interface to the L2 cache means that both the data and tag information from both ways of the L2 cache can be brought on-chip at the same time, with the total latency being the same as the time to bring the data and tag from way predicted by the way predictor on chip. We thus have one-half of the number of pins required. Since each way requires at least—I have included only the absolutely necessary pins from the specification—138 bits for data, 19 bits for set address (index), and 33 bits for the set tag, the number of pins saved is significant.

Note that one possible implementation would be to bring both tags on-chip at one time, and only the data from the predicted side. Depending on the method of handling L2 cache misses (i.e., if a request for the data is not sent to memory until a miss has been established,) this can lower the latency for those misses.

For each of the two cases, we have several steps that can add latency:

1. Both sets of tags and data are brought onto the processor at once:
   a. Request and wait for the tags (also request the data) for both ways.
   b. Perform the tag comparisons. (We now have to do two comparisons, introducing another issue—are there two sets of comparison hardware? I will ignore this by assuming we can do both in the time taken by one.)
2. The predicted tag and data is brought onto the processor first:
   a. Predict the way.
   b. Request and wait for the tag (also request the data) of the predicted way.
   c. Perform the tag comparison. If the tag matches, use the data. If not, continue.
   d. Request and wait for the tag (also request the data) for the non-predicted way.
   e. Perform the tag comparison.

The performance difference between the two can depend on both the accuracy of the way predictor and whether or not the access hits in the L2 cache, if the request to the next level of the memory hierarchy is delayed until a hit/miss is established. I will ignore the latter. Each time the way predictor is incorrect, an additional penalty of steps 2.c to 2.e is incurred. Thus, if the way predictor is perfect, the performance is the same as for the larger-pin-count implementation. Each time the way predictor is incorrect, the time to get the data doubles, and the traffic on the buses to the L2 cache doubles.

b. What is the performance drawback of just using the same smaller number of pins but not including way prediction?

   Assuming that the way predictor works, the number of times that the access time doubles (see the answer to part a) will increase. We might assume that the accuracy will now be equivalent to a way predictor that is correct 50% of the time. However, due to the temporal and spatial locality of memory references, the accuracy may be better. Deciding whether the increase in accuracy provided by the way predictor is worth the extra hardware/complexity is an excellent example of what computer architects must do when designing a processor. Simulation of real-world benchmarks is often used to make this type of decision.

c. Assume that [sic] the R10K uses most-recently used way prediction. What are reasonable design choices for the cache state update(s) to make when the desired data is in the predicted way, the desired data is in the non-predicted way, and the desired data is not in the secondary cache?

I believe that this answer must include information for updating both the L2 cache and the way predictor. For each case:

1. When the desired data is in the predicted way, mark the L2 cache way as most-recently used (MRU) and do not change the way predictor state.
2. When the desired data is in the non-predicted way, mark the L2 cache way as MRU and invert the way predictor state.
3. When the desired data is not in the secondary cache, bring the missed data into the L2 cache, mark it as MRU, and set the way predictor to point to its set.

d. If a 512 KB L2 cache has 64-byte blocks, how many way prediction table entries are needed? How would the R10K support this need?

This cache has 4K sets, each with two 64-byte blocks. Since the R10000 has 8K entries in its way prediction table, it would support it fine. Half of the entries would remain unused.

e. For a 4 MB L2 cache with 128-byte blocks, how is the usefulness of the R10K way prediction table analogous to that of a branch history table?

This cache has 4 times as many sets as the one in part d, or 16K. Since this is more sets than the way prediction table has entries, each way prediction table is going to be predicting the way for two sets. Because of this, it is possible for interference (also called conflicts) to occur. This is similar to what occurs in a branch history table, where size limitations can result in an entry being accessed for more than one static branch. As in the branch history table, the state updates that occur when one of these sets is accessed will interfere with the state used to make predictions for the other. Whether this interference is constructive or destructive depends on whether or not the two sets should be predicted the same.

Students wishing to learn more about this can study the branch prediction literature, specifically, the bi-mode branch predictor, which attempts to group interfering branches so that their state updates will always be constructive. See Chih-Chieh Lee, I-Cheng K. Chen, Trevor N. Mudge, "The Bi-Mode Branch Predictor," 30th International Symposium on Microarchitecture, pp. 4–13, Dec. 1997.

5.15 No solution provided.

5.16 a. Program basic blocks are often short (less than 10 instructions). Even program run blocks, sequences of instructions executed between branches, are not very long. Prefetching obtains the next sequential block, but program execution does not continue to follow locations PC, PC + 4, PC + 8, . . . , for very long. So as blocks get larger the probability that a program will not execute all instructions in the block, but rather take a branch to another instruction address, increases. Prefetching instructions benefit performance when the program continues straight-line execution into the next block. So as instruction cache blocks increase in size, prefetching becomes less attractive.

b. Data structures often comprise lengthy sequences of memory addresses. Program access of a data structure often takes the form of a sequential sweep. Large data blocks work well with such access patterns, and prefetching is likely still of value due to the highly sequential access patterns.

5.17 a. The Alpha 21264 TLBs are fully associative. Given a tag, each entry in the TLB is searched for a matching entry. However, searching 128 entries is slow and adds complexity to the hardware. Reducing the number of entries to search decreases the time needed to find a match. A two-way set-associative cache accomplishes this goal. In order to use such a cache for a TLB, one needs to split the virtual page number (VPN) into two strings. One string is the index into the cache, and the other string is compared against the tags in the particular set pointed to by the index. Keeping the number of entries the same, the resulting cache structure consists of 64 sets with two entries each. Therefore, the size of the index is 6 bits leaving 29 bits for each entry's tag (35 bits for the VPN – 6 bits). However the use of this cache scheme would result in conflict misses. A program might have several frequently used TLB entries that map into the same set which results in trashing. In order to disperse the cache indices and decrease conflict misses, the 6-bit index is taken out of the lower portion of the VPN.

b. The tag length in the Alpha 21264 TLB is set at 35 bits. In order for the TLB to map more than 8 KB, i.e., multiple page sizes, it has to support partial tag matching. For instance, a 4MB mapping would only require a 26 bit entry (a 4MB page takes 9 less bits than an 8KB page, this results in a 26 bit entry (35–9)). The appropriate logic would look to see how many bits of the entry it would need to compare to the tag in order to find a match. With a two-way set associative cache, the task becomes difficult. There is a gap between the index (as computed in part a) and the last valid bit of a tag for a 4MB entry—3 bits to be exact. One possible solution is to search 8 sets. Another possible solution is to have the OS duplicate the entry 8 times, but this would effectively reduce the overall size of the TLB. Searching 16 entries in parallel might be a bit impractical. However 8 entries are searched in parallel in the virtually indexed data cache to solve the synonym problem.

5.18 a. The 21264 does not allow writes into the executable segment of the program. The instruction cache is read only. However multiple virtual pages might refer to the same physical page as is the case with shared libraries. Even if multiple aliases are present in the instruction cache, they would all return the same value since no process can write to the cached values.

b. The problem states that we have 64-bit virtual addresses, 8-bit process identifiers, and 64GB of main memory. It takes 36-bits to address a location in main memory. Since we do not have any information about cache sizes, we can only say that a virtual address tag would require 28-bits more in space than a physical address tag. In addition, we now have to store an 8-bit process tag since programs can have virtual addresses in common. This gives a total of 36-bits in overhead. If we keep the page size constant, changing the cache

block size does not have an effect on the overhead. It would only serve to decrease the number of bits used for the cache index.

5.19 a. We can expect software to be slower due to the overhead of a context switch to the handler code, but the sophistication of the replacement algorithm can be higher for software and a wider variety of virtual memory organizations can be readily accommodated.

Hardware should be faster, but less flexible.

b. Factors other than whether miss handling is done in software or hardware can quickly dominate handling time. Is the page table itself paged? Can software implement a more efficient page table search algorithm than hardware? What about hardware TLB entry prefetching?

c. Page table structures that change dynamically would be difficult to handle in hardware but possible in software.

d.

Program	Weight	TLB misses/1000 instructions
gcc	50%	0.30
perl	25%	0.26
ijpeg	25%	0.10

$$\begin{aligned}\text{Workload miss rate} &= \sum_i \text{Weight}_i \times (\text{TLB misses/1000})_i \\ &= 0.5 \times 0.30 + 0.25 \times 0.26 + 0.25 \times 0.10 \\ &= 0.22/1000 \text{ instructions}\end{aligned}$$

Cost of a hardware handler is given as 10 cycles/miss, so penalty is 2.175 cycles/1000 instructions and the effect on CPI is an increase of 0.0022 clocks/instruction.

Similarly, for a software handler costing 30 cycles/miss, the effect on CPI is 0.0066 clocks/instruction.

The second workload is

Program	Weight	TLB misses/1000 instructions
swim	30%	0.10
wave5	30%	0.89
hydro2d	20%	0.19
gcc	10%	0.30

$$\text{Workload miss rate} = \sum_i \text{Weight}_i \times (\text{TLB misses/1000})_i = 0.37$$

The hardware penalty is 0.0037 clocks/instruction. The software penalty is 0.011 clocks/instruction.

e. The TLB miss times are too small. Handling a TLB miss requires finding and transferring a page table entry in main memory to the TLB. A main memory access typically takes on the order of 100 clocks, already much greater than the miss times in part (d).

f. Floating-point programs often traverse large data structures and thus more often reference a large number of pages. It is thus more likely that the TLB will experience a higher rate of capacity misses.

5.20 No solution provided.

5.21

5.22 No solution provided.

5.23 a. The answer to the question is how many distinct 16-byte blocks do the 100,000 instruction references refer to, or touch, in the memory address space. Each distinct block referenced will contribute a compulsory miss to the total.

If we assume that a 16-byte block can hold at least two instructions, then we can imagine a program with a tight loop that touches only a single block during the first 100,000 instructions fetched. In this case there would be only one compulsory miss. It is not possible to reference 1 or more instructions without generating a compulsory miss, so 1 such miss is the minimum number possible.

The total number of distinct 16-byte blocks in the memory space is simply the size of the address space in bytes divided by 16. The number of 16-byte blocks in the 8 KB direct-mapped cache is 512. With a 4-byte instruction size, such as for MIPS, straight line code (sequential instruction access, no branching) will touch 25,000 distinct blocks in 100,000 instruction references, if the address space is sufficiently large. Thus, all 600 of the misses could be compulsory. So, in summary, the number of misses that are compulsory ranges from a low of 1 to a high of 600.

b. After the first 100,000 memory references, there are no more compulsory misses for the rest of the program's execution. Assuming no more misses, there are only 600 misses total as given by part a—yielding a final miss rate of 600/1,000,000 or .06%. If it is assumed that the rest of the references are misses, then the final miss rate is (600 + 900,000)/1,000,000 or 90%. However these misses would have to entirely come from either capacity or conflict misses.

c. Reducing the number of blocks for a given block size and set associativity results in a smaller cache size. This has the potential effect of increasing capacity and conflict misses. By definition, this cannot change the compulsory miss rate. Compulsory misses are calculated using an infinite sized cache. Since we have already computed the worst case miss rate in part b, the range of miss rates remain the same.

d. If the memory trace is lengthened, then the worst case miss rate would increase and the best case miss rate would decrease.

5.24

5.25 Out-of-order (OOO) execution will change both the timing of and sequence of cache accesses with respect to that of in-order execution. Some specific differences and their effect on what cache design is most desirable are explored in the following.

Because OOO reduces data hazard stalls, the pace of cache access, both to instructions and data, will be higher than if execution were in order. Thus, the pipeline demand for available cache bandwidth is higher with OOO. This affects cache design in areas such as block size, write policy, and prefetching.

Block size has a strong effect on the delivered bandwidth between the cache and the next lower level in the memory hierarchy. A write-through write policy requires more bandwidth to the next lower memory level than does write back, generally, and use of a dirty bit further reduces the bandwidth demand of a write-back policy. Prefetching increases the bandwidth demand. Each of these cache design parameters—block size, write policy, and prefetching—is in competition with the pipeline for cache bandwidth, and OOO increases the competition. Cache design should adapt for this shift in bandwidth demand toward the pipeline.

Cache accesses for data and, because of exceptions, instructions occur during execution. OOO execution will change the sequence of these accesses and may also change their pacing.

A change in sequence will interact with the cache replacement policy. Thus, a particular cache and replacement policy that performs well on a chosen application when execution of the superscalar pipeline is in order may perform differently—even quite differently—when execution is OOO.

If there are multiple functional units for memory access, then OOO execution may allow bunching multiple accesses into the same clock cycle. Thus, the instantaneous or peak memory access bandwidth from the execution portion of the superscalar can be higher with OOO.

Imprecise exceptions are another cause of change in the sequence of memory accesses from that of in-order execution. With OOO some instructions from earlier in the program order may not have made their memory accesses, if any, at the time of the exception. Such accesses may become interleaved with instruction and data accesses of the exception-handling code. This increases the opportunity for capacity and conflict misses. So a cache design with size and/or associativity to deliver lower numbers of capacity and conflict misses may be needed to meet the demands of OOO.

5.26 No solution provided.

5.27 No solution provided.

5.28 Given a new technology called magnetic RAM (MRAM), we are asked to propose applications for two different configurations of MRAM. The first configura-

tion has a high data density resulting in a slow update speed. Mobile devices such as personal data assistants (PDAs) and high-end cell phones would be an ideal use of this configuration. The operating system and user programs for these devices have to be stored in non-volatile memory that can be updated. When program updates or security patches become available, it is highly desirable to download them instead of replacing the device. The slow write speed of this MRAM configuration is tolerable since these events happen infrequently (one would hope!). The second MRAM configuration has a low data density format but a faster memory access time. A possible use for this configuration is in smart cards. These cards are used to pay for goods and services while protecting the card from fraudulent uses. It would be very useful for the card to "download" the receipt when it is used to pay for the goods or services instead of waiting for a paper copy. Such a receipt typically has a small memory footprint and the download time would have to be fast. Another possible use is for write buffers to hard drives. Since writing to disk is relatively slow, it is desirable to write the incoming data to a faster temporary storage space that can handle the I/O bandwidth. Since the power may fail at any time, this buffer has to be non-volatile in order to prevent data loss.

5.29 No solution provided.

Chapter 6 Solutions

6.1 The general form for Amdahl's Law (as shown on the inside front cover of this text) is

$$\text{Speedup} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}}$$

all that needs to be done to compute the formula for speedup in this multiprocessor case is to derive the new execution time.

The exercise states that for the portion of the original execution time that can use i processors is given by $F(i,p)$. If we let $\textit{Execution time}_{old}$ be 1, then the relative time for the application on p processors is given by summing the times required for each portion of the execution time that can be sped up using i processors, where i is between 1 and p. This yields

$$\text{Execution time}_{\text{new}} = \sum_{i=1}^{p} \frac{F(i,p)}{i}$$

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

6.2 No solution provided.

6.3 No solution provided.

6.4 To keep the figures from becoming cluttered, the coherence protocol is split into two parts as was done in Figure 6.11 in the text. Figure S.32 presents the CPU portion of the coherence protocol, and Figure S.33 presents the bus portion of the protocol. In both of these figures, the arcs indicate transitions and the text along each arc indicates the stimulus (in normal text) and bus action (in bold text) that occurs during the transition between states. Finally, like the text, we assume a write hit is handled as a write miss.

Figure S.32 presents the behavior of state transitions caused by the CPU itself. In this case, a write to a block in either the invalid or shared state causes us to broadcast a "write invalidate" to flush the block from any other caches that hold the block and move to the exclusive state. We can leave the exclusive state through either an invalidate from another processor (which occurs on the bus side of the coherence protocol state diagram), or a read miss generated by the CPU (which occurs when an exclusive block of data is displaced from the cache by a second block). In the shared state only a write by the CPU or an invalidate from another processor can move us out of this state. In the case of transitions caused by events external to the CPU, the state diagram is fairly simple, as shown in Figure S.33. When another processor writes a block that is resident in our cache, we unconditionally invalidate the corresponding block in our cache. This ensures that the next time we read the data, we will load the updated value of the block from memory. Also, whenever the bus sees a read miss, it must change the state of an exclusive block to shared as the block is no longer exclusive to a single cache.

