# 5008: Computer Architecture
# HW#4 Solution

## ◆ Limits on Instruction-Level Parallelism

### 3.1

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

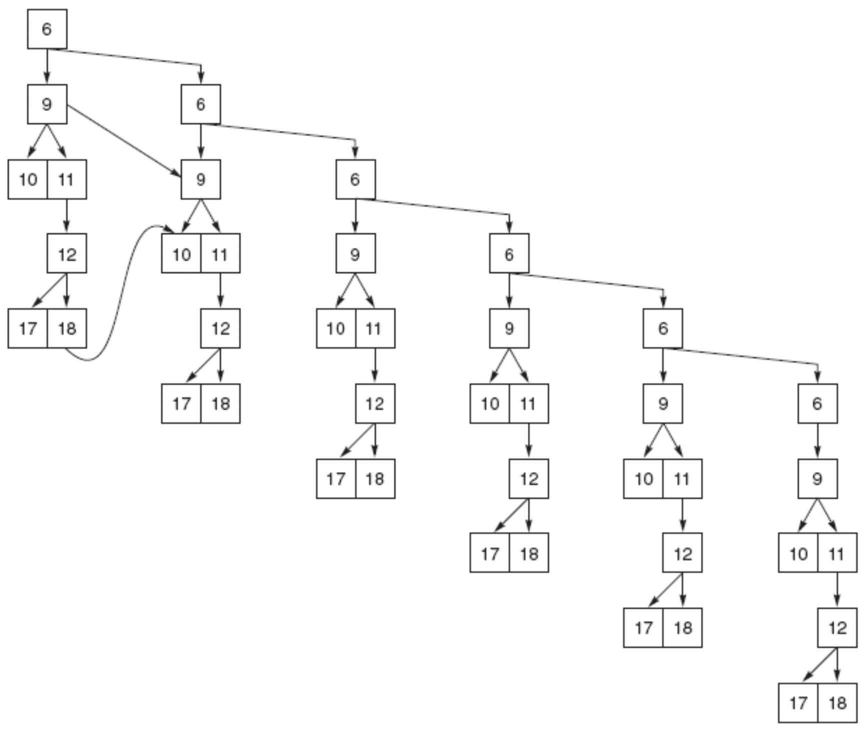

**Figure L.20 Dynamic dependence graph for six insertions under the ideal case.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Figure L.21** Hash table code example.

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

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

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

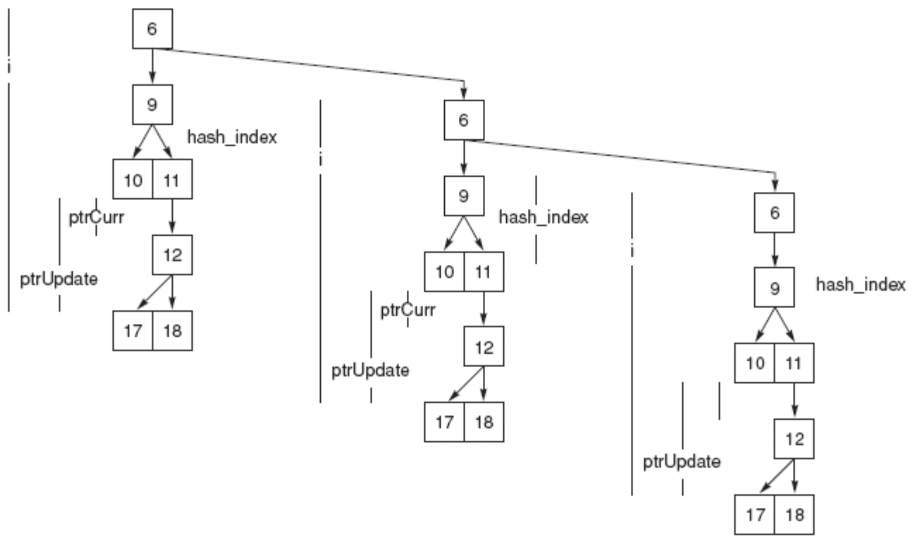

**Figure L.22** Register lifetime graph for the ideal case.

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

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

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

**Figure L.23 Instruction issue timing.**

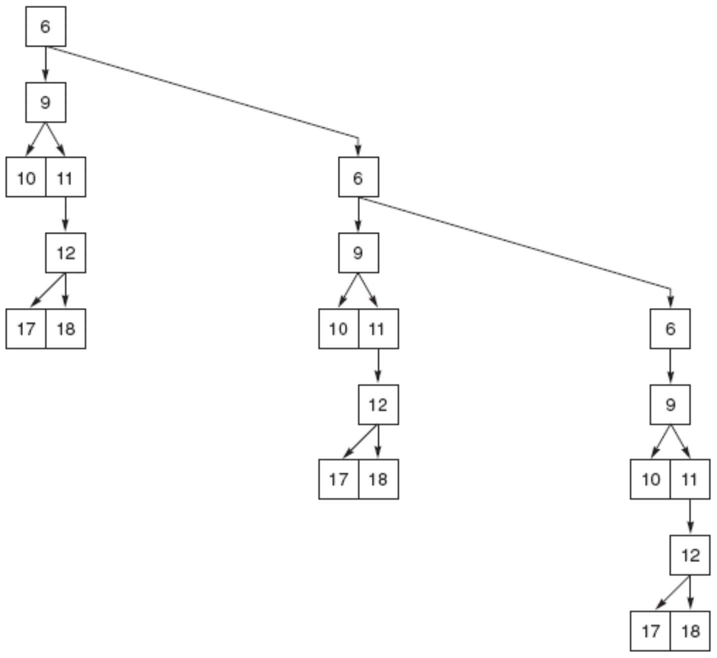

**Figure L.24 Execution timing due to limited issue rate for the ideal case.**

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

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

## ◆ Review of Memory Hierarchy

1

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.

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

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.

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

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.

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

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.

## 2

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.

# 3

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.

**4**

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.