Only Problem Set Two will be graded. Turn in only Problem Set Two before December 7, 2012, 11:00 AM.

1 Problem Set One

- Hennessy & Patterson (5th Ed) B.1
- Hennessy & Patterson (5th Ed) B.3
- Hennessy & Patterson (5th Ed) B.4
- 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
2 Problem Set Two

1 (Caches)

The following table contains a set of cache optimization techniques. Each of these techniques possibly impacts the Miss penalty, the Miss rate, and the Hit time. Please assess whether each of the proposed techniques has a positive (“+”), negative (“-”) or no (“N”) impact on each of the three attributes. Please provide a brief explanation in the space given for your answer for each technique.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss penalty</th>
<th>Miss rate</th>
<th>Hit time</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multilevel caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical word first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging write buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Victim caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Larger block size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Larger cache size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher associativity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudoassociativity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware prefetching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small and simple caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avoiding address translation for cache indexing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipelined cache access</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace caches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2 (Way prediction)

Using the *way prediction* technique, only one way in an associative cache needs to be accessed if the way of the next cache access is predicted correctly. However, a mis-prediction results in the check of the other ways for matches in the next clock cycle.

(Part A) One limitation of the *way prediction* technique is that the way predictor itself needs to be accessed before indicating a cache access. Assume accessing one way of the cache takes $T_{\text{one}}$ ns, while accessing all the ways simultaneously takes $T_{\text{all}}$ ns. Furthermore, assume the way predictor itself needs $T_{\text{pred}}$ ns to access. What should the prediction accuracy of the way predictor be, in order for the way predictor to be able to improve the average hit time of the cache?

(Part B) Another potential benefit of way prediction is the possible reduction in energy consumption, since only one way needs to be accessed if the next cache access is predicted correctly. For an *n-way* associative cache, assume the energy consumed in accessing each way of the cache is $E$, while the energy consumed in accessing the way predictor is $P$. What should the prediction accuracy of the way predictor be, in order for the way predictor to be able to reduce the overall energy consumed in cache accesses?
The following loop which has 4 load/store accesses in each iteration is going to be executed exclusively on a particular processor with the memory configuration of an L1 cache that is a two-way associative cache with 1 word per block and a 6 bit index. Furthermore, assume the starting addresses (in bytes) of arrays \( A[\ ] \) and \( B[\ ] \) to be 0 and 4100, respectively. All the data stored in \( A[\ ] \) and \( B[\ ] \) are 4 bytes long.

\[
\text{for } i = 0 \text{ to } 1000 \text{ by step 1} \\
\hspace{1em} A[i] = i \times B[i+1]; \\
\hspace{1em} B[i] = i + B[i+3]; \\
\text{end for}
\]

(Part C) For each set of the cache, an engineer has proposed to use a 2-bit saturating counter, similar to what is used in a local branch predictor, to generate way prediction. The counter is incremented/decremented according to the particular way that is accessed within that set. Please comment on the prediction accuracy of this counter for the loop presented above.

(Part D) Since assigning a 2-bit counter to each set of the cache introduces a lot of hardware overhead, another engineer has proposed to keep record of the \textit{global history} regarding the ways of the most recent accesses to the cache for way prediction. In order to achieve maximal prediction accuracy, what is the \textbf{minimal} number of bits of global history that should be recorded? Compared with the predictor used in (Part C), does this predictor achieve a reduction in hardware overhead? Please provide a justification for your answer.
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:

```c
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:

```c
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.
4 (Spatial & temporal localities)

Assume a direct-mapped cache with 1 word/block and 6 bit index, and write-back and write-allocate policies in effect. We want to reduce the miss rate for certain applications, by implementing only one of the following two optimizations. Both optimizations keep the cache size constant.

- Increase the block size to 2 words/block.
- Increase the associativity to 2-way, with an LRU replacement policy.

(Part A) For each of the following two code fragments, determine which of the two optimizations provided above can help to reduce the miss rate. Provide a brief reasoning for your answer.

```plaintext
for i = 0 to 60 by step 2
    A[i] = i*i;
end for

for i = 1 to 61 by step 2
    A[i] = 2*i;
end for
```

```plaintext
for i = 0 to 120
end for
```

(Part B) For a program with strong spatial locality, which of the two optimizations listed on the previous page will help the program get more hits in the cache? On the other hand, for a program with strong temporal locality, which of the two optimizations will help the program get more hits in the cache? Provide a brief reasoning for these two cases, respectively.
(Part C) Assume the following loop is going to be executed.

```plaintext
for i = 1 to 120
    B[i+4] = B[i-1] + C[i-1];
    C[i+5] = A[i+8] * 2;
end for
```

Since three arrays are accessed simultaneously per iteration, a **partitioned cache** is suggested to be used here, in which A[], B[] and C[] are stored separately in their reserved space.

Assume the cache has 32 lines in total, and the block size is 1 word/block. A[], B[] and C[] contain **128 integers**, respectively. In order to capture as much temporal reuse as possible for all A[], B[] and C[], what is the **minimum** number of cache lines that each partition needs to have, respectively? Explain your answer clearly.

(Part D) Since the **partitioned cache** stores different arrays separately, a new addressing method should be employed to map the virtual address given by the processor to the exact data stored in the cache. In order to implement a correct addressing method, what information is indispensable? How is a virtual address translated in this new addressing method? Explain your answer clearly.