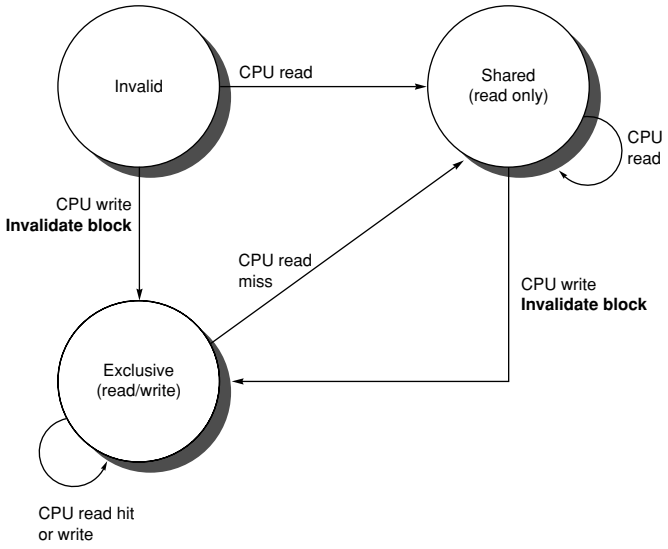

Figure S.32 CPU portion of the simple cache coherency protocol for write-through caches.

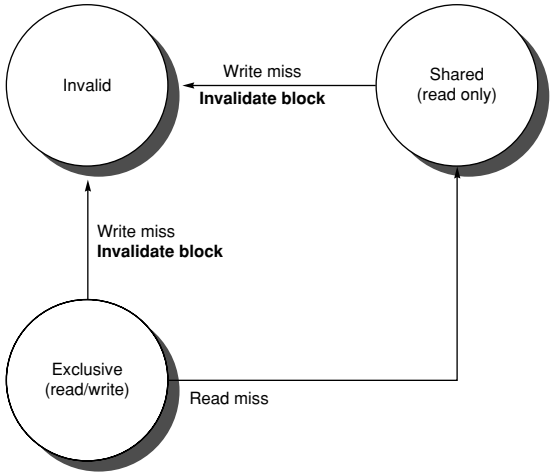

Figure S.33 Bus portion of the simple cache coherency protocol for write-through caches.

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

6.5 To augment the snooping protocol of Figure 6.12 with a Clean Exclusive state we assume that the cache can distinguish a read miss that will allocate a block destined to have the Clean Exclusive state from a read miss that will deliver a Shared block. Without further discussion we assume that there is some mechanism to do so.

The three states of Figure 6.12 and the transitions between them are unchanged, with the possible clarifying exception of renaming the Exclusive (read/write) state to Dirty Exclusive (read/write).

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

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

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

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

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

6.6 An obvious complication introduced by providing a valid bit per word is the need to match not only the tag of the block but also the offset within the block when snooping the bus. This is easy, involving just looking at a few more bits. In addition, however, the cache must be changed to support write-back of partial cache blocks. When writing back a block, only those words that are valid should be written to memory because the contents of invalid words are not necessarily coherent with the system. Finally, given that the state machine of Figure 6.12 is applied at each cache block, there must be a way to allow this diagram to apply when state can be different from word to word within a block. The easiest way to do this would be to provide the state information of the figure for each word in the block. Doing so would require much more than one valid bit per word, though. Without replication of state information the only solution is to change the coherence protocol slightly.

6.7

6.8 a. The instruction execution component would be significantly sped up because the out-of-order execution and multiple instruction issue allows the latency of this component to be overlapped.

The cache access component would be similarly sped up due to overlap with other instructions, but since cache accesses take longer than functional unit latencies, they would need more instructions to be issued in parallel to overlap their entire latency. So the speedup for this component would be lower.

The memory access time component would also be improved, but the speedup here would be lower than the previous two cases. Because the memory comprises local and remote memory accesses and possibly other cache-to-cache transfers, the latencies of these operations are likely to be very high (100's of processor cycles). The 64-entry instruction window in this example is not likely to allow enough instructions to overlap with such long latencies. There is, however, one case when large latencies can be overlapped: when they are hidden under other long latency operations. This leads to a technique called miss-clustering that has been the subject of some compiler optimizations.

The other-stall component would generally be improved because they mainly consist of resource stalls, branch mispredictions, and the like. The synchronization component if any will not be sped up much.

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

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

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

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

Padding a cache block frame containing a shared data object with unused memory locations may lead to rather inefficient use of memory space. A cache block may contain a shared object plus objects that are read-only as a trade-off between memory use efficiency and incurring some false-sharing misses. This optimiza-

tion almost certainly require programmer analysis to determine if it would be worthwhile.

Generally, careful attention to data distribution with respect to cache lines and partitioning the computation across processors is needed.

6.10 The problem illustrates the complexity of cache coherence protocols. In this case, this could mean that the processor P1 evicted that cache block from its cache and immediately requested the block in subsequent instructions. Given that the write-back message is longer than the request message, with networks that allow out-of-order requests, the new request can arrive before the write back arrives at the directory. One solution to this problem would be to have the directory wait for the write back and then respond to the request. Alternatively, the directory can send out a negative acknowledgment (NACK). Note that these solutions need to be thought out very carefully since they have potential to lead to deadlocks based on the particular implementation details of the system. Formal methods are often used to check for races and deadlocks.

6.11

6.12 a. The average miss time is given by a weighted sum of the various miss times (local, dirty, and unowned). Figure 6.50 gives a miss time of 342 ns for unowned blocks and 482 ns for dirty blocks measured by processor restart. Note that local memory times are used since this problem involves a bus based distributed memory system (as opposed to going over the network to service misses).

$$\begin{aligned}\text{Average miss time} &= 60\%*100\text{ ns} + 40\%*(20\%*342\text{ ns} + 80\%*482\text{ ns}) \\ &= 242\text{ ns}\end{aligned}$$

The first term represents the weighted miss time of local misses. The remaining 40% is split between unowned blocks and dirty data respectively.

b. At a 1 GHz, the average miss time translates into a 242 cycle stall on average for a miss in memory (including both loads and stores). Assuming that every other instruction is either a load or a store, the new CPI is given below.

$$\begin{aligned}\text{CPI}_{\text{effective}} &= 1 + 50\% * 2\% * 242\text{ (cycles/inst)} \\ &= 3.42\text{ (cycles/inst)}\end{aligned}$$

The frequency of loads/stores is $1/(2 * \text{CPI}_{\text{effective}} * (1\text{ ns/cycle})) = 146.2$ MHz.

c. The bandwidth requirement is given by isolating the external memory transitions from the load/store frequency calculation from part b and multiplying it with the number of bytes consumed by each bus transaction (one request, one ack, and one data transfer).

$$\begin{aligned}\text{Processor Bandwidth} &= 146.2\text{ MHz} * (2\% * 40\%) * (16\text{ B} + 16\text{ B} + 80\text{ B}) \\ &= 131\text{ MB/s}\end{aligned}$$

d. The total bus bandwidth for an E6000 node is 2.7 GB/s as given by Figure 6.51. Since the processor requires only 131 MB/s in bandwidth (from part c), one does not have to worry about maxing out the sustainable bandwidths for

unowned and dirty transfers. The total number of processors is given by 2700/131 = 21.

6.13 a. Figure 6.50 presents two cases for average remote access time where the accessed remote block is in its home node: (1) shared or invalid in the home caches (called unowned) and (2) dirty. The less likely situation of a remote access to a clean block in the exclusive state within its home node is ignored.

An important assumption in this exercise is to use the average remote memory latency data in the second section of Figure 6.50. A more realistic situation would have some nonzero fraction of remote accesses of the three-hop variety.

With only unowned and dirty blocks, average remote access time is a function of the fractions of accesses that are unowned and dirty.

$$\text{Wildfire}_{\text{avg}} = (1 - 0.8) \times 1774 \text{ ns} + (0.8) \times 2162 \text{ ns} = 2056 \text{ ns}$$

$$\text{Origin}_{\text{avg}} = (1 - 0.8) \times 973 \text{ ns} + (0.8) \times 1531 \text{ ns} = 1419 \text{ ns}$$

b. For unowned blocks, the difference in average remote access time between Wildfire and Origin is 1774 ns – 973 ns = 801 ns. For dirty blocks, the difference is 2162 ns – 1531 = 631 ns. Thus, to minimize the difference in average remote access time all remote accesses should be to dirty blocks. Any fraction of accesses to unowned blocks will only increase the difference.

c. If all accesses are either unowned or dirty, then

$$1 - \text{Fraction dirty} = \text{Fraction unowned}$$

Using the data from the first section of Figure 6.51, the average bandwidth measurements for mixtures of local accesses on the two machines will be

$$\text{Wildfire}_{\text{avg local BW}} = \text{Fraction unowned} \times 312 \text{ MB/sec} + \text{Fraction dirty} \times 246 \text{ MB/sec}$$

$$\text{Origin}_{\text{avg local BW}} = \text{Fraction unowned} \times 554 \text{ MB/sec} + \text{Fraction dirty} \times 182 \text{ MB/sec}$$

Setting the two machine bandwidths equal, substituting 1 – Fraction dirty for Fraction unowned and solving yields 79% dirty accesses.

6.14 Local: LMAT = 300 + 20*(n–1)
Remote: ARAT = LMAT + 100 + 100*SQRT(128/n)

a. The problem asks us to compute the optimum node size under a uniform distribution of remote accesses. As mentioned in the problem, the goal is to minimize the average remote access time (ARAT). One can take the derivative and set it to zero to find the number of nodes that minimizes the ARAT. Otherwise, one could use a spreadsheet to compute the range of ARAT values with respect to n. For this particular set of equations, n = 9.

b. As it stands, we have to change the local access time to make a one node implementation faster than the optimal node size from part (a). In order for such a design to be faster, this equation has to hold:

$$\text{ARAT}_{\text{one-node}} <= \text{ARAT}_{\text{nine-node}}.$$

However this equation is not true, $LMAT_{one\text{-}node}$ would have to be a negative value. Therefore, the remote access time has to change in order for a one-node design to be effective.

c. We have to minimize the Average Memory Access Time (AMAT) given a distribution of remote accesses. AMAT = (P)*LMAT + (1–P)*ARAT. P is given by the table in the problem. Solving for *n* under these conditions, *n* = 8 at 627.6 ns.

6.15 The problem asks us to resolve problem 6.14c but using a difference remote memory access time. The new time is local memory access time(LMAT) + 200 ns.

The formula used to compute the table below is:

AMAT = LMAT*40% + (Pintranode*LMAT + (1–Pintranode)*(LMAT+200 ns))*60%
LMAT = 300 + 20(*n*–1)ns

Pintranode is the probability of the request between inside the node as given in 6.14c. The number of processors equal *n*. The intranode access time with *n* processors is 300 + 20(*n*–1)ns

Processors per node	Average Memory Access Time
2–3	410–430
4–7	420–480
8–15	485–625
16–31	630–930
32-63	935–1555
>=64	1560

Clearly, the optimal number of processors per node is 2.

6.16

6.17 Executing the identical program on more than one processor improves system ability to tolerate faults. The multiple processors can compare results and identify a faulty unit by its mismatching results. Overall system availability is increased.

6.18 With a broadcast protocol for lock variables, every time a lock is released, the new value of the lock variable is broadcast to all processors. This means processors will never miss when reading the lock variable.

Assume that every cache has the block containing the lock variable initially. The table shows that the average will be 550 clocks over the course of all 10 lock/unlock pairs. The total time will be 5500 cycles.

Event	Duration
Read (hit) of lock by all processors	0
Write (miss) broadcast by releasing processor	100
Read (hit) of lock (processors see lock is free)	0
Write/swap broadcast by one processor plus nine extra write broadcasts	1000
Total time for one processor to acquire and release	1100

6.19

```
int n = 0;              /* Initialize array pointer/counter */
int release[nproc-1];   /* nproc = number of processors */
mythread()
{
...
barrier();
...
}
barrier()
{
    local_sense != local_sense;
    fetch.and.increment(n);
    if(n < nproc )
      spin(release[n-1] == local_sense);
     else {
      n = 0;
      for(i=0;i<nproc-1;i++)
        release[i] = local_sense;
    }
}
```

This simple code illustrates the basic behavior of a queuing lock used for a barrier. Each node spins on a unique memory location as determined by n. However, the last node does not enter the queue. Instead, it individually frees each node waiting in the queue. This requires n bus transaction for the fetch.and.increment, $n - 1$ read misses for the elements in the release array, and finally, $n - 1$ read misses for the elements in the array. In all, this requires $3n - 1$ bus transactions. However, a dedicated hardware approach might transfer both the address and memory value (as opposed to just the fetch.and.increment) to prevent the $n - 1$ read misses for the spinlock.

6.20

```
fetch_and_increment (count); /* atomic update */
if (count == total) {        /* all processes arrived */
    count = 0;               /* reset counter */
    release = 1;             /* release processes */
}
else {                       /* more to arrive at barrier */
    spin (release == 1);     /* wait for signal to proceed */
```

For n processes, this code executes n fetch-and-increment operations, n cache misses to access `release`, and $n - 1$ more cache misses when `release` is finally set to 1 by the last processor to reach the barrier. (The last processor to reach the barrier will hit in its cache when it reads `release` the first time when spinning.) This is a total of $3n - 1$ bus transactions. For 10 processors this is 29 bus transactions or 2900 clock cycles.

6.21 The following code takes the combining tree barrier implementation from Figure 6.42 and implements a combining tree for the release. Each node in the tree has its own release variable. Children wait on these variables instead of a global release variable--there by reducing read contention in the release process. The process that makes it into the next level of the tree is responsible for releasing its peers when it returns from the barrier operation.

```
struct node { /* a node in the combining tree */
    int counterlock; /* lock for this node */
    int count; /* counter for this node */
    int parent; /* parent in the tree = 0..P-1 except for the
      root */
    int release;  /* Each node has its own release variable */
};

struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */

barrier(int mynode) {
   lock(tree[mynode].counterlock); /* protect count */
   tree[mynode].count++;
   if(tree[mynode].count == k) {/* all arrived at mynode */
      if(tree[mynode].parent >= 0 )
         barrier(tree[mynode].parent); /* continue on to the
           next tree level */
      tree[mynode].release = local_sense;
      tree[mynode].count = 0; /* reset for next time */
   }
   unlock(tree[mynode].counterlock);
   spin(tree[mynode].release == local_sense); /* wait on peer
     to issue the release  */
}

/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

6.22 Assume a cache line that has a synchronization variable and the data guarded by that synchronization variable in the same cache line. Assume a two processor system with one processor performing multiple writes on the data and the other processor spinning on the synchronization variable. With an invalidate protocol, false sharing will mean that every access to the cache line ends up being a miss resulting in significant performance penalties

6.23 As with similar examples and exercises in the text, we ignore the time to actually read or write a lock. Each of the n processes requires c cycles to lock the counter associated with the barrier, update its value, and release the lock. In the worst case, all n processors simultaneously attempt to lock the counter and update the value. Because the lock serializes access to the counter, the processors update the counter one at a time. Thus, it takes nc cycles for all processors to arrive at the barrier.

6.24 The problem asks us to find the time for n processes to synchronize using a combining tree. We know from problem 6.23 that k processes take ck time to synchronize using a noncombining tree barrier, where c is the time it takes for one process to update the count and release the lock. Using a k-ary combining tree, we divide the problem into parallel subproblems at each level of the tree. Since each node has k children, it takes ck time to finish one layer of tree. Therefore, the running time for a combining tree implementation is ck*floor($\log_k n$). The floor is taken because the leaves of tree (the individual processes) do not require a lock.

6.25 No solution provided.

6.26 a. Prefetching is beneficial when data access patterns can be predicted well in advance of the actual need for the data—typically regular communication patterns such as strided or blocked array element accesses. Prefetching is inapplicable for irregular communication patterns, pipelined loops, and synchronization. In many of these cases, producer-initiated communication can be applicable. In particular, in the case of queued locks and tree barriers, the processor releasing the lock could identify the next processor waiting for the lock and initiate a producer-initiated transfer of the data, saving some additional spin time.

b. The systems spends its time as follows: 60% stalled on memory access, 20% stalled for synchronization, and 20% doing useful work. Producer-initiated communication will not reduce the time spent doing useful work. For the memory stall time 20%, or 12% of the total time, is wasted during producer-consumer access patterns where the writer can identify the next reader. These stalls can be improved from 300 cycles to 1 cycle. Another 30% of the memory stall time, or 18% of the total time, involves producer-consumer access patterns where the writer cannot identify the next reader. Whichever processor is the next reader, the stall time can be reduced from 300 cycles to 100 cycles if the writer postflushes the data from its cache to memory, making the data more quickly accessible for the reader. The remaining 50% of memory access stalls, or 30% of the total time are not helped by producer-initiated communication. Synchronization stalls comprise 20% of the total time, and producer-initiated communication can reduce that by 40%.

Thus,

$$\begin{aligned}\text{Execution time}_{\text{original}} &= \sum_i \text{fraction}_i \\ &= \text{Memory stalls} + \text{Synchronization stalls} + \text{Useful work} \\ &= (12\% + 18\% + 30\%) + 20\% + 20\% \\ &= 1\end{aligned}$$

$$\begin{aligned}\text{Execution time}_{\text{enhanced}} &= \sum_i \frac{\text{fraction}_i}{\text{speedup}_i} \\ &= \text{Memory stalls} + \text{Synchronization stalls} + \text{Useful work} \\ &= \left(\left(\frac{12\%}{300}\right) + 18\%\left(\frac{100}{300}\right) + 30\%\right) + 20\%(1 - 40\%) + 20\% \\ &= 0.68\end{aligned}$$

The total execution time reduction is 32% for a speedup of 47%.

c. The cache coherence protocol needs to be changed to include support for producer-initiated communication. However, these changes need to be done with care to avoid the possibility of deadlocks. The program software also needs to be modified to include user-specified hints for producer-initiated communication, just as with prefetching.

6.27 a. Given in Figure 6.31, the remote miss rate for Ocean running on 64 processors is 2.5% and the local miss rate is 4%. The average time to service a miss is 2.5%*100 + 4%*40 or 4.1 cycles. Now all of the remote misses can be handled by doing a DMA transfer. The problem states that this has a 10 cycle overhead and results in a cache miss. The new average time to service a miss is 2.5%*(40+10) + 4%*40 or 2.85 cycles. DMA reduces the miss time by a factor of 1.44!

b. No solution provided.

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

b. Case 1: If the write to flag reached P2 faster than the write to A.
Case 2: If the read to A was faster than the read to flag.

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

d. Here C is guarded by the flag variable. We need extra synchronization between the two variables.

