University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 1998 D.A. Patterson
Solution #2

Solutions 2
CS252 Graduate Computer Architecture

B.4

Let's say the memory access starts in bank 0. The first element will take 8 cycles. The next element will go to bank (20 mod 8 = 4); it will not conflict, so it will take just one more cycle. The third element will go to bank (40 mod 8) = 0, so this will have to wait for the first element to finish before it can begin.

Since memory access will bounce between banks 0 and 4, and each bank can be accessed only once in 9 cycles, it will take $(64/2) \times 8 + 1$ or 257 cycles.

If we accessed with a stride of 1, the entire access would only take $8 + 63 = 71$ cycles to complete. Thus, for the stride 20 case, we are only achieving $71/257 = 28\%$ of the available memory bandwidth.

5.5

a) First of all, I assumed that in the abscence of memory stalls, the processor executed with a CPI of 1.0.

The equation for CPI can be calculated as follows:

$$\mathrm{CPI}_{\text{total}} = \mathrm{CPI}_{\text{execution}} + \mathrm{CPI}_{\text{mem stalls}}$$

In the equation above, $\mathrm{CPI}_{\text{execution}}$ gives the CPI in the absence of memory stalls for all instructions. This can be calculated as follows:

$$\mathrm{CPI}_{\text{total}} = \text{freq}_{\text{non-mem}} \times \text{cycles}_{\text{non-mem}} + \text{freq}_{\text{loads}} \times \text{cycles}_{\text{loads}} + \text{freq}_{\text{stores}} \times \text{cycles}_{\text{stores}}$$

The frequency for each instruction is given on page 105. The cycles for non-mem instructions are assumed to be one. In the absence of memory stalls, the problem specifies that loads take one clock cycle, and stores take two clock cycles.

To calculate $\mathrm{CPI}_{\text{mem stalls}}$, we have to look at two separate components:

$$\text{CPI}_{\text{mem stalls}} = \text{CPI}_{\text{I fetch stalls}} + \text{CPI}_{\text{D stalls}}$$

$\text{CPI}_{\text{I fetch stalls}}$ can be calculated by multiplying the I cache miss rate times the I cache miss penalty. The I cache miss rate is given as 0.5%; the I cache miss penalty is simply 50 clock cycles. (Since the I cache won't be written to, it won't have dirty blocks, and therefore won't ever have to write back values to memory on a cache miss.)

$\text{CPI}_{\text{D stalls}}$ can be calculated in terms of loads and stores:

$$\text{CPI}_{\text{D stalls}} = \text{freq}_{\text{loads}} \times \text{stall cycles}_{\text{loads}} + \text{freq}_{\text{stores}} \times \text{stall cycles}_{\text{stores}}$$
$$\text{stall cycles} = \text{miss frequency} \times \text{miss penalty}$$

The frequency of each instruction is given on page 105; the miss frequency is given as 1%. The miss penalty depends on the type of instruction and cache organization. For the write-through cache, a store miss has a penalty of 50 cycles (we have to read in the memory block, modify it, and write it back out again; the write buffer hides the latency of the write to memory), and a load miss has a penalty of 50 cycles. For the write-back cache, the store miss penalty and load miss penalty depend on if the block is dirty:

$$\text{store miss penalty} = \text{\% of blocks that are dirty} \times \text{memory write time} + \text{memory read time}$$
$$\text{load miss penalty} = \text{\% of blocks that are dirty} \times \text{memory write time} + \text{memory read time}$$

The problem specifies that 50% of the blocks are dirty, memory write time is 50 cycles, and memory read time is 50 cycles.

OK, we're ready to actually calculate for the CPI for the different programs!

Write-through Cache

program	$\text{CPI}_{\text{execution}}$	$\text{CPI}_{\text{mem stalls}}$: $\text{CPI}_{\text{I fetch stalls}}$	$\text{CPI}_{\text{D stalls}}$	$\text{CPI}_{\text{all mem stalls}}$	CPI
compress	1.056	.25	.127	.377	1.433
eqntott	1.006	.25	.156	.406	1.412
espresso	1.051	.25	.13	.38	1.431
gcc	1.143	.25	.1855	.4355	1.5785
li	1.167	.25	.24	.49	1.657
average	1.09	.25	.175	.425	1.515

Write-back Cache

program	$\text{CPI}_{\text{execution}}$	$\text{CPI}_{\text{mem stalls}}$: $\text{CPI}_{\text{I fetch stalls}}$	$\text{CPI}_{\text{D stalls}}$	$\text{CPI}_{\text{all mem stalls}}$	CPI
compress	1.056	.25	.1905	.4405	1.4965
eqntott	1.006	.25	.234	.484	1.49
espresso	1.051	.25	.195	.445	1.496
gcc	1.143	.25	.27825	.5285	1.67125
li	1.167	.25	.36	.61	1.777
average	1.09	.25	.2625	.5125	1.6025

Based on the average of the programs, the write-through cache is 1.6025/1.515 = 1.058 times faster than the write-back cache.

b) Everything is the same as part (a) above, except that now in the abscence of stalls, stores only take one cycle to execute for the write-through cache. This means that the $CPI_{execute}$ will be 1.0 for the write-through cache. The performance is summarized here:

Write-through Cache

program	$CPI_{execution}$	$CPI_{mem\ stalls}$			CPI
		$CPI_{I\ fetch\ stalls}$	$CPI_{D\ stalls}$	$CPI_{all\ mem\ stalls}$	
compress	1.0	.25	.127	.377	1.377
eqntott	1.0	.25	.156	.406	1.406
espresso	1.0	.25	.13	.38	1.38
gcc	1.0	.25	.1855	.4355	1.4355
li	1.0	.25	.24	.49	1.49
average	1.0	.25	.175	.425	1.425

Now, the write-through cache is 1.6025/1.425 = 1.125 times faster than the write-back cache.

