

Principles in Computer Architecture

CSE 240A Homework Three

March 5, 2015

*Only Problem Set Two will be graded. **Turn in only Problem Set Two** on March 13, 2015 (Friday) by 1:00 pm.*

*Submit homework via csemoodle <csemoodle.ucsd.edu> as a **PDF**. Other formats may or may not be viewed and evaluated accurately.*

1 Problem Set One

- Hennessy & Patterson (5th Ed) B.1
- Hennessy & Patterson (5th Ed) B.3
- Hennessy & Patterson (5th Ed) B.9
- Hennessy & Patterson (5th Ed) B.10
- Hennessy & Patterson (5th Ed) 2.2
- Hennessy & Patterson (5th Ed) 2.4
- Hennessy & Patterson (5th Ed) 2.9
- Hennessy & Patterson (5th Ed) 2.11
- Hennessy & Patterson (5th Ed) 2.12

2 Problem Set Two

1. Loop Unrolling and Caches

Loop unrolling is a useful technique to extract instruction level parallelism. The number of iterations that can be usefully unrolled is typically determined by architectural parameters such as the issue width and the number of functional units. In the unrolling process, one should also take into consideration the cache behavior in terms of *temporal and spatial locality* of data accesses.

Assume the processor has an **unlimited** number of functional units and can issue up to 4 instructions at a time. The L1 data cache is a direct-mapped cache with 2 words/block, 16 sets, and 1 cycle hit time. For each of the following code segments, please determine whether loop unrolling is useful or not. If you think it is useful, please identify the maximum number of iterations that can be usefully unrolled. If you think loop unrolling is useless, please briefly describe the reasoning.

```
for i = 0 to 1000 by step 1
    A[i] = A[i+10] + A[i+30];
end for
```

```
for i = 0 to 1000 by step 1
    A[i] = A[i+6] + A[i+66];
end for
```

```
for i = 0 to 1000 by step 2
    A[i] = A[i+8] + A[i+138];
end for
```

2. Prefetching and Data Reuse

As we have been discussing the memory subsystem in our CSE240A class, we have concentrated on various techniques for improving its performance, perhaps in the process somewhat forgetting to emphasize the fact that the various levels of the memory hierarchy always represent somewhat constrained resources, whose capable management may deliver significantly improved results. Perhaps this oversight has been due to the fact that the highest levels of the memory hierarchy, i.e., the register space is typically under the complete control of the compiler, while the cache management even though under hardware control is mostly discussed in statistical terms. Yet prefetching does provide us a possibility of discussing the impact of memory hierarchy techniques in a more deterministic manner. This question invites you to think about these ideas.

(Part A) As we discussed in class, the traditional prefetching is to prepare the data to reside in the L1 data cache when it is needed. While being able to prefetch the needed data is useful, occasionally as we discussed in class such prefetching may have adverse effects. Please identify the possible causes under which prefetching may generate adverse effects.

(Part B) A similar idea that can be entertained is to prefetch from memory (or L2 if applicable) directly into the register space, but with the additional proviso that the loading of the result into the L1 cache be disabled. Please evaluate the impact of this disablement and prefetching into registers.

To explore some of the ideas raised in the previous part, we invite you to look more concretely at a piece of code. The code given below is a simple loop whose memory reference patterns as can be seen are driven by 4 references comprised of 2 references each to two arrays. The execution of some of these references is control-dependent. While the references to the *B* array are purely *reads*, the references to the *A* array are comprised of a run-ahead *read* (subject to a control dependency) and a possible reuse of that location later on by a *store*.

The organizational aspects of the processor on which the loop is going to be executed are as follows: all the data stored in *A*[] and *B*[] are 4 bytes long with the starting addresses (in bytes) of arrays *A*[] and *B*[] at 0 and 2828 (in bytes), respectively. The data cache is a direct-mapped cache with 1 word/block, 64 sets, and 1 cycle hit time. A cache miss of a **load** instruction requires the main memory to be accessed, which takes 30 cycles. However, a cache miss of a **store** instruction will not block the processor.

```
for (i=0;i<1000;i++) {
    load R1, B[i];
    ...
    if () {                //branch not taken
        load R2, A[i+4];
        ...
    }
    else {                 //branch taken
        load R2, B[i+2];
        ...
    }
    S.D R3, A[i+2];
}
```

(Part C) Assume that no prefetching instruction is inserted in the baseline design. Please analyze *the access pattern to each cache set*, as well as *the miss rate of each load/store instruction in the loop body*, assuming that the probability for the branch to be taken is *p*.

(Part D) Since there exists potential temporal reuse in both arrays $A[]$ & $B[]$, we are considering adding an explicit instruction at the beginning of the loop, to prefetch either $A[i + 4]$ or $B[i + 2]$ in the *if-then-else* statement directly into the cache. Assume that the prefetch instruction can always bring data into the cache in time. If the word to be prefetched is already in the cache, the prefetch instruction will transform itself into a *noop*.

Obviously, the amount of reduction in the average memory access time achieved by either prefetching choice is determined by the taken/not-taken statistics of the branch. In order for the prefetching of $A[i + 4]$ to be more beneficial, what should the T/NT distribution statistics of the branch be? Please clearly show your analysis and your calculation steps. *For partial credit, you may want to analyze the impact of each prefetch choice to the miss rate of each load/store instruction in the loop body.*

3. Cache Behaviors

Assume two different cache organizations: (a) Direct-mapped cache and (b) 2-way set associative cache with an LRU replacement policy, both with 1 word/block and a 6-bit index.