```
P1                  P2
A = 2000            while (flag == 1) {;}
Barrier             Barrier
flag = 1            C = A
```

6.29 a. Destructive interference among the threads.

b. The processor will context switch on a branch misprediction. Consequently, even though there are more mispredicted branches, less time is spent executing instructions on these wrong paths.

c. Even though there are conflicts among the threads in the branch prediction mechanism, some of the performance impact of the conflicts is hidden by the latency-hiding potential of multithreading. As long as the conflicts are not too great, the performance loss is likely to be small.

6.30 Constructive cache interference is likely to happen when near-identical threads comprise the workload. Each thread will execute nearly identical code, so cache blocks become shared among the threads. Also, sharing of data among the threads leads to cache block reuse. Overall, the cache miss rates can be similar to that for a single thread with appropriate software-mapping policies for the threads.

6.31 a. This heuristic gives highest priority to threads that are moving instructions through the instruction queue most efficiently; prevents one thread from filling up all the slots in the decode, rename, and instruction queues; and provides a fairly even mix of instructions from the various threads.

b. Assume that seven of the eight threads are spinning on a synchronization variable that needs to be written by the eighth thread. The heuristic will give precedence to the spinning processes because they will have very few instructions stalled and thus present in the decode, rename, and instruction queues. Consequently, a potential exists to starve the eighth thread.

c. Some possibilities are round-robin, branch count, miss count, and stalled instruction count. See Tullsen et al. [1996] (23rd ISCA) for more details.

6.32 Inclusion states that each higher level of cache contains all the values present in the lower cache levels, i.e., if a block is in L1 then it is also in L2. The problem states that L2 has equal or higher associativity than L1, both use LRU, and both have the same block size. When a miss is serviced from memory, the block is placed into all the caches, i.e., it is placed in L1 and L2. Also, a hit in L1 is recorded in L2 in terms of updating LRU information. Another key property of LRU is the following. Let A and B both be sets whose elements are ordered by their latest use. If A is a subset of B such that they share their most recently used elements, then the LRU element of B must either be the LRU element of A or not be an element of A. This simply states that the LRU ordering is the same regardless if there are 10 entries or 100.

Let us assume that we have a block, D, that is in L1, but not in L2. Since D initially had to be resident in L2, it must have been evicted. At the time of eviction D must have been the least recently used block. Since an L2 eviction took place, the processor must have requested a block not resident in L1 and obviously not in L2. The new block from memory was placed in L2 (causing the eviction) and placed in L1 causing yet another eviction. L1 would have picked the least recently used block to evict. Since we know that D is in L1, it must be the LRU entry since it was the LRU entry in L2 by the argument made in the prior paragraph. This means that L1 would have had to pick D to evict. This results in D not being in L1 which results in a contradiction from what we assumed. If an element is in L1 it has to be in L2 (inclusion) given the problem's assumptions about the cache.

6.33 Longer latency means lower speedup for programs with more communication. Larger units of coherence means more false sharing and possibly the need for greater interconnection network bandwidth. To address the latency problem try a faster interconnection network, lower software overhead in the OS handler,

producer-initiated communication, prefetching, and reducing the number of messages by optimizing the program. To help with the negative effect of larger coherence units try to use the smallest unit of coherence possible, pad out data, and identify synchronization and data associated with synchronization to minimize false sharing.

6.34 a. Four cores, each 64 KB, means 256 KB of data. If that is duplicated in the 1 MB L2 cache, then that is a 25% upper bound on wasted capacity.

Effectively the L2 now behaves as a large victim cache for the four core L1 caches. Clean blocks that are replaced in an L1 cache cause a write back to the L2 cache.

With a multiprocessor system there are more issues related to coherence and allocation/replacement policies. We may need to keep a duplicate copy of the L1 tags and state at the L2 controllers. Additionally, we may also need to extend the state information in the L1 cache to include, for example, a notion of ownership.

6.35 Task scheduling is harder and more specialized with heterogeneous processors. Binary compatibility is very hard and typically not supported for such processor designs.

6.36 Writing programs for the heterogeneous chip is more difficult that for the homogeneous processor. The advantage of the heterogeneous chip is that there can be a close match between the hardware platform and the application. The drawback is the use of more programmer time to write and debug the heterogeneous code: there is more code (each different unit has its own code) and the code interactions are more complex. Binary software compatibility is less likely to be maintained for the heterogeneous system.

6.37 No solution provided.

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

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

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

6.38 No solution provided.

6.39 No solution provided.

6.40 No solution provided.

6.41 No solution provided.

6.42 No solution provided.

6.43 No solution provided.

6.44 No solution provided.

6.45 No solution provided.

Chapter 7 Solutions

7.1

Figure 7.2 data		Seagate	Travelstar	Microdrive
Number of cylinders		14100	21664	7167
Average read access seek (ms)		5.6	12	12
Minimum read access seek (ms)		0.6	2.5	1.0
Maximum read access seek (ms)		14	23	19
One-third cylinders seek distance		4700	7221	2389
Linear seek time model formula for seek time (ms)		15.6	31.0	34.0
Difference from manufacturer's reported average seek time (ms)		10.0	19.0	22.0
Chen and Lee (1995) model:				
Coefficients, to two significant digits				
	a =	0.022	0.091	0.30
	b =	0.00076	0.00033	−0.00098
	c =	0.60	2.5	1.0
Chen and Lee model formula for seek time (distance)		5.7	12.5	12.8
Difference from manufacturer's reported average seek time (ms)		0.09	0.50	0.81

The Chen and Lee coefficients, in combination with their nonlinear model, predict the seek times of the three disk drives of Figure 7.2 well, certainly much better than does the model that seek time is linear with seek distance.

7.2

```
#include <stdlib.h>
#include <math.h>

static double a, b, c;
static double minSeek, avgSeek, maxSeek;
static double coeff1;

double seekTime1 (int d)
{
  return (minSeek + ((double)d) * coeff1);
}
  double seekTime2 (int d)
{
  return (a * sqrt ((double)(d-1)) + b * ((double)(d-1)) + c);
}

/*-------------------------------------------------------------
 * Assume that seeks always go from the lower numbered to the
 * higher numbered track. This is reasonable because seek time
 * (from the formulas, anyway) is symmetric. Then, a seek of
 * length c cylinders is possible only from tracks 1-(cyl-c),
 * with tracks numbered 1-cyl. We use this to avoid calculating
 * seek time for all possible pairs, since our formulas show
 * (perhaps incorrectly) that all seeks of distance d take the
 * same time.
*-------------------------------------------------------------
*/
int main (int argc, char *argv[])
{
  int cyls;
  int i;
  double totalTime1 = 0.0, totalTime2 = 0.0;
  int totalSeeks = 0;

if (argc != 5) {
  printf ("Usage: %s <cylinders> <min seek> <avg seek>
    <max seek>\n",
   argv[0]);
  exit (-1);
}
cyls = atoi (argv[1]);
minSeek = atof (argv[2]);
avgSeek = atof (argv[3]);
maxSeek = atof (argv[4]);
a = (-10 * minSeek + 15 * avgSeek - 5 * maxSeek) /
    (3 * sqrt ((double)cyls));
b = (7 * minSeek - 15 * avgSeek + 8 * maxSeek) /
    (3 * (double)cyls);
```