5.8

a) For an ideal TLB, we will have no TLB misses.

We can write the equation for CPI like this:

$$\begin{aligned} CPI &= CPI_{execution} + CPI_{memory\ stalls} \\ CPI_{memory\ stalls} &= (\text{memory accesses per cycle}) \times (\text{miss rate}) \times (\text{miss penalty}) \end{aligned}$$

$CPI_{execution}$ is given as 1.5. Memory accesses per cycle are 1.2 (one for the instruction itself, and it is given that 20% of the instructions are data transfer instructions).

From page 391, the miss rate of the 16KB DM cache is 2.9%, the miss rate of the 16KB 2-way cache is 2.2%, and the miss rate of the 32KB DM cache is 2.0%.

The miss penalty depends on if the block is dirty. The penalty is

$$\text{penalty} = \%\text{ of time block is dirty} \times \text{memory write time} + \text{memory read time}$$

The problem states the 50% of the blocks are dirty, and that memory read or write time is 48 cycles (40 cycles of latency, plus 32/4 = 8 cycles to transfer the 32 B in each block). So, the penalty is 72 cycles.

We can now determine the CPI for each organization:

Organization	$CPI_{execution}$	$CPI_{mem\ stalls}$	CPI_{total}
16KB DM	1.5	2.5056	4.0056
16KB 2-way	1.5	1.9008	3.4008
32KB DM	1.5	1.728	3.228

b) The TLB will affect the memory stalls per cycle. The additional number of stalls per cycle is:

$$\text{TLB stalls} = \text{TLB accesses per cycle} \times \text{TLB miss frequency} \times \text{TLB miss penalty}$$

The number of TLB accesses per cycle is 1.2, the miss frequency is given as 0.2%, and the TLB miss penalty is 20 cycles. Thus, we can add to the memory stalls per cycle the TLB stalls, which work out to .048 stalls per cycle.

We can now determine the new CPI of the different configurations:

Organization	$CPI_{execution}$	$CPI_{mem\ stalls}$	CPI_{total}
16KB DM	1.5	2.5536	4.0536
16KB 2-way	1.5	1.9488	3.4488
32KB DM	1.5	1.776	3.276

c) If the caches are physically addressed, that means that the TLB access is on the critical path during every memory access, so it is particularly important that it be able to satisfy the request quickly. If the caches are virtually addressed, it might be more advantageous to make a larger or more complicated TLB that will have a longer hit time, but also a lower miss rate.

Additional Problem

a) The highest performance uses the greatest number of fastest parts. In this case, we are limited to 8 33 MHz parts. One processor will be able to do the non-parallelizable part of each frame in (2M * 20%) = 400k instructions. The parallelizable part will be split over all the processors, which means each has to execute (2M * 80% / 8) = 200k instructions. Thus, each frame will be decoded in the time it takes one processor to execute 600k instructions. This means we can get 55 frames/sec. The power is 490mW * 8 procs = 3.92 Watts. The energy consumed per frame is (3.92 Joules) / (55 frames per sec) = 71.2 mJ / frame.

b) From the table, we can see find the most energy efficient single processor configuration by dividing the power consumed by the number of operations performed per second:

Supply Voltage (V)	Frequency (MHz)	Power (mW)	Energy per cycle (nJ)
2.7	33	490	14.8
2.3	25	290	11.6
2.2	20	190	9.5
2.0	16	130	8.1

So, the 2.0V chips consume the least energy per cycle; if we use any more than one of these, there is some time when chips are not executing useful instructions (and are therefore wasting energy). So, a single 2.0V chip is the most energy efficient.

Since this one chip will be decoding the entire frame, we can get (16 MHz) / (2M instructions / frame) = 8 frames/sec from this configuration. It consumes 130 mW, and therefore consumes (.130 Joules) / (8 frames per second) = 16.25 mJ per frame.

c) We have to examine multiple configurations to find the best option. We need to have at least 15 frames / second; any less than that, and we won't meet the hard realtime demand. On the other hand, any more than that does us no good.

First, we examine the most energy efficient single chip design. We can find that only the 33 MHz chip will work, having the capability to deliver (33/2) = 16.5 frames/second. This will consume (.490 Joules) / (15 frames per sec) = 33 mJ/frame.

If we look at two-chip configurations, we find that 2 2.0V chips will not work. We can decode a frame in the time it takes one chip to execute (20%*2M) + (80%*2M/2) = 1.2M instructions. This means we can only decode 16/1.2 = 13.3 frames/sec with the 2.0V chips. But with the 2.2V chips, we can decode up to 20/1.2 = 16.7 frames/sec. This will consume (2 * .190 Joules) / (15 frames per sec) = 25 mJ/frame.

For the four-chip configurations, we can decode a frame in the time it takes one chip to execute (20%*2M) + (80%*2M/4) = .8M instructions. Now, we can use 2.0V chips; they will allow us to decode up to 16/.8 = 20 frames/sec. This will consume (4 * .130 Joules) / (15 frames per sec) = 35 mJ/frame.

So, the most best choice is to use 2 2.2V chips. As stated before, this will use 25 mJ/frame.