(Part A) For the following unfinished code segment with a series of incomplete read references from arrays, please complete the code by filling in the index portion of the array references to **minimize** the cache *reuse* possibility (*maximize* cache *miss rate*) for both cache organizations described above. Please also give a brief explanation for your code behaviors. Assume the starting addresses of A[], B[] and C[] arrays in memory are 0, 520 and 144 (byte address), respectively.

(a) Direct mapped cache:

```
int A[128], B[128], C[128]; // 4 byte integers
```

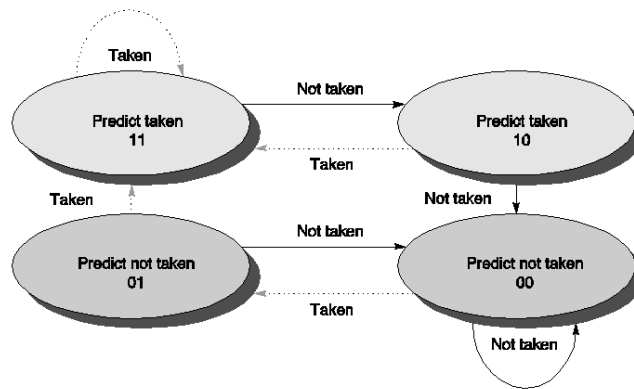
```
for i=0,116
    A[i+    ]
    A[i+    ]
    B[i+    ]
    B[i+    ]
    C[i+    ]
end
```

(b) 2-way set associative cache:

```
int A[128], B[128], C[128]; // 4 byte integers
```

```
for i=0,116
    A[i+    ]
    A[i+    ]
    B[i+    ]
    B[i+    ]
    C[i+    ]
end
```

(Part B) We are now considering a 2-way set associative cache with an LRU replacement policy and also interested in incorporating a *way-prediction* hardware with a 2-bit saturating counter for each set. For the similar code segments presented in (Part A), please complete the code by filling the index portion of the array references along with the proper initial conditions of the way predictor to make the way prediction *continuously wrong* and also give a brief explanation for your code behaviors and way-prediction behaviors as well. We are giving you below the FSM for a 2 bit history way predictor, which is identical to the one we studied in class.



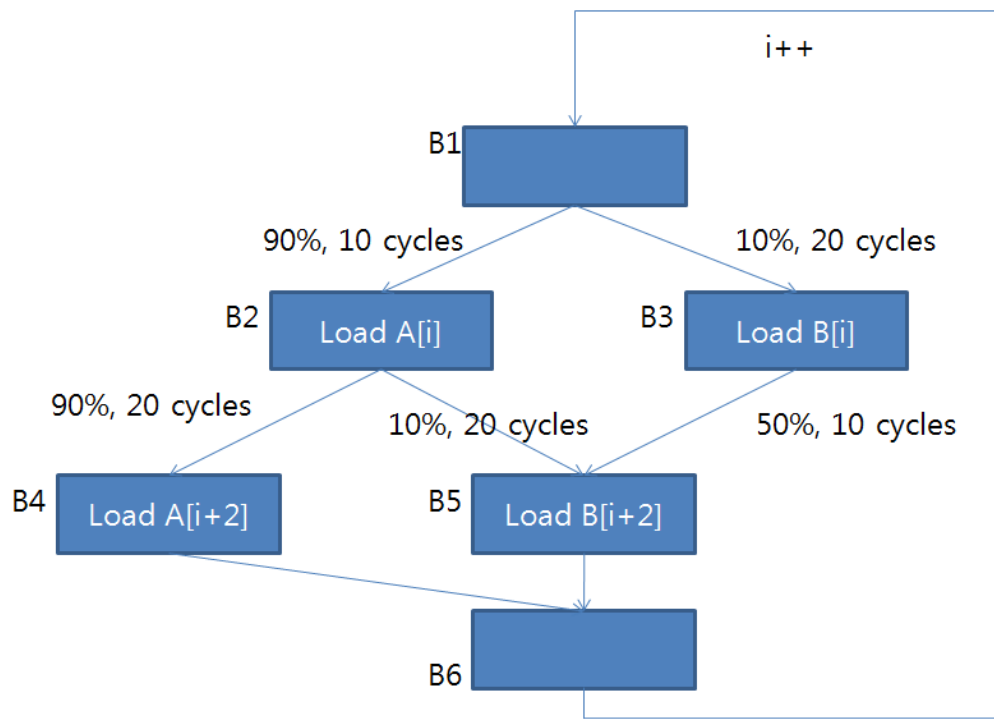
```
int A[128], B[128], C[128]; // 4 byte integers
```

```
for i=0,116
```

```
    A[i+    ]
    B[i+    ]
    A[i+    ]
    B[i+    ]
    C[i+    ]
```

```
end
```

(Part C) We now come back to the direct mapped cache, with 1 word/block but try to utilize software prefetching schemes to enhance the cache behavior. For the following Control Flow Graph (CFG), which consists of 6 basic blocks, all the memory references within the code fragments are shown in the CFG. The CFG shows the T/NT probabilities along with the cycle times between consecutive basic blocks, generated by static profiling, of both branches. Assume the starting addresses of integer arrays A[] and B[] are 0 and 520 (byte address). On a read cache miss, the memory would need to be accessed, which takes an additional 30 cycles. On the other hand, a write cache miss will not block the processor.



We now have a total of four different load instructions from the basic blocks B2, B3, B4 and B5 which can be prefetched. Please rank the 4 different prefetching solutions insofar as the miss penalty is concerned.