```
c = minSeek;
coeff1 = (maxSeek - minSeek) * 1.0 / (double)cyls;
for (i = 1; i < cyls; i++) {
  totalTime1 += seekTime1 (cyls - i) * (double)i;
  totalTime2 += seekTime2 (cyls - i) * (double)i;
  totalSeeks += i;
}
printf ("Seek formula 1: average=%.2f ms, 1/3 seek=%.2f\n",
totalTime1 / (double)totalSeeks, seekTime1 (cyls/3));
printf ("Seek formula 2: average=%.2f ms, 1/3 seek=%.2f\n",
totalTime2 / (double)totalSeeks, seekTime2 (cyls/3));
}
```

Running this code on the disks in Figure 7.2 on page 682 gives:

Disk	Formula 1 avg	Formula 1 1/3	Formula 2 avg	Formula 2 1/3
ST173404LC	5.07	5.07	5.60	5.72
DJSA-232	9.33	9.33	12.00	12.59
DSCM-11000	7.00	7.00	12.00	13.10

Note that we used maximum seek distance and time rather than average in Formula 1. Had we used average seek time in that formula, we'd have been defining average seek time with itself.

7.3 a. The average seek distance for the Unix workload is 81.8 cylinders, and the average seek for the business workload is 14.2 cylinders. This assumes that the "midpoint" for the business workload is 104 for the range 97–112. (Rounding up instead only changes the average to 14.5 cylinders.)

b. If you ran the workloads on the disks in Figure 7.2, you'd find that the IBM Travelstar did significantly worse than the other disks because of its poor performance for short seeks. You'd also find that the Cheetah really isn't that much faster (for seeks, anyway) than the Microdrive because its minimum seek time isn't that much better than the Microdrive's, particularly for writes. For most workloads, it's short seeks that matter, and the Microdrive does quite well at those.

7.4 The average seek times for the workloads, using the assumptions from Problem 7.3a and the data from the problem, are summarized in the following tables.

For the Unix workload:

Distance	Percentage	Cheetah	Travelstar	Microdrive
188	3%	1.05	3.80	6.65
173	3%	1.03	3.74	6.45
158	2%	1.00	3.69	6.24
143	3%	0.98	3.63	6.02
128	2%	0.95	3.56	5.78
113	3%	0.92	3.50	5.52
98	3%	0.90	3.42	5.23
83	1%	0.87	3.35	4.92
68	3%	0.83	3.26	4.58
53	3%	0.80	3.17	4.18
38	4%	0.76	3.06	3.71
23	8%	0.72	2.93	3.12
8	23%	0.66	2.74	2.22
0	24%	0.60	2.50	1.00
300	15%	1.22	4.17	7.88
Average		0.81	3.13	3.78

For the business workload:

Distance	Percentage	Cheetah	Travelstar	Microdrive
201	0%	1.07	3.85	6.82
185	0%	1.04	3.79	6.61
169	0%	1.02	3.73	6.40
153	0%	0.99	3.67	6.17
137	3%	0.97	3.60	5.92
121	1%	0.94	3.53	5.66
105	1%	0.91	3.46	5.37
89	3%	0.88	3.38	5.05
73	1%	0.85	3.29	4.70
57	1%	0.81	3.20	4.29
41	1%	0.77	3.09	3.82
25	3%	0.73	2.95	3.21
9	11%	0.67	2.76	2.30
0	61%	0.60	2.50	1.00
Average		0.56	2.29	1.53

These numbers are significantly smaller than the 1/3-disk average seek times reported by the manufacturers.

7.5 No solution provided.

7.6 a. Number of heads = 15; number of platters = 8

b. Rotational latency = 8.33 ms

c. Head switch time = 1.4 ms

d. Cylinder switch time = 2.1 ms

e. Minimum time to media plus transfer time = 2.0 ms

7.7 a. The account file must handle 10,000 transactions per minute (tpm), so it must be 10000 × 0.07 = 700 GB. This requires either 18 small disks or 9 large disks—we must round up because the data set won't fit if we round the number of disks down. Each transaction requires 6 random disk I/Os (4 reads and 2 writes) to process it. Each transaction also requires 6 × 35,000 + 50,000 = 260,000 CPU instructions to process it.

The I/O bus of the computer can handle $\frac{1000 \times 10^6}{0.5 \times 10^3} = 2 \times 10^6$ disk requests per second. Unless we exceed 2 million disk requests per second, the I/O bus will not be the bottleneck. The 2500 MIPS CPU runs at 2.5×10^9 instructions per second, and can process $\frac{2.5 \times 10^9}{260{,}000} = 9615$ transactions per second.

The bottleneck in the system is clearly the disks. At 100 requests per second, each can support 100/6 = 16.67 transactions per second. This results in an overall speed of 16.67 × 9 = 150 transactions per second for a system with the large disks and 16.67 × 18 = 300 transactions per second for a system with the small disks.

b. The small and large disk systems both cost $20,000 plus the cost of disk for each system. For the small disks, cost is 20,000 + 400 × 18 = 27,200. For the large disks, it's 20,000 + 800 × 9 = 27,200, the same amount. However, the small system's cost per transaction per second (TPS) is 27,200 / 300 = 90.7, while the large system costs twice that per TPS, or 27,200 / 150 = 181.3.

c. In order to make the I/O bus a bottleneck, the system would have to process 2×10^6 requests per second, or $2 \times 10^6/6 = 333{,}333$ TPS. This would require 333,333 × 260,000 = 86.7×10^9 instructions per second, or 86.7 GIPS (1 GIPS = 1000 MIPS). You'll really need a lot of disks to hit this speed!

d. Cutting the number of database instructions per transaction to 30,000 and reducing the I/Os from 6 to 3 would reduce the instructions per transaction to 30,000 + 3 × 35,000 = 135,000. This would be an improvement of $\frac{260{,}000}{135{,}000} = 1.93$. The CPU would have to run 1.93 times as fast to gain the same performance benefit. Of course, the database group would make the system much cheaper because of the decrease in the number of disks. . . .

e. If the new CPU is 100% faster, it can handle $\frac{5000 \times 10^6}{260{,}000}$ = 19231 transactions per second. Since the old small disks can support 16.67 TPS, we'd need 19231/16.67 = 1154 small disks to make the CPU the bottleneck in our new system. Such a system would cost 20,000 + 400 × 1154 = $481,600, or nearly half a million dollars!

f. In our original design, we had 18 small disks. We want the disks to be able to run 19,231 TPS at 6 requests per transaction. That's a total of 19,231 × 6 = 115,386 requests per second. Since we have 18 disks, each disk must handle 115,386 / 18 = 6410 I/O requests per second. This is significantly faster than any disk today. In all fairness, though, the original system couldn't run anywhere near this fast.

7.8 From the seek-time formula, seek time = 5.13 ms. Reading 1 MB requires accessing 2048 512-byte sectors (if 1 MB = 2^{20} bytes; if 1 MB is taken as 10^6 bytes, then adjust accordingly). Disk rotation allow access to

$$\frac{528 \times 10{,}000}{60} = 88{,}000 \text{ sectors/sec}$$

Thus, 23.3 ms are needed to read 1 MB.

The read time for the single disk is

$$\text{Time}_{\text{single}} = 1 + 3 + 5.13 + 23.3 = 32.4 \text{ ms}$$

The read time for the four-disk array is

$$\text{Time}_{\text{array}} = 1 + 3 + 5.13 + \frac{23.3}{4} = 15.0 \text{ ms}$$

These times imply transfer rates as follows:

$$\text{I/O}_{\text{single}} = \frac{1}{32.4 \text{ ms}} = 30.9 \text{ I/O/sec or } 30.9 \text{ MB/sec}$$

$$\text{I/O}_{\text{array}} = \frac{1}{15.0 \text{ ms}} = 66.7 \text{ I/O/sec or } 66.7 \text{ MB/sec}$$

7.9 For this problem, we need to read 4 KB of largely sequential sectors where the average seek distance is 10 tracks. First, we note that, for the single disk case, the 8 sectors will be on different tracks 7/528 of the time (if sector 0 through 6 of the request is in the last few sectors of the track). For such cases, we'll pay a head switch or track-to-track seek of 0.5 ms. For the multiple-disk case, we only have this happen 1/528 of the time because there are only two sectors per request per disk.

The 10 track seek will take 0.598 ms, and a single-track seek requires 0.5 ms. The disk requires 60/10000 = 0.006 seconds, or 6 ms, for a single rotation. A single sector is read in 1/528 × 6 ms = 0.011 ms. A single request is made up of controller delay, rotational latency, seek time, and transfer time. Controller delay and rotational latency are the same for both single disk and array, 0.3 + 3 = 3.3 ms. Transfer time is 0.011 × 8 = 0.088 ms for the single disk and 0.011 × 2 = 0.022 ms for the array. Seek time is 0.598 ms, except in the case that a track-to-track seek must be done, which would add 0.5 ms.

For the single disk, average time excluding rotation and controller overhead is 0.088 + 0.598 + (7/528) × 0.5 = 0.693 ms. For the array, average time is 0.022 + 0.598 + (1/528) × 0.5 = 0.621 ms. The total time required is thus 3.993 ms for the single disk and 3.921 ms for the array. This translates to an I/O rate of 1000/3.993 = 250.4 requests per second for the disk and 1000/3.921 = 255.0 requests per second for the array. At these rates, and 4 KB per request, the disk will transfer 250.4 × 4 = 1001.6 KB/sec, and the array will transfer 255.0 × 4 = 1010 KB/sec.

7.10 For this problem, we need to transfer 2048 sectors for each 1 MB request.

This is 3 full tracks (1584 sectors) and an additional 2048 – 1584 = 464 sectors for the single disk; the 464 sectors might be spread across multiple tracks or be on a single track. The 3 full tracks would each require 0.5 + 6.0 = 6.5 ms to transfer. The remaining sectors are more complex. If the remaining sectors are on a single track, the head is either in the middle of the data or in the "empty" portion of the track. If in the middle of the data, a full revolution will be sufficient to read all of the data. If in the empty part of the track, the empty sectors behind the head may be skipped. Since we need to read 464 sectors, the expected time to read the track is

$$6 \times \left(\frac{464}{528} + \frac{1}{528} \sum_{i=0}^{53} \frac{464 + i}{528} \right) = 5.96 \text{ ms} .$$

If the remaining data is split between two tracks, the expected time to read and rotate is the same because it doesn't matter where we start—we'll still have the same rotation requirements, but split between the first and last track; there will, of course, be an extra 0.5 ms of seek time. The data will be split across tracks 463/528 of the time, for an added latency of 0.5 × 463/528 = 0.438 ms. Including controller time and the initial seek, total request time for the single disk is 0.3 + 4.9 + 6.5 × 3 + 5.96 + 0.438 = 31.1 ms. This corresponds to 1000/31.1 = 32.1 requests per second, or 32.1 MB per second.

For the disk array, things are a little different. Since the 2048 sectors are spread across four disks, we'll only have to read 512 sectors per disk. Using the same methods as for a single array, we can read this track in

$$6 \times \left(\frac{512}{528} + \frac{1}{528} \sum_{i=0}^{25} \frac{512 + i}{528} \right) = 5.997 \text{ ms} .$$

This is rounded up to 6 ms. We'll have to pay a single track switch most of the time (511/528), costing 0.5 ms. Total time including the controller is thus 0.3 + 4.9 + 6 + $\frac{511}{528}$ = 11.7 ms. This corresponds to 1000/11.7 = 85.6 requests per second, or 85.6 MB per second.

7.11 There are two effects of this policy. The first is that requests that miss will now take twice the transfer time, due to the need to fill the cache with the read-ahead sector. This adds 1/528 × 6 ms = 0.011 ms to the time to satisfy a single I/O request that misses in the cache, which happens 90% of the time.

10% of the time, we pay only the controller overhead (0.3 ms) and 50 ns per word transferred, or $0.050\mu s \times \frac{512}{4} = 0.0064$ ms, for a total time of 0.3064 ms.

We know from the solution to Problem 7.8 that average seek time is 5.13 ms, and that rotational latency is 3 ms on average, and we know that transfer time is 0.011 ms per sector. Our penalty is thus $0.9 \times 0.011 = 0.0099$ ms per request, and our savings is $0.1 \times (3 + 5.13 + 0.011 - 0.0064) = 0.8135$ ms. Overall savings is thus $0.8135 - 0.0099 = 0.8036$ ms per request. Read-ahead saves a lot of time because it's much faster to read out of cache than it is to pay the small penalty to transfer an additional sector.

7.12

7.13 No solution provided.

7.14 No solution provided.

7.15 No solution provided.

7.16 No solutions provided.

7.17 No solution provided.

7.18 We are asked to compute the average length of the queue and the average length of the system for the three examples that start on page 728.

Page of Example	Length of Queue	Length of Server	Length of System
728	3.2	0.8	4.0
729	0.3	0.4	0.7
730	0.15	0.4	0.55

Notes: The length of the system is the combined lengths of the queue and the server. For all of the examples length of queue is given by (arrival rate) * (time in queue). The length of server is given by server utilization. The last example had to use the new formulations given in the example since we are dealing with a M/M/m queue.

7.19 This problem asks us to redo the example on page 728 with $C^2 = 2$. The proper equation to use to derive the answer is found on page 726. The abbreviations correspond to the terms used in page 726.

```
Tq          = SU*(0.5*Ts*(1+C²)) + AR*Tq*Ts
            = SU*1.5*Ts + AR*Tq*Ts
            = SU*1.5*Ts + SU*Tq      (SU = AR*Ts)
Tq – SU*Tq  = SU*1.5*Ts
Tq*(1–SU)   = SU*1.5*Ts
Tq          = SU*1.5*Ts/(1–SU)
```

Since SU = 0.8 and Ts = 20 ms, Tq = 0.8*1.5*20 ms/(1-0.8) = 120 ms. The total time between queue and server is 140 ms. On average a request spends 86% of its time waiting in the queue compared to 80% with C2 = 1.

7.20 As calculated on page 745, the maximum IOPS per an Ultra3 SCSI controller is 2000 IOPS. Alternatively stated, each SCSI string can do 2000 IOPS. The following numbers are found on page 750.

At 40% string utilization:
80 GB disks, 3 strings = Min(50K, 50K, 31K, 2448, **2400**) = 2400 IOPS
40 GB disks, 4 strings = Min(50K, 50K, 31K, 4896, **3200**) = 3200 IOPS

At 50% string utilization:
80 GB disks, 3 strings = Min(50K, 50K, 31K, **2448**, 3000) = 2448 IOPS
40 GB disks, 4 strings = Min(50K, 50K, 31K, 4896, **4000**) = 4000 IOPS

At 60% string utilization:
80 GB disks, 3 strings = Min(50K, 50K, 31K, **2448**, 3600) = 2448 IOPS
40 GB disks, 4 strings = Min(50K, 50K, 31K, 4896, **4800**) = 4800 IOPS

At 70% string utilization:
80 GB disks, 3 strings = Min(50K, 50K, 31K, **2448**, 4200) = 2448 IOPS
40 GB disks, 4 strings = Min(50K, 50K, 31K, **4896**, 5600) = 3200 IOPS

At 50% string utilization, the disk performance becomes the bottleneck with the 80 GB disks with 3 strings configuration. At 70% string utilization, the disk performance now becomes the bottleneck with the 40 GB disks with 4 strings configuration.

7.21 a. Each tape of 60 GB read at 11 MB/sec takes 5455 seconds to read from start to finish. If 450 tape changes can be made per hour, then a tape change takes 8 seconds. Thus, we need 5455 + 8 = 5463 seconds per tape. With 16 readers, the PowderHorn can process 16 tapes in parallel from the silo. Thus, to read all 6000 tapes will take

$$\begin{aligned}\text{Time} &= \frac{\text{Number of tapes}}{\text{Number of tape readers}} \times \text{Time/tape} \\ &= \frac{6000}{16} \times 5463\,\text{sec} \\ &= 2,048,625\ \text{sec} = 569\ \text{hrs} = 23.7\ \text{days}\end{aligned}$$

b. At 1.52 hours/tape a 2000-hour rated helical scan head can scan 1316 tapes. With 16 readers we can scan all 6000 tapes (375 tapes per reader) about 3.5 times before it is time to replace all 16 heads, i.e., all 16 readers. In continuous use, this is about every 83 days.

7.22 No solution provided.

7.23 No solution provided.

7.24 Examples of competing technologies are

- for software distribution: CD-ROM, DVD, and internet connection
- for mass storage: DVD and hard disk
- for backup: CD-ROM, DVD, hard disk
- for disaster insurance: off-site storage via internet access, removeable media such as DVD, and readily portable hard disks such as the pocket-sized 2.5 inch disks.

CD-ROM and DVD offer longer lifetime than tape. Hard drive backup offers higher bandwidth for read and write than tape. Hard drive, CD-ROM, and DVD support random access rather than sequential access. Hard drive, CD-ROM, and DVD are more widely supported for small computer systems than is tape. CD-ROM and DVD read/write mechanisms are cheaper than tape read/write mechanisms.

Tape faces its strongest competition in software distribution, as seen by its complete replacement by other technologies for this purpose. Perhaps the area with the second strongest challenge for tape is small to medium scale backup, where the simplicity of a hard disk solution and competitive cost is very attractive.

The major advantage of tape is for truly massive data archives and for its energy efficiency (kilowatt hours per terabyte per month of storage).

7.25 This example is not removed from the field of computing but it is impressive. The first transistor was made in late 1947. By 2001, estimates put semiconductor industry production at 9×10^9 transistors per second, or more than one transistor for every person every second! To reach this production rate over the 54 years since 1947 has required annual manufacturing growth of 110%.

Another example might be the personal MP3 players available today that hold thousands of songs. A typical DVD movie is a data object equal in size to perhaps 2000 MP3 songs. With a 10,000 fold improvement, a personal MP3 player could hold 10,000 2-hour movies. There are science fiction stories in which characters continuously video record their lives. In the not distant future the storage cost and volume for such a record will no longer be prohibitive.

7.26 No solution provided.

7.27 No solution provided.

Chapter 8 Solutions

8.1 Delivered bandwidth = Megabits per second including overhead, so the formula is

$$\frac{\text{Packet size} \times 8}{\text{Overhead} + (\text{Ethernet header/trailer size} + \text{Packet size}) \times 8/(0.9 \times 1000\text{M bits/sec})}$$

$$= \frac{\text{Packet size} \times 8}{100\ \mu\text{s} + (56 + \text{Packet size}) \times 8/(900\ \text{bits}/\mu\text{s})}$$

Thus a stream of 32-byte payloads delivers a user data bandwidth of 2.5M bits per second, and a stream of 1500-byte payloads delivers a user data bandwidth of 105.4M bits per second. The graph is in Figure S.34.

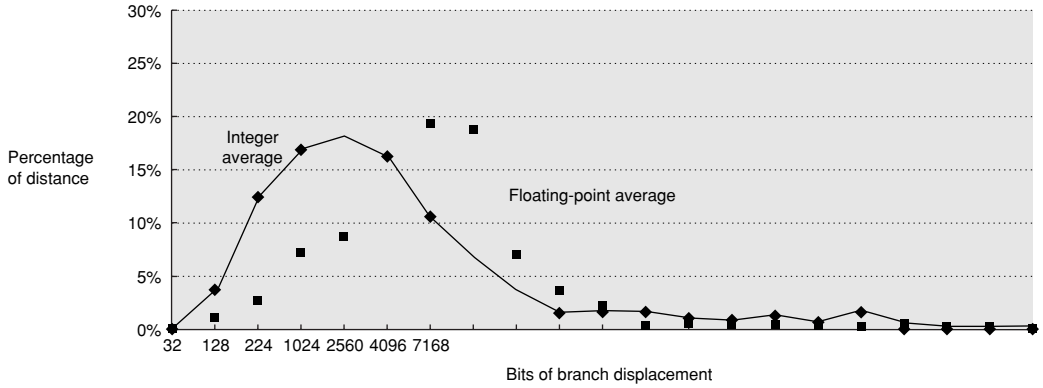

Figure S.34 M bits per second of user data as payload size varies.

8.2 After solving the equation from Exercise 8.1 the answer is 14,132 bytes, or almost a factor of 10 greater than the size of the maximum Ethernet payload.

8.3 The respective latencies are

$$\text{Worst case Ethernet latency} = \frac{(56 + 1500) \times 8}{1000} = 12.4 \text{ microseconds}$$

$$\text{Worst case ATM latency} = \frac{(5 + 48) \times 8}{622} = 0.68 \text{ microseconds}$$

8.4 The impact is almost an order of magnitude:

$$\text{Ethernet latency} = \frac{(56 + 14132.5) \times 8}{1000} = 113.5 \text{ microseconds}$$

8.5 a. All message sizes of 224 bytes or more have a total time for ATM that is less than for Ethernet; therefore, ATM delivers higher bandwidth.

b. The 155M bit/sec ATM takes 3 seconds to transmit 4820 messages of size 224, while the 10M bit/sec Ethernet takes 4 seconds. The effective bandwidths are 2.9 and 2.2M bits/sec, respectively. For the largest message size the effective bandwidths are 42.4 and 8.3M bits/sec, respectively. Figure S.36 shows the effective bandwidths for all the message sizes.

8.6 Because Figure 8.50 on page 868 only reports total time in coarse units (seconds) and per-message overhead will be much smaller, we need to pick the message sizes that are used in great numbers so that round-off error does not significantly affect the result. The five most-used message sizes are 32, 96, 128, 160, and 8192 bytes. For ATM the overhead in microseconds for each size is 690, 690, 690, 691,

and 689. Thus, 690 microseconds seems like a good estimate for ATM overhead in this figure. For Ethernet the overheads are 505, 504, 504, 505, and 504. Thus, 504 microseconds is a good estimate for Ethernet overhead in this figure.

8.7 For the traffic of Figure 8.50 on page 868, 95% of the messages are 160 bytes or less, and 10 Mbit/s Ethernet transmits them faster than ATM. Yet 71% of the total data sent is in messages 224 bytes or larger, and 155 Mbit/s ATM transmits that message size faster than Ethernet.

8.8 $$\text{Effective bandwidth} = \frac{\text{Message size} \times \text{Number of messages} \times 8}{\text{Total time} \times 10^6}$$

It is convenient to use a spreadsheet to compute the value of effective bandwidth for each row of data in Figure 8.50 on page 868. The results show that the smallest message size for Ethernet that exceeds half of the peak bandwidth is 1024 bytes (5.3 Mbit/s). The largest message size, 8192 bytes, delivers only 46.9 Mbit/s. For 155 Mbit/s ATM to achieve $155 \div 2$ or 77.5 Mbit/s, we need to extrapolate the message size. The answer to Exercise 8.6 suggests an overhead of 690 microseconds per message for ATM. Assuming that we could use 90% of the peak bandwidth, how large a message would spend half its time in overhead and half in data transmission? The answer is 12032 bytes. (Alas, the peak bandwidth observed in the study summarized in Figure 8.50 for ATM data transfer was only 77.0 Mbit/s, so in reality there is no message size that delivers half of the peak bandwidth.)

8.9 The two Internet alternatives take 99 hours or 45 hours, respectively, for the whole 1000 GB to arrive. At 15 hours door to door, the overnight delivery service is faster.

8.10 Overnight delivery bandwidth is $\frac{1000 \times 1000 \times 8 \text{ Mbits}}{15 \times 60 \times 60 \text{ seconds}}$ or 148 Mbit/s.

8.11 The slowest link would have to be 300 Mbit/s so that 50% of that bandwidth would beat overnight delivery. This requires the slowest link be at least OC-12 (622 Mbit/s) or 1000 Mbit/s Ethernet.

8.12 Calculating time of flight:

$$\text{Time of flight} = \frac{2.5 \text{ km}}{(2/3) \times 300,000 \text{ km/sec}} = 12.5 \times 10^{-6} \text{ sec}$$

The factor of $2/3$ is an estimate of the ratio of propagation speed of light in free space in a vacuum to that in a conductor or waveguide (see the Example on page 798). The number of bytes in transit on the 10 Mbit/s network is

$$\begin{aligned}\text{Time of flight} &= \text{Delivered bandwidth} \times \text{Time of flight} \\ &= \frac{0.9 \times 10 \text{ Mbit/sec}}{8} \times 0.125 \times 10^{-4} \text{sec} \\ &= 1.125 \text{ MB/sec} \times 0.125 \times 10^{-4} \text{sec} \\ &= 14 \text{ bytes}\end{aligned}$$

8.13 Calculating time of flight:

$$\text{Time of flight} = \frac{0.1 \text{ km}}{2/3 \times 300,000 \text{ km/sec}} = 0.5 \times 10^{-6} \text{ sec}$$

The LAN delivers 90% of its peak bandwidth, so the number of bytes in transit on a 10,000 Mbit/sec network is

$$\begin{aligned}\text{Bytes in transit} &= \text{Delivered bandwidth} \times \text{Time of flight} \\ &= \frac{0.9 \times 10000 \text{ Mbit/sec}}{8} \times 0.5 \times 10^{-6} \text{sec} \\ &= 1125 \text{ MB/sec} \times 0.5 \times 10^{-6} \text{sec} \\ &= 562.5 \text{ bytes}\end{aligned}$$

8.14 The length of a full Ethernet packet is 1500 + 56 bytes. The time to transfer 1556 bytes over a 1G bit/sec Ethernet link (assuming 90% of peak bandwidth) is

$$\frac{1556 \text{ bytes}}{0.9 \times 1000\text{M bits/sec}} = \frac{1556 \text{ bytes}}{112.5 \text{ MB/sec}} = 13.8\mu\text{s}$$

Then the time for store and forward is

$$\begin{aligned}&(\text{Switches} \times \text{Switch delay}) + ((\text{Switches} + 1) \times \text{Transfer time}) \\ &= (7 \times 1.0) + (8 \times 13.8) = 117.6\mu\text{s}\end{aligned}$$

while wormhole routing is

$$(\text{Switches} \times \text{Switch delay}) + \text{Transfer time} = (7 \times 1.0) + 13.8 = 20.8\ \mu\text{s}$$

For this example, wormhole routing improves latency by more than a factor of 5.

8.15 The length of this Ethernet packet is 32+56 bytes. The time to transfer 1556 bytes over a 1 Gb/s Ethernet link is (assuming 90% of peak bandwidth) is

$$\frac{88 \text{ bytes}}{0.9 \times 1000 \text{ Mbit/sec}} = \frac{88 \text{ bytes}}{112.5 \text{ MB/sec}} = 0.78\mu\text{s}$$

The time to store and forward is

$$\begin{aligned}&(\text{Switches} \times \text{Switch delay}) + ((\text{Switches} + 1) \times \text{Transfer time}) \\ &= (7 \times 1.0) + (8 \times 0.78) = 13\mu\text{s}\end{aligned}$$

while wormhole routing is

$$(\text{Switches} \times \text{Switch delay}) + \text{Transfer time} = (7 \times 1.0) + 0.78 = 7.8\mu\text{s}$$

For this example, wormhole routing improves latency by 1.7 times.

8.16 The answers to the exercises above tell us that the time of flight is 13.8 microseconds for 1500 bytes and 0.78 microseconds for 32 bytes.

Then the time for store and forward for 1500 bytes is

$$\begin{aligned}&(\text{Switches} \times \text{Switch delay}) + ((\text{Switches} + 1) \times \text{Transfer time}) \\ &= (3 \times 1.0) + (4 \times 13.8) = 58.2\mu\text{s}\end{aligned}$$

while wormhole routing is

$$(\text{Switches} \times \text{Switch delay}) + \text{Transfer time} = (3 \times 1.0) + 13.8 = 16.8\mu\text{s}$$

The time for store and forward for 32 bytes is

$$(\text{Switches} \times \text{Switch delay}) + ((\text{Switches} + 1) \times \text{Transfer time}) = (3 \times 1.0) + (4 \times 0.78) = 6.1\ \mu\text{s}$$

while wormhole routing is

$$(\text{Switches} \times \text{Switch delay}) + \text{Transfer time} = (3 \times 1.0) + 0.78 = 3.8\ \mu\text{s}$$

Wormhole routing improves latency by 1.6 times for 32-byte packets and by 3.5 time for 1500-byte packets.

8.17 To make a *d*-cube, create two *(d-1)*-cubes and connect the corresponding vertices of the two lower order cubes. Doing so leads to the drawing shown in Figure S.35.

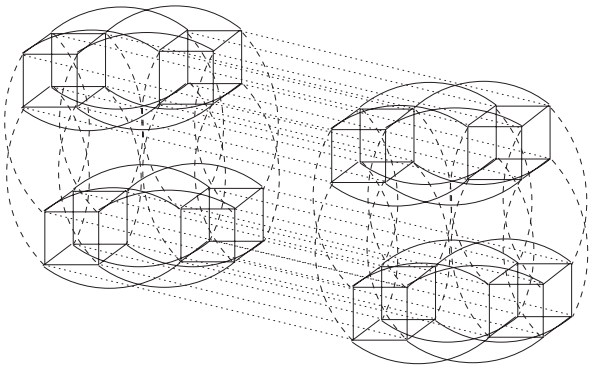

Figure S.35 Drawing of a 6-dimensional Boolean hypercube.

8.18 a. The low order bit position of the label number written in binary is complemented.

b. The bits of the successor stage switch input label are the left circular shift of the bits of the switch output label of the preceding stage.

c. Finding the intended answer is frustrated by an error in the drawing of Figure 8.13(b). The output links from the rightmost column of switches should connect switch outputs to path arrowheads in exactly the same pattern shown for the preceding two pairs of switch columns, rather than the straight horizontal connections shown. With this correction made the answer is as follows.

A message enters the Omega network (technically an Inverse Omega network is shown in the figure, but this makes no difference to the answer) at the port with the same label as the processor that sent it. That message exits the Omega network at a port with the same label as the processor to receive it. The answers to parts (a) and (b) show that each switch provides an opportu-

nity to complement the low order label bit and that each stage of links shifts a new label bit into a position where it can be complemented or not. With n stages, one n-bit label can be transformed into any n-bit label, including itself (identity function). The sending processor can compute a routing tag as the bitwise exclusive-or of the sending and receiving processor labels. A routing tag bit of 0 indicates that a switch should be set to straight, a tag bit of 1 means set to exchange. Switches in the first column of the network set themselves based on the low-order routing tag bit. Switches in the next column use the next to low-order routing bit, and so on.

8.19 The depiction of the crossbar switch in Figure 8.13(a) shows each processor node with two unidirectional ports. Data is transmitted unidirectionally from a processor node output port rightward into the switch array until it reaches a switch that is closed (all switches are normally open) at which point the data moves to the vertically-drawn buses and travels to the input port of the selected processor.

If the processor nodes are viewed as having a single bi-directional port then the crossbar operation and drawing in Figure 8.13(a) change as follows. Data leaves a processor rightward on the single, bidirectional link and enters the switch array. In the array the necessary two switches are closed to connect the sending node to the receiving node via a bi-directional vertical segment and a second, horizontal bus. Data exits the array leftward on the horizontal bus to the destination processor. The vertical buses no longer wrap counterclockwise around the nodes to a second port, but link only the columns of switch points in the array. With this definition of bidirectional crossbar switch ports, we can answer the exercise as follows.

Use 16 ports from each of four 18-port switches (bi-directional ports) to connect to the 64 nodes. Then use the two remaining ports on each switch to join the switches in a ring. This design allows a message to travel entirely within the switches between the transmitting and receiving processors.

Another network topology is possible using four unidirectional-port crossbar switches if messages are allowed to be forwarded by processor(s) between the sender and receiver. Connect 16 nodes to the first crossbar inputs and a different 16 nodes to the crossbar outputs. Then connect the second set of 16 nodes to a second crossbar switch and connect the outputs of that switch to a third different set of 16 nodes. Continue in this fashion for a third and fourth switch, but connect the output ports of the fourth switch back to the first set of 16 nodes.

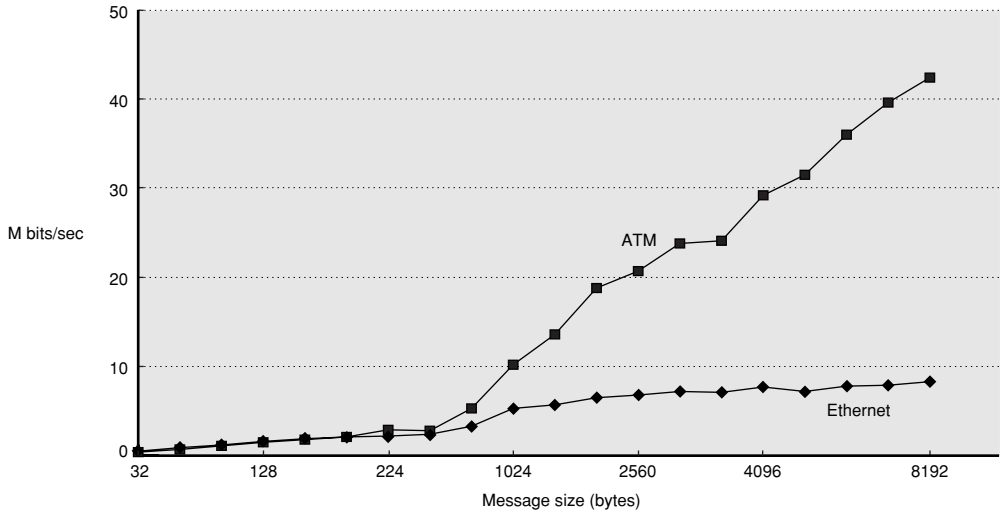

Figure S.36 Effective delivered bandwidth for ATM and Ethernet.

8.20 One solution is to use 5 bi-directional port switches, each connected to the 64 nodes (13 nodes per switch except for 12 nodes to the fifth switch). This leaves 5 or 6 ports unused on each switch. Use 4 of these ports on each switch to connect to the other switches to join all the nodes. The latency between the nodes on the same switch is 1 and the latency between all other nodes is 2. The potential for congestion is high because there is only one wire connecting each set of 13 nodes on one switch to the 13 nodes on one other switch.

8.21 One solution is to use 32 bi-directional port switches to build a fat-tree, see Figure 8.14. The bottom layer uses 8 switches to connect to the 64 nodes, leaving 4 ports each to go to the left and right halves of the second level of the fat-tree. This second layer consists of four pairs of switches. These are in turn connected to the third layer that consists of two quads of switches. The top layer of the fat-tree has 8 switches.

If unidirectional crossbar switches are used (see the discussion about unidirectional and bi-directional crossbar switch design in the solution to Exercise 8.19), then there is another solution that uses only 8 switches. Connect the 64 nodes to the inputs of a column of 4 switches, 16 per switch. This will look like the connection between the processors and the leftmost column of switches in the Omega network shown in Figure 8.13(b). Then, use 16 of the outputs from each switch to connect to a second column of 4 switches. Wire 4 output ports from each switch in the first column to 4 input ports of a single switch in the second column, and repeat this for each second-column switch. Finally, connect 16 of the output ports of each second-column of switch to the input ports of the 64 processors. With a uniform traffic pattern each wire will be equally loaded.

This network design is also a potentially good solution for Exercise 8.20. While the minimum delay is 2 for this network instead of 1, the potential for congestion is one-fourth as much, for uniform traffic, and the maximum latency is still just 2.

8.22

Network	Best case	Worst case	From P0 to P6	From P1 to P7
Crossbar	1	1	1	1
Omega	3	3	3	3
Fat Tree	1	6	6	6

Figure S.37

8.23 Forwarding is the process of deciding whether to discard an Ethernet packet or to pass it to the other LAN. Thus, the time to forward is $\text{Time}_{\text{server}}$ for the gateway.

a. $$\begin{aligned}\text{Utilization} &= \text{Arrival rate} \times \text{Time}_{\text{server}} \\ &= 200/\text{second} \times 0.002 \text{ second} \\ &= 0.4\end{aligned}$$

b. From Little's Law

$$\begin{aligned}\text{Mean number of packets} &= \text{Arrival rate} \times \text{Mean response time} \\ &= \text{Arrival rate} \times (\text{Time}_{\text{queue}} + \text{Time}_{\text{server}}) \\ &= \text{Arrival rate} \times \left(\frac{\text{Time}_{\text{server}}}{1 - \text{Utilization}}\right) \\ &= 200/\text{second} \times \left(\frac{0.002 \text{ second}}{1 - 0.4}\right) \\ &= 0.67\end{aligned}$$

c. $$\begin{aligned}\text{Time}_{\text{gateway}} &= \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} \\ &= \text{Time}_{\text{server}} \times \left(\frac{\text{Utilization}}{1 - \text{Utilization}}\right) + \text{Time}_{\text{server}} \\ &= \text{Time}_{\text{server}}/(1 - \text{Utilization}) \\ &= 0.002 \text{ seconds}/(1 - 0.4) \\ &= 3.3 \text{ ms}\end{aligned}$$

d. Varying the arrival rate is the same as varying the utilization of the gateway. Figure S.38 shows the effect on response time.

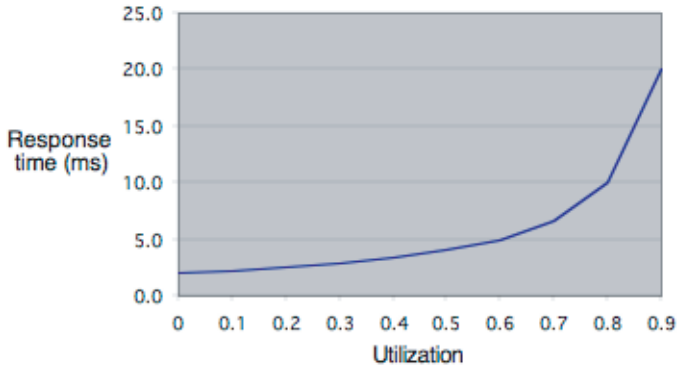

Figure S.38 Response time as a function of gateway utilization.

e. For a server with no queue, overflow occurs whenever there is an arrival when the server is busy, that is when there is one task in the system. Thus, a system that can hold k tasks will overflow the queue whenever there are greater than $k + 1$ tasks in the system.

Thus, for a FIFO that holds 10 tasks,

$$\begin{aligned}\text{Probability of overflow} &= (\text{Utilization})^{10+1+1} \\ &= 0.4^{12} \\ &= 1.67 \times 10^{-5}\end{aligned}$$

f. To achieve less than one packet per million lost due to FIFO overflow

$$\begin{aligned}1.0 \times 10^{-6} &> \text{Probability of overflow} \\ &= (\text{Utilization})^{k+2} \\ &= (0.4)^{k+2}\end{aligned}$$

So $k + 2 \geq 16$ and the FIFO must hold at least 14 packets.

8.24 For the unbalanced case, sending machine A will initiate a new send as soon as the current send completes. If sends complete faster than receives, then receiving machine B will face a continuous stream of messages to receive with no time to send its own messages. With bandwidth matching a new send by A is delayed enough to that B will complete processing of the received message some time before the next message from A can arrive. This interval between messages allows B to begin sending its own messages to A, and from that point A and B will each alternate between sending and receiving until running out of messages to send.

Let s, n, and r denote send overhead, network latency plus time of flight, and receive overhead, respectively. Figure S.39 shows the unbalanced case assuming A begins sending enough in advance of B that the first action for B is to receive.

The figure shows the matched case assuming A and B start simultaneously. Changing the relative starting times of A and B affects only the timing of the initial message; after that the message transfer will proceed as shown.

For i messages and ignoring minor differences due to different relative starting times, the total time to exchange is modeled by the following equations.

$$\text{Time}_{\text{unbalanced}} = 2(s + n + ir)$$

$$\text{Time}_{\text{matched}} = i(s + r) + \frac{i}{2}(2n) = i(s + n + r)$$

For the given values of s, n, r, and i

$$\text{Time}_{\text{unbalanced}} = 2(200 + 15 + 100 \times 300) = 60,430 \text{ microseconds}$$

$$\text{Time}_{\text{matched}} = 100(200 + 15 + 300) = 51,500 \text{ microseconds}$$

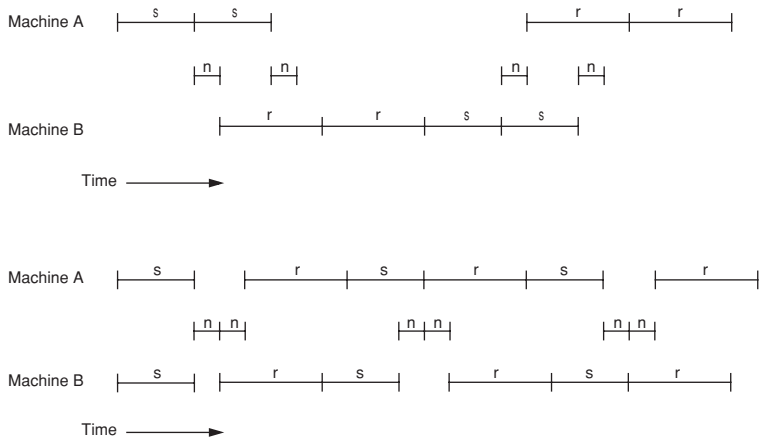

Figure S.39 Timing diagrams for unbalanced and bandwidth-matched exchange. Send overhead, network latency plus time of flight, and receive overhead and labeled *s, n,* and *r,* respectively. Time increases to the right. For space reasons the unbalanced case shows the exchange of only two messages. The bandwidth-matched case tracks the exchange of three messages to show the self-synchronizing nature of this scheme: a machine that is behind quickly catches up.

8.25 There are many ways to modify UDP code as proposed in the exercise and to then conduct the experiment, so no specific answer can be given. The two questions asked have specific answers

How should you change the send rate when two nodes send to the same destination? Answer: Delay each send enough so that the receiver can complete processing of the two received messages some time before either node sends again.

What if one sender sends to two destinations? Answer: Consider the sends to one destination separately from sends to the other. Then for each set of sends make sure that the send intervals satisfy the bandwidth matching constraint for a single destination.

8.26 No solution provided.

8.27 No solution provided.

8.28 No solution provided.

8.29 No solution provided.

8.30 No solution provided.

8.31 a. This exercise says to use the same assumptions for components as in the second example in Section 7.11 except for a 1G bit Ethernet switch. The first step, then, in solving this exercise is to relate the example of Section 7.11 to the components of the three cluster computers so that each can be assigned an MTTF. The set of components in the example is CPU/memory, disk, SCSI controller, power supply, fan, SCSI cable, and enclosure for disks external to the CPU/memory housing. The external disk enclosure contains one fan and one power supply.

A number of additional assumptions are necessary or helpful in mapping all the cluster components to those in the example and in taking care of all the details. These assumptions are

- Each processor and 1 GB of memory has the MTTF of a CPU/memory unit.
- CPU/memory unit MTTF is unaffected by clock rate or L2 size.
- Disk MTTF is independent of disk capacity.
- Each server has one Emulex cLAN-1000 host adapter (1G bit) card with MTTF equal to that of a SCSI controller.
- Ethernet switch MTTF is independent of the number of switch ports.
- The two disks that each computer can hold have a SCSI controller.
- Each power supply has the same MTTF, regardless of the rating in watts.
- There is one fan in each power supply.
- The EXP300 storage enclosure contains one power supply.
- The Emulex cLAN-1000 10-meter cable and the Ultra2 SCSI 4-meter cable each have the same MTTF as a SCSI cable.
- A standard 19-inch rack (44 VME rack units) has an infinite MTTF.

Using the above assumptions we can generate the table of component and quantity information in Figure S.40. Collecting the data of Figure S.40 together, we compute failure rates for each cluster:

$$\text{Failure rate}_{1\text{-way}} = \frac{32+64+2+37}{10^6} + \frac{32+32}{2\times 10^5} + \frac{32+32}{5\times 10^6} = \frac{583}{10^6}$$

$$\text{Failure rate}_{2\text{-way}} = \frac{32+32+1+17}{10^6} + \frac{16+16}{2\times 10^5} + \frac{16+16}{5\times 10^6} = \frac{306}{10^6}$$

$$\text{Failure rate}_{8\text{-way}} = \frac{32+32+4+4+1+5}{10^6} + \frac{12+12+4+4}{2\times 10^5} + \frac{8+4}{5\times 10^6} = \frac{262}{10^6}$$

Component	MTTF (hours)	How many? 1-way	2-way	8-way
CPU/memory	1,000,000	32	32	32
Disk	1,000,000	64	32	32
CPU power supply	200,000	32	16	12
CPU/memory power supply fan	200,000	32	16	12
SCSI controller	500,000	32	16	8
EXP300 storage enclosure	1,000,000	0	0	4
EXP300 power supply	200,000	0	0	4
EXP300 fan	200,000	0	0	4
SCSI cable	1,000,000	0	0	4
cLAN host adapter	500,000	32	16	4
1G bit Ethernet switch	1,000,000	2	1	1
cLAN-1000 10-meter cable (between servers and switch and between switches)	1,000,000	37	17	5
Standard 19-inch rack	N/A	1	1	2
Total component count		264	147	120

Figure S.40 Quantities of components for each cluster configuration.

The MTTF for the clusters is just the inverse of the failure rates.

$$\text{MTTF}_{1\text{-way}} = \frac{1}{\text{Failure rate}_{1\text{-way}}} = \frac{1{,}000{,}000 \text{ hours}}{583} = 1715 \text{ hours}$$

$$\text{MTTF}_{2\text{-way}} = \frac{1}{\text{Failure rate}_{2\text{-way}}} = \frac{1{,}000{,}000 \text{ hours}}{306} = 3268 \text{ hours}$$

$$\text{MTTF}_{8\text{-way}} = \frac{1}{\text{Failure rate}_{8\text{-way}}} = \frac{1{,}000{,}000 \text{ hours}}{262} = 3817 \text{ hours}$$

To put the MTTF times in some perspective, a year lasts about 8766 hours. The 8-way cluster design achieves its better MTTF by having fewer total components, but in particular by having fewer of the low-MTTF power supply and fan components.

b. To name the single point of failure there must first be a definition of failure. For these clusters the goal is to have a system with 32 processors, 32 GB of memory, and more than 2 TB of disk. The organizations shown in Figure 8.34 also show an Internet connection. The definition of failure then is falling below the required numbers and amounts of processors, memory, and storage plus losing Internet access.

For the uniprocessor-based cluster in Figure 8.34 the single points of failure are

- Any processor (cluster falls below required number of processors and amount of memory)
- Any CPU power supply (processor cannot operate)
- Any CPU power supply fan (power supply will go offline)
- Any cLAN-1000 host adapter (isolates a processor, its memory, and two disks)
- An Ethernet switch failure (isolates half of the processors, memory, and disk and may cut Internet access)
- An Ethernet cable failure other than that of the four cables connecting the two switches (cuts off one processor module or cuts Internet access)

Failure of one disk or one SCSI controller will not cause the uniprocessor-based cluster to fail to meet its goals.

For the two-way SMP cluster the single points of failure are

- Any processor (cluster falls below required number of processors and amount of memory)
- Any CPU power supply (two processors cannot operate)
- Any CPU power supply fan (power supply will go offline)
- Any cLAN-1000 host adapter (isolates two processors, their memory, and two disks)
- An Ethernet switch failure (isolates all the processors, memory, and disk and cuts Internet access)
- Any Ethernet cable failure (cuts off two processors or cuts Internet access)

Failure of one disk or one SCSI controller will not cause the two-way SMP cluster to fail to meet its goals.

For the eight-way SMP cluster the single points of failure are

- Any processor (cluster falls below required number of processors and amount of memory)
- Any CPU power supply (assumes that the other two power supplies will be insufficient or not configured to carry the total power load of the server)
- Any CPU power supply fan (power supply will go offline)
- Any cLAN-1000 host adapter (isolates eight processors, their memory, and eight disks)
- An Ethernet switch failure (isolates all the processors, memory, and disk and cuts Internet access)
- Any Ethernet cable failure (cuts off eight processors or cuts Internet access)
- Any SCSI controller for the disks in an EXP300 storage enclosure (cluster loses six disks and falls below the required amount of storage)

c. Generally, the strategy to improve MTTF is some combination of reduced component count, substitution of higher-MTTF components for lower-MTTF components, and design with redundancy so that there are fewer or no single points of failure.

For each of the cluster designs loss of one processor is a failure, so adding a spare CPU/memory component would go a long way to improving cluster MTTF. All of the cluster designs have unused ports on the Ethernet switch, so adding another server requires only the additional server and another Ethernet cable. Interestingly, the uniprocessor-based cluster, which has the lowest MTTF because of its high component count, is the cheapest of the clusters to incrementally grow the number of processors. Assuming the cluster software could automatically handle single-processor failure, adding one spare server to the uniprocessor design should improve cluster MTTF significantly. Adding a spare processor is more expensive with the two-way and eight-way servers, but also offers cluster MTTF improvement.

Replicating other aspects of the clusters may be more difficult. For components such as the power supplies, building servers with two power supplies operating in parallel and each supplying half the server power while running at just 40% of rated capacity of the power supply and with just one fan per power supply might have sufficient cost-performance to be a good choice. Such a configuration can fail over to one power supply for all the server's power needs in the event of a single fan or supply failure without overstressing and quickly failing the supply that picks up the full load. Replicating the Ethernet switches would double the number of needed host adapters and cables and would require a dual Internet connection. Perhaps a better solution would be to mount the switch in the coolest part of the standard 19-inch rack (likely at the bottom) and hope that cooler operating temperatures will lead to above-average MTTF for that component.

8.32 a. The MTTF for the IBM items in the following chart have been estimated from surrounding information. They are not the actual MTTF figures from IBM. The rest of the information for the failure rate computations comes from Figure B.17. Note that the extra storage space for the 8-way configuration is removed since the SAN device space accounts for all of the hard disk space. The 8-way configuration still has the highest MTTF.

Hardware Item	Qty	MTTF (hrs)
IBM FC-AL RAID Server	1	1,000,000
IBM 73.4 GB Disks	32	1,000,000
IBM FC-AL Host Adapter	(32, 16, 4)	500,000

Failure Rates:

FR1-way = (32 + 2 + 37 + 1 + 32)/1M + (32 + 32)/200K + (32 + 32)/500K
FR2-way = (32 + 1 + 17 + 1 + 32)/1M + (16 + 16)/200K + (16 + 16)/500K
FR8-way = (32 + 1 + 5 + 1 +32)/1M + (12 + 12)/200K + (4 + 4)/500K

MTTF1-way = 1/FR1-way = 1812 hrs
MTTF2-way = 1/FR2-way = 3257 hrs
MTTF8-way = 1/FR8-way = 4831 hrs

b. The SAN configuration might make disk repair and management easier; however it becomes a single point of failure. Since all of the disks are behind the RAID controller, once the controller fails it renders all the disks useless. Each one of the FC-AL host adapters adds a point of failure. The failures mentioned in problem 8.31 are applicable as well. However switching in a spare cluster node becomes easier. Since a failed cluster node in the SAN example does not contain data (as opposed to the scenario in problem 8.31), no time is lost from manually transferring data from the disks of a failed node into the spare.

c. c) Since the RAID controller in the SAN device is a single point of failure, one might duplicate the SAN device. The secondary SAN device would simply mirror the primary SAN. Another approach is to have redundant controllers in the RAID server. We could also introduce redundant FC-AL host adapters within each cluster node.

8.33 a. The following table lists the most important parts to consider in the MTTF calculation of the TPC-C cluster. We assume that the missing parts do not contribute to the MTTF.

Hardware Item	Qty	MTTF (hrs)
CPU Processor	32	1,000,000
RAID Controller	28	1,000,000
Ultra160 SCSI Controller	28x4	500,000
9.1 GB Disk	560	1,000,000
18.2 GB Disk	140	1,000,000
1Gbit Ethernet Switch	1	1,000,000

Failure Rate = (32 + 28 + 560 + 140 +1)/1M + 28*4/500K
= .000985 per hour
MTTF = 1/Failure Rate
= 1015 hrs

b. In addition to the failures pointed out in the solution of problem 8.31, the sheer number of disks and SCSI controllers significantly decreases the MTTF. Since there are no backups, each one of these components may become a single point of failure.

c. One possible MTTF improvement is to use larger disks. This would cut the needed number of disks to reach 8 TB at the cost of sacrificing IOPS performance (unless individual disk IOPS increase). It would also reduce the number of controllers.

8.34 a. Using the information from Section 7.11 pages 744–745, the IOPS of the smaller hard drive is 1/(2 + 5 + 1.1 ms) = 123 IOPS. The IOPS of the larger hard drive is 1/(2 + 5 + 0.7 ms) = 129 IOPS. The latter term in both of these calculations is the time to transfer an average block size (32KB) from the disk using the information given to us in the problem. There are 64 small disks in the 1-way cluster. Since the capacity of a singular SCSI controller (2000 IOPS) can handle the bandwidth from two disks, the bottleneck becomes the bandwidth of the disk. This configuration yields a total IOPS of 7872. At a cost of $180,000, each IOPS costs $23 dollars. There are 32 large disks in the 2-way cluster. This yields 4128 IOPS. The 2-way configuration costs $161,000 making it a mere $39 dollars per IOPS. The 8-way configuration packs 8 large disks per cluster node. This is split between two controllers; controller bandwidth is still not the limitation. This configuration matches the performance of the 2-way IOPS figure. However at a cost of $253,000, its cost performance increases to $61 dollars per IOPS.

b. With the disk utilization restriction imposed in Section 7.11, the revised small disk IOPS drops to 98 and large disk IOPS drops 103. The string utilization also drops to 800 IOPS. The SCSI controller in a 1-way cluster easily supports two small disks (196 IOPS < 800 IOPS). With 64 disks at 98 IOPS, this yields a cost-performance figure of $29 dollars per IOPS. The SCSI controller in a 2-way cluster supports two of the larger disks (206 < 800 IOPS) giving an aggregate IOPS of 3296. The cost per IOPS increases to $49 dollars. If we had to place 8 disks on one controller in the 8-way example, the controller would have become the bottleneck. However since two disks are place on an internal controller and 6 disks are place on the external controller the disks once again become the bottleneck. The aggregate disk IOPS remains at 3296, but the cost per IOPS increases to $77 dollars.

c. The disks limit the overall IOPS rating of the cluster. The only configuration that places any stress on the SCSI controller is the 8-way configuration. This is even with the 60% reduction on SCSI controller utilization.

d. One solution to improve the overall IOPS rating is to have more smaller disks or more larger disks if one wants to increase capacity as well. Alternatively one might seek to replace the disks with faster ones. Adding more disks requires more space as well as energy. Faster disks would utilize the space more effectively in terms of IOPS.

e. The switch must accommodate each cluster node transmitting at 1 Gb/s. With 32 cluster nodes plus an Internet connection, the aggregate switch bandwidth between two switches must be at least 33 Gb/s. This assumes that the Ethernet adapter can transmit and received simultaneously at full speed. With a 16

node cluster, the switch bandwidth drops to 17 Gb/s. Finally with an 8 node cluster, the switch bandwidth decreases to 9 Gb/s.

8.35 a. For this problem, we assume that the IBM FC-AL high-availability RAID storage server and the FC-AL host adapters are not the bottlenecks. The example uses 32 IBM 73.4 GB 10K RPM FC-AL disks. Using the IOPS calculation from problem 8.34, the SAN device yields a total of 4128 IOPS. At a cost of $598,000 dollars for a 1-way SAN configuration, the cost per IOPS is $145. For a $497,000 dollar 2-way SAN configuration, the figure becomes $120. The $594,000 dollar 8-way SAN configuration yields a cost per IOPS of $144 dollars.

b. For this problem, we assume that the IBM FC-AL high-availability RAID storage server and the FC-AL host adapters are not the bottlenecks. The example uses 32 IBM 73.4 GB 10K RPM FC-AL disks. Using the IOPS calculation from problem 8.34, the SAN device yields a total of 4128 IOPS. However if we restrict the disk utilization to 80% (as suggested in Section 7.11), the IOPS drops 3296. At a cost of $598,000 dollars for a 1-way SAN configuration, the cost per IOPS is $181. For a $497,000 dollar 2-way SAN configuration, the figure becomes $151. The $594,000 dollar 8-way SAN configuration yields a cost per IOPS of $180 dollars.

c. In this example the disks are the bottleneck. However the host adapters might be a bottleneck if they cannot deliver the IOPS bandwidth coming from the disks. At full disk utilization, this would place a 129 IOPS demand on a FC-AL host adapter of a 1-way configuration, 258 IOPS on a 2-way configuration, and a 1032 IOPS demand on an 8-way configuration.

d. One might split the SAN device in two. Doubling the number of disks would double the IOPS rating if the other components in the system can handle the added strain. Instead of adding another device, one might make use of the space already present within each cluster node to add more hard drives.

e. The switch must accommodate each cluster node transmitting at 1 Gb/s. With 32 cluster nodes plus an Internet connection, the aggregate switch bandwidth between two switches must be at least 33 Gb/s. This assumes that the Ethernet adapter can transmit and received simultaneously at full speed. With a 16 node cluster, the switch bandwidth drops to 17 Gb/s. Finally with an 8 node cluster, the switch bandwidth decreases to 9 Gb/s. There are no changes between the FC-AL example and the first example as in problem 8.34. However since the SAN connects on a separate adapter, this might reduce the overall load across the switch.

8.36 a. Using the information found on page 745, each SCSI controller (string) can support up to 15 disks per controller. There are four strings per RAID controller. Each cluster node contains 7 RAID controllers. Each cluster node supports up to 420 disks at full resource utilization. From Figure 8.40, each cluster connects to 13 disk enclosures (each with 14 disks) yielding a total of 182 disks. Since the actual number of disks falls below the supported number

of disks by the SCSI controllers, the IOPS per cluster node is limited by the total IOPS of the disks. Since each disk provides 128 IOPS, the TPC-C cluster yields a total of 93184 IOPS (182*128*4 IOPS). At $2.2 million dollars, the cost of each IOPS is $23 dollars.

b. Using the resource limitations imposed by Section 7.11, 40% string utilization limit yields a maximum of 800 IOPS per string. A 80% disk utilization limits yields a maximum of 102 IOPS per disk. Assuming a uniform distribution of resources, the total disk IOPS for the entire cluster becomes 80%*93184 IOPS = 74547 IOPS. The total number of SCSI controllers in the TPC-C cluster is (4 controllers per RAID) X (7 RAID controllers per node) X (4 nodes) or 112 controllers. This achieves an IOPS of 89600 under the 40% restriction. The disks remain the bottleneck. The cost per IOPS increases to $29 dollars.

c. Clearly the disks are the performance bottleneck. The disk IOPS falls short of the controller IOPS. The Ethernet switch may become a bottleneck if there is cross-cluster communication due to the large volume of data stored at each cluster node.

d. Adding more disks until we hit the maximum IOPS rating of the individual controllers is one possible way of increasing IOPS. We could opt to use smaller disks to offer the same capacity or larger disks to offer more capacity. Another possibility is seeking faster disks with lower latency seek times and/or higher transfer throughput. Once the controllers become the bottleneck, we could add more controllers. However, we will exhaust available bus IOPS. Alternative techniques include adding more cluster nodes and or SAN storage if processing power would not become the bottleneck.

e. The switch must accommodate each cluster node transmitting at 1 Gb/s. There are four cluster nodes and an Internet connection. Therefore the switch bandwidth must be at least 5 Gb/s. This assumes that the Ethernet adapter can transmit and received simultaneously at full speed.

8.37 No solution provided.

8.38 No solution provided.

8.39 No solution provided.

8.40

Appendix A Solutions

A.1 a. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD	R1, 0(R2)	F	D	X	M	W													
DADDI	R1, R1, #1		F	s	s	D	X	M	W										
SD	0(R2), R1					F	s	s	D	X	M	W							
DADDI	R2, R2, #4								F	D	X	M	W						
DSUB	R4, R3, R2									F	s	s	D	X	M	W			
BNEZ	R4, Loop												F	s	s	D	X	M	W
LD	R1, 0(R2)																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is 98*16 + 18 = 1584. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

b. Now we are allowed normal bypassing and forwarding circuitry. Branch outcomes and targets are known now at the end of decode.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
LD	R1, 0(R2)	F	D	X	M	W									
DADDI	R1, R1, #1		F	s	D	X	M	W							
SD	0(R2), R1				F	D	X	M	W						
DADDI	R2, R2, #4					F	D	X	M	W					
DSUB	R4, R3, R2						F	D	X	M	W				
BNEZ	R4, Loop							F	s	D	X	M	W		
LD	R1, 0(R2)										F	D	X	M	W

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is 9*98 + 12 = 894. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

c.

		1	2	3	4	5	6	7	8	9	10	11
LD	R1, 0(R2)	F	D	X	M	W						
DADDI	R2, R2, #4		F	D	X	M	W					
DSUB	R4, R3, R2			F	D	X	M	W				
DADDI	R1, R1, #1				F	D	X	M	W			
BNEZ	R4, Loop					F	D	X	M	W		
SD	-4 (R2), R1						F	D	X	M	W	
LD	R1, 0(R2)							F	D	X	M	W

The branch now has a single-cycle delayed branch slot. It is assumed that this slot will always be executed regardless of whether or not the branch was taken. The instructions have been rearranged in order to remove all RAW hazard stalls. The store was safety moved to the delay slot with the minor exception that the R2 had changed. This was compensated by decrementing four from the register. Now, there are only 6 cycles between loops and the last cycle requires 4 extra cycles to complete: 98*6 + 10 = 598.

A.2 The pipeline of Sections A.4 and A.5 resolves branches in ID and has multiple execution function units. More than one instruction may be in execution at the same time, but the exercise statement says write-back contention is possible and is handled by processing one instruction at a time in that stage.

Figure A.30 lists the latencies for the functional units; however, it is important to note that these data are for functional unit results forwarded to the EX stage. In particular, despite a latency of 1 for data memory access, it is still the case that for a cache hit the MEM stage completes memory access in one clock cycle.

Finally, examining the code reveals that the loop iterates 99 times. With this and analysis of iteration timing on the pipeline, we can determine loop execution time.

a. Figure S.41 shows the timing of instructions from the loop for the first version of pipeline hardware.

There are several stall cycles shown in the timing diagram:

- Cycles 5–6: `MUL.D` stalls in ID to wait for `F0` and `F4` to be written back by the `L.D` instructions.
- Cycles 8–15: `ADD.D` stalls in ID to wait for `MUL.D` to write back `F0`.
- Cycle 19: `DSUBU` stalls in ID to wait for `DADDUI` to write back `R2`.
- Cycles 21–22: `BNEZ` stalls in ID to wait for `DSUBU` to write back `R5`. Because the register file can read and write in the same cycle, the `BNEZ` can read the `DSUBU` result and resolve in the same cycle in which `DSUBU` writes that result.

		Clock cycle																							
Instruction		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**...**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
L.D	F0,0(R2)	F	D	E	M	W																			
L.D	F4,0(R3)		F	D	E	M	W																		
MUL.D	F0,F0,F4			F	D	s	s	E	E	...	E	M	W												
ADD.D	F2,F0,F2				F	s	s	D	s	...	s	s	s	E	E	E	E	M	W						
DADDUI	R2,R2,#8							F	s	...	s	s	s	D	E	M	W								
DADDUI	R3,R3,#8													F	D	E	M	W							
DSUBU	R5,R4,R2														F	D	s	E	M	W					
BNEZ	R5,Loop															F	s	D	s	r					
L.D	F0,0(R2)																	F	s	s	F	D	E	M	W

Figure S.41 Pipeline timing diagram for the pipeline without forwarding, branches that flush the pipeline, memory references that hit in cache, and FP latencies from Figure A.30. The abbreviations F, D, E, M, and W denote the fetch, decode, execute, memory access, and write-back stages, respectively. Pipeline stalls are indicated by s, branch resolution by r. One complete loop iteration plus the first instruction of the subsequent iteration is shown to make clear how the branch is handled. Because branch instructions complete (resolve) in the decode stage, use of the following stages by the BNEZ instruction is not depicted.

- Cycle 20: While not labeled a stall, because initially it does not appear to be a stall, the fetch made in this cycle will be discarded because this pipeline design handles the uncertainty of where to fetch after a branch by flushing the stages with instructions fetched after the branch. The pipeline begins processing after the branch with the correct fetch in cycle 23.

There are no structural hazard stalls due to write-back contention because processing instructions as soon as otherwise possible happens to use WB at most once in any clock cycle.

Figure S.41 shows two instructions simultaneously in execution in clock cycles 17 and 18, but because different functional units are handling each instruction there is no structural hazard.

The first iteration ends with cycle 22, and the next iteration starts with cycle 23. Thus, each of the 99 loop iterations will take 22 cycles, so the total loop execution time is $99 \times 22 = 2178$ clock cycles.

b. Figure S.42 shows the timing of instructions from the loop for the second version of pipeline hardware.

There are several stall cycles shown in the timing diagram:

- Cycle 5: MUL.D stalls at ID waiting for L.D to forward F4 to EX from MEM. F0 reaches the register file by the first half of cycle 5 and thus is read by ID during this stall cycle.
- Cycles 7–12: ADD.D stalls at ID waiting for MUL.D to produce and forward

the new value for F0.

- Cycle 16: DSUBU is stalled at ID to avoid contention with ADD.D for the WB stage. Note the complexity of pipeline state analysis that the ID stage must perform to ensure correct pipeline operation.
- Cycle 18: BNEZ stalls in ID to wait for DSUBU to produce and forward the new value for R5. While forwarding may deliver the needed value earlier in the clock cycle than can reading from the register file, and so in principle the branch could resolve earlier in the cycle, the next PC value cannot be used until the IF stage is ready, which will be with cycle 19.
- Cycle 17: Initially this cycle does not appear to be a stall because branches are predicted not taken and this fetch is from the fall-through location. However, for all but the last loop iteration this branch is mispredicted. Thus, the fetch in cycle 17 must be redone at the branch target, as shown in cycle 19.

		Clock cycle																			
Instruction		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**...**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
L.D	F0,0(R2)	F	D	E	M	W															
L.D	F4,0(R3)		F	D	E	M	W														
MUL.D	F0,F0,F4			F	D	s	E	E	...	E	M	W									
ADD.D	F2,F0,F2				F	s	D	s	...	s	E	E	E	E	M	W					
DADDUI	R2,R2,#8						F	s	...	s	D	E	M	W							
DADDUI	R3,R3,#8										F	D	E	M	W						
DSUBU	R5,R4,R2											F	D	s	E	M	W				
BNEZ	R5,Loop												F	s	D	r					
L.D	F0,0(R2)														F	s	F	D	E	M	W

Figure S.42 Pipeline timing diagram for the pipeline with forwarding, branches handled by predicted-not-taken, memory references that hit in cache, and FP latencies from Figure A.30. The notation used is the same as in Figure S.41.

Again, there are instances of two instructions in the execute stage simultaneously, but using different functional units.

The first iteration ends in cycle 19 when DSUBU writes back R5. The second iteration begins with the fetch of L.D F0, 0(R2) in cycle 19. Thus, all iterations, except the last, take 18 cycles. The last iteration completes in a total of 19 cycles. However, if there were code following the instructions of the loop, they would start after only 16 cycles of the last iteration because the branch is predicted correctly for the last iteration.

The total loop execution time is $98 \times 18 + 19 = 1783$ clock cycles.

A.3 This exercise asks, "How much faster would the machine be . . . ," which should make you immediately think speedup. In this case, we are interested in how the

presence or absence of control hazards changes the pipeline speedup. Recall one of the expressions for the speedup from pipelining presented on page A-13

$$\text{Pipeline speedup} = \frac{1}{1 + \text{Pipeline stalls}} \times \text{Pipeline depth} \tag{S.4}$$

where the only contributions to Pipeline stalls arise from control hazards because the exercise is only focused on such hazards. To solve this exercise, we will compute the speedup due to pipelining both with and without control hazards and then compare these two numbers.

For the "ideal" case where there are no control hazards, and thus stalls, Equation B.4 yields

$$\text{Pipeline speedup}_{\text{ideal}} = \frac{1}{1 + 0} \ (4) = 4 \tag{S.5}$$

where, from the exercise statement the pipeline depth is 4 and the number of stalls is 0 as there are no control hazards.

For the "real" case where there are control hazards, the pipeline depth is still 4, but the number of stalls is no longer 0 as it was in Equation B.5. To determine the value of Pipeline stalls, which includes the effects of control hazards, we need three pieces of information. First, we must establish the "types" of control flow instructions we can encounter in a program. From the exercise statement, there are three types of control flow instructions: taken conditional branches, not-taken conditional branches, and jumps and calls. Second, we must evaluate the number of stall cycles caused by each type of control flow instruction. And third, we must find the frequency at which each type of control flow instruction occurs in code. Such values are given in the exercise statement.

To determine the second piece of information, the number of stall cycles created by each of the three types of control flow instructions, we examine how the pipeline behaves under the appropriate conditions. For the purposes of discussion, we will assume the four stages of the pipeline are Instruction Fetch, Instruction Decode, Execute, and Write Back (abbreviated IF, ID, EX, and WB, respectively). A specific structure is not necessary to solve the exercise; this structure was chosen simply to ease the following discussion.

First, let us consider how the pipeline handles a jump or call. Figure S.43 illustrates the behavior of the pipeline during the execution of a jump or call. Because the first pipe stage can always be done independently of whether the control flow instruction goes or not, in cycle 2 the pipeline fetches the instruction following the jump or call (note that this is all we can do—IF must update the PC, and the next sequential address is the only address known at this point; however, this behavior will prove to be beneficial for conditional branches as we will see shortly). By the end of cycle 2, the jump or call resolves (recall that the exercise specifies that calls and jumps resolve at the end of the second stage), and the pipeline realizes that the fetch it issued in cycle 2 was to the wrong address

(remember, the fetch in cycle 2 retrieves the instruction immediately following the control flow instruction rather than the target instruction), so the pipeline reissues the fetch of instruction $i + 1$ in cycle 3. This causes a one-cycle stall in the pipeline since the fetches of instructions after $i + 1$ occur one cycle later than they ideally could have.

Figure S.44 illustrates how the pipeline stalls for two cycles when it encounters a taken conditional branch. As was the case for unconditional branches, the fetch issued in cycle 2 fetches the instruction after the branch rather than the instruction at the target of the branch. Therefore, when the branch finally resolves in cycle 3 (recall that the exercise specifies that conditional branches resolve at the end of the third stage), the pipeline realizes it must reissue the fetch for instruction $i + 1$ in cycle 4, which creates the two-cycle penalty.

Figure S.45 illustrates how the pipeline stalls for a single cycle when it encounters a not-taken conditional branch. For not-taken conditional branches, the fetch of instruction $i + 1$ issued in cycle 2 actually obtains the correct instruction. This occurs because the pipeline fetches the next sequential instruction from the program by default—which happens to be the instruction that follows a not-taken branch. Once the conditional branch resolves in cycle 3, the pipeline determines it does not need to reissue the fetch of instruction $i + 1$ and therefore can resume executing the instruction it fetched in cycle 2. Instruction $i + 1$ cannot leave the IF stage until *after* the branch resolves because the exercise specifies the pipeline is only capable of using the IF stage while a branch is being resolved.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Jump or call	IF	ID	EX	WB		
$i + 1$		IF	IF	ID	EX	. . .
$i + 2$			*stall*	IF	ID	. . .
$i + 3$				*stall*	IF	. . .

Figure S.43 Effects of a jump or call Instruction on the pipeline.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Taken branch	IF	ID	EX	WB		
$i + 1$		IF	*stall*	IF	ID	. . .
$i + 2$			*stall*	*stall*	IF	. . .
$i + 3$				*stall*	*stall*	. . .

Figure S.44 Effects of a taken conditional branch on the pipeline.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Not-taken branch	IF	ID	EX	WB		
$i + 1$		IF	*stall*	ID	EX	. . .
$i + 2$			*stall*	IF	ID	. . .
$i + 3$				*stall*	IF	. . .

Figure S.45 Effects of a not-taken conditional branch on the pipeline.

Combining all of our information on control flow instruction type, stall cycles, and frequency leads us to Figure S.46. Note that this figure accounts for the taken/not-taken nature of conditional branches. With this information we can compute the stall cycles caused by control flow instructions:

$$\text{Pipeline stalls}_{\text{real}} = (1 \times 1\%) + (2 \times 9\%) + (1 \times 6\%) = 0.24$$

where each term is the product of a frequency and a penalty. We can now plug the appropriate value for Pipeline $\text{stalls}_{\text{real}}$ into Equation B.4 to arrive at the pipeline speedup in the "real" case:

$$\text{Pipeline speedup}_{\text{real}} = \frac{1}{1 + 0.24} (4.0) = 3.23 \tag{S.6}$$

Finding the speedup of the ideal over the real pipelining speedups from Equations B.5 and B.6 leads us to the final answer:

$$\text{Pipeline speedup}_{\text{without control hazards}} = \frac{4}{3.23} = 1.24$$

Control flow type	**Frequency (per instruction)**	**Stalls (cycles)**
Jumps and calls	1%	1
Conditional (taken)	$15\% \times 60\% = 9\%$	2
Conditional (not taken)	$15\% \times 40\% = 6\%$	1

Figure S.46 A summary of the behavior of control flow instructions.

Thus, the presence of control hazards in the pipeline loses approximately 24% of the speedup you achieve without such hazards.

A.4 If a branch outcome is to be determined earlier, then the branch must be able to read its operand equally early. Branch direction is controlled by a register value that may either be loaded or computed. If the branch register value comparison is performed in the EX stage, then forwarding can deliver a computed value produced by the immediately preceding instruction if that instruction needs only one cycle in EX. There is no data hazard stall for this case. Forwarding can deliver a loaded value without a data hazard stall if the load can perform memory access in a single cycle and if at least one instruction separates it from the branch.

If now the branch compare is done in the ID stage, the two forwarding cases just discussed will each result in one data hazard stall cycle because the branch will need its operand one cycle before it exists in the pipeline. Often, instructions can be scheduled so that there are more instructions between the one producing the value and the dependent branch. With enough separation there is no data hazard stall.

So, resolving branches early reduces the control hazard stalls in a pipeline. However, without a sufficient combination of forwarding and scheduling, the savings in control hazard stalls will be offset by an increase in data hazard stalls.

A.5

	Pipeline Stages											
	← Cycle Number →											
1	1	2	3	4	5	6	7	8	9	10	11	
I	1	IF	RF	ALU1	MEM	ALU2	WB					
N	2		IF	RF	ALU1	MEM	ALU2	WB				
S	3			IF	RF	ALU1	MEM	ALU2	WB			
T	4				IF	RF	ALU1	MEM	ALU2	WB		
#	5					IF	RF	ALU1	MEM	ALU2	WB	
	6						IF	RF	ALU1	MEM	ALU2	WB

a. The simultaneous occurrence of IF, ALU1, and ALU2 maximizes the number of needed adders. This situation happens when instructions 2 and 4 are reg-mem ALU ADD instructions and instruction 6 can be any instruction that changes the PC via an increment.

b. The WB and RF stages both access the register file. A reg-reg ALU operation will require two values from the register file in the RF stage. If this operation occurs as instruction 6 in the pipeline diagram above and an operation that writes to the register occurs as instruction 2, then we well need a 2 read port, 1 write port register file to sustain this combination of instructions. The IF and MEM stages both access memory. When these stages occur together, we will need up to 2 memory read ports and 1 write port.

c. In the following table, we only consider data forwarding between operations that use the ALU.

The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
ALU2(1)	ALU2(2)	ALUop (including Load result forwarding) to ALUop
ALU2(1)	MEM(3)	ALUop to ALUop via MEM
ALU2(1)	ALU1(4)	ALUop to a memory or branch ALUop

d. In the following table, we consider data forwarding between operations that do not use the ALU.

The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
MEM(1)	MEM(2)	Load ALUop to Store ALUop
ALU2(1)	MEM(3)	Load ALUop to Store ALUop

e. In the following table, we show the number of stalls it takes to resolve dependencies between various instructions. The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Stalls	Remarks
ALU2(1)	ALU1(2)	1	ALUop producing value for branch or memory op
ALU2(2)	ALU1(3)	2	ALUop producing value for branch or memory op
MEM(1)	ALU1(3)	1	Load op producing value for branch or memory op

f. Assume Instruction 1 is a branch. It will take two cycles to determine if we are on the wrong path. If we are on the wrong path, Instructions 2 and 3 will have to be flushed.

A.6 a. The traditional five stage pipeline of IF, ID, EX, MEM, WB can be modified to IF, ID, MEM, EX, WB if we only want to support register indirect addressing.

b. The following illustrates the forwarding paths in the modified pipeline.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
MEM	MEM	Load to Store (data value or index)
EX	MEM	ALU OP producing the register value for a Load or Store
EX	EX	Dependant ALU operation

c. The following illustrates the new data hazards in the modified pipeline.

Examples:		Remarks
ADD	R1, R2, R3	
LD	R5, 0(R1)	Instruction has to stall for R1
ADD	R1, R2, R3	
SW	0(R7), R1	Instruction has to stall for R1

d. There are several ways that this modified pipeline can have a different instruction count for a given program over the original pipeline. The modified pipeline enables the compiler to merge loads and ALU operations, thereby reducing instruction count. However, these instructions cannot be merged if the memory operation requires an offset. If a memory operation requires an offset and we are not able to merge it with another instruction, the modified pipeline requires an additional instruction to compute the offset value and store it in a register.

Examples:				
Original Pipeline		Modified Pipeline		Remarks
`LD R9, 16(R1)`		`ADDUI R2, R1, #16`		Requires an additional instruction
		`LD R9, 0(R2)`		
`LD R6, 8(R1)`		`ADDUI R2, R1, #8`		No gain nor loss
`ADD R7, R6, R8`		`ADD R7, R8, 0(R2)`		(See Part (e))
`LD R6, 0(R1)`		`ADD R7, R8, 0(R1)`		We are able to merge two instructions
`ADD R7, R6, R8`				

e. In the original pipeline, a load producing a value for the next instruction causes a stall. Since this is not the cause for the modified pipeline, the CPI decreases. The stall can also be "removed" by merging these two instructions in the modified pipeline. However, if an offset value has to be calculated the

execute stage cannot deliver the value to the MEM stage fast enough, thereby causing a one cycle stall.

Example:		Remarks:	
		Original Pipeline	**Modify Pipeline**
`LD`	`R6, 0(R1)`		
`ADD`	`R7, R6, R8`	1 Cycle Stall	No Stalls
`ADDUI`	`R6, R5, #8`		
`LD`	`R7, 0(R6)`	No Stall	1 Cycle Stall

A.7 a. Delayed Branch—Penalties

	From Before	FromFall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	0	0	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. If the compiler takes an instruction from fall-through, it must make sure that there are no dependencies between that instruction and instructions from the target of the branch. If it takes an instruction from target, the compiler must make sure that there are no dependencies between that instruction and the instruction from fall-through. Please note that these rules only apply to true dependencies. If an instruction from the correct path overwrites the value produced from the instruction in the branch delay slot, then essentially we are correcting the value stored in that particular register. However, we would still have to pay the penalty.

b. Cancel-If-Not-Taken—Penalties

	From Before	From Fall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	1	1	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. However we have to be concerned that the instruction in the delay slot might be cancelled. Therefore, the compiler has to account for this possibility. Since there would be a penalty regardless of the branch being taken or not, the compiler shouldn't take an instruction from fall-through. Since the branch delay slot instruction is canceled if not taken, the compiler does not have to worry about dependencies when taking an instruction from target to fill the slot.

c. If the compiler seeks to fill the branch delay slot from before the branch, then the delayed branch scheme is the best. This scheme will ensure the execution of the branch delay slot instruction regardless of the branch's outcome.

If the compiler seeks to fill the branch delay slot from fall-through, then the branch delayed scheme is the best. As pointed out in part b, there is a penalty using this scheme regardless of the branch's outcome.

If the compiler seeks to fill the branch delay slot from target, cancel-if-not-taken is better since the compiler would not have to worry about ensuring correctness when the branch is not taken.

A.8

Source Latch	Destination Latch	Comments
EX/DF	RF/EX	ALU to ALU
DF/DS	EX/DF	ALU to ALU
DF/DS	RF/EX	ALU to ALU
DS/TC	EX/DF	Load to Store
DS/TC	RF/EX	Load to ALU
TC/WB	EX/DF	Load/ALU to Store
TC/WB	RF/EX	Load/ALU to ALU

A.9 Detecting MIPS R4000 integer hazards requires examining the pipeline for all stages where integer results are produced and for all stages where integer results are used. Integer results can be written either to the register file or to memory. A result written to memory cannot be used by the pipeline until a load instruction reads it, so we need only consider integer instructions that read from and write to the register file to detect pipeline integer hazards.

Integer ALU and load instructions are the MIPS R4000 instructions that produce integer results written to the register file. ALU results are produced at the end of EX; load results are produced at the end of DS. (Although we must wait for the tag check in TC before knowing if the cache access was a hit, we ignore this issue here, as it is outside the functional scope of the integer hazard detection circuitry.) ALU, load, store, and branch instructions all have integer operands. ALU instructions use their operands in EX. Load instructions use their operand in EX. Store instructions use their integer operands in both EX and DF. MIPS R4000 evaluates branch conditions in EX, thus branch instruction integer register operands are used in EX. With this information about result creation and use locations, we can now determine the integer hazards in the MIPS R4000.

An ALU instruction may produce a result that is used by another ALU instruction or by a load, store, or branch instruction. All of these result uses are in either the EX or DF stages. ALU instructions take only one cycle to execute in the MIPS R4000 pipeline, so forwarding from the output of the EX stage or later in the pipeline can prevent a stall for these uses even if the using instruction immediately follows the producing ALU instruction.

The other integer instruction that produces a result for the register file is the load. Because R4000 uses a deeper pipeline with three memory access stages, the

opportunity for forwarding to prevent data dependences from causing data hazards is less than for the classic five-stage RISC pipeline. The fact that the R4000 pipeline can forward the value of a data cache hit from the DS stage reduces, but does not eliminate, data hazard stalls. For loads there can be hazards.

The answer given in the table assumes that the pipeline interlock stops instruction progress at the RF stage (see Figure A.37).

Note that if a value was forwarded from the DS stage and then there is a cache miss, then when the pipeline is "backed up a cycle" (see Figure A.38 caption) to recover from the data cache miss, the effect of the forwarding of incorrect data must also be undone.

Location of load instruction	Opcode field of IS/RF	Matching operand fields	Interlock for how long? (cycles)
RF/EX	Reg-reg ALU, ALU immediate, load, store, or branch	Any register operand	2
EX/DF	Same as above	Same as above	1
DF/DS	N/A	N/A	Forwarding handles this case

Figure S.47 The logic to detect integer interlocks during the RF stage of the MIPS R4000. Of the integer instructions, only a load may require a pipeline stall. In all cases the load destination register is compared to all of the register operands of the instruction in the RF stage. Remember that the IS/RF register holds the state of the instruction in RF. The duration of the interlock is the time necessary for the load instruction to reach the output of the DS stage, at which point the value can be forwarded.

A.10

Source Latch	Destination Latch	Comments
M7/MEM	ID/M1	FP Multiply to FP Multiply
M7/MEM	ID/A1	FP Multiply to FP Add
A4/MEM	ID/A1	FP Add to FP Add
A4/MEM	ID/M1	FP Add to FP Multiply
EX/MEM	ID/EX	Integer ALU to Integer ALU
MEM/WB	ID/M1	Any Arithmetic/Load Op to FP Multiply
MEM/WB	ID/A1	Any Arithmetic/Load Op to FP Add
MEM/WB	ID/EX	Any Arithmetic/Load Op to Integer ALU

A.11

Incoming FP Instruction	Older FP Instruction	Matching Destination Operand Fields
Load	Multiply	IF/ID == ID/M1
Load	Multiply	IF/ID == ID/M2
Load	Multiply	IF/ID == ID/M3
Load	Multiply	IF/ID == ID/M4
Load	Multiply	IF/ID == ID/M5
Load	Multiply	IF/ID == ID/M6
Load	Add	IF/ID == ID/A1
Load	Add	IF/ID == ID/A2
Load	Add	IF/ID == ID/A3
Add	Multiply	IF/ID == ID/M1
Add	Multiply	IF/ID == ID/M2
Add	Multiply	IF/ID == ID/M3

A.12 a.

Instruction		Starts EX	Stalls
L.D	F2, 0(R1)	3	0
MUL.D	F4, F2, F0	5	1
L.D	F6, 0(R2)	6	0
ADD.D	F6, F4, F6	12	5
S.D	0(R2), F6	16	3
DADDUI	R1, R1, #8	17	0
DADDUI	R2, R2, #8	18	0
DSGTUI	R3, R1, done	19	0
BEQZ	R3, foo	21	0
CPI = (9 + 9)/9 = 2			

b.

Instruction		I	RO	EC	WR	Remarks
L.D	F2, 0(R1)	X	X	X	X	
MUL.D	F4, F2, F0	X	X	X	X	
L.D	F6, 0(R2)	X	X	X	X	
ADD.D	F6, F4, F6	X	X	X	X	
S.D	0(R2), F6	X	X	X		RAW Hazard
DADDUI	R1, R1, #8	X	X	X	X	Structural Hazard
DADDUI	R2, R2, #8	X	X	X	X	
DSGTUI	R3, R1, done	X	X	X	X	
BEQZ	R3, foo	X				

We are asked to show the scoreboard when DSGTUI enters WR. The only active instruction in the scoreboard at this time is DSGTUI, excluding BEQZ. In the Functional unit status section, Integer is Busy—all other functional units are idle. Fi, Fk, and Fk equal R3, R1, and Immediate respectively. Qj and Qk are blank. Rj and Rk are both No.

c.

Instruction		I	RO	EC	WR
L.D	F2, 0(R1)	X	X	X	X
MUL.D	F4, F2, F0	X	X	X	X
L.D	F6, 0(R2)	X	X	X	X
ADD.D	F6, F4, F6	X	X	X	X
S.D	0(R2), F6	X	X	X	
DADDUI	R1, R1, #8	X	X	X	X
DADDUI	R2, R2, #8	X	X	X	X
DSGTUI	R3, R1, done	X	X	X	X
BEQZ	R3, foo	X			

The scoreboard is empty when the second loop iteration begins. The loop takes 9 clock cycles to complete.

Notes:	
I	Issue
RO	Read Operands
EC	Execution Completes
WR	Write Results

A.13