

CSE 240A Homework Three

May 24, 2011

*Only **Problem Set Two** will be graded. Turn in only **Problem Set Two** before June 3, 2011, 4:00pm.*

1 Problem Set One

- Hennessy & Patterson (4th Ed) 5.2(a)(b)(c)
- Hennessy & Patterson (4th Ed) 5.4
- Hennessy & Patterson (4th Ed) 5.5
- Hennessy & Patterson (4th Ed) 5.7
- Hennessy & Patterson (4th Ed) 5.18

2 Problem Set Two

1 (Simultaneous Multithreading) (30 points)

One critical issue in computer architecture is the limitation of instruction-level parallelism, which motivates the use of the *multithreading* technique to allow multiple threads to share the functional units of a single processor in an overlapping fashion. Recall that to permit this sharing, the processor needs to add more hardware to preserve the independent state of each thread.

(Part A) There is a raging debate going on within a company that is trying to adopt the SMT technique for a new processor they are designing. Being newcomers to the block, their business strategy is to undercut the competition particularly on price (of course at no impact on performance!). The engineers are involved in heated discussions trying to figure out the feasibility & desirability of adopting shared structures and those in the camp of shared structures are arguing further whether the shared structure is the only way to go, whether the shared structure is neutral performance-wise or whether the shared structure will impose a stiff performance degradation on the system. Of course, this being a low-cost provider, no additional functionality or hardware is envisioned in the case of the shared structure. Here are the four possibilities:

A The shared structure will not work, period.

B The shared structure not only will work but is the only way to go; anything else will result in incorrectly functioning processors.

C The shared structure will work and it does not make a difference performance-wise whether the structure is shared or not.

D The shared structure will work but the performance will degrade stiffly.

For the following 5 cases, decide which of the 4 options above holds and provide a brief reasoning.

Instruction fetch queue:

Register renaming table:

L1 cache:

Load/store queue:

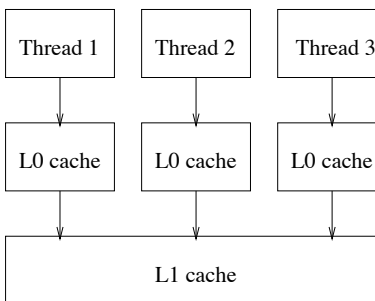
Reorder buffer:

(Part B) One of the limiting factors on an SMT is the bandwidth of getting instructions into the fetch queue. While giving preference to the preferred thread(s) is beneficial in terms of the latency of the preferred thread, it may impact the ability of the processor to utilize fully the processing units. Why?

If your preferred thread(s) were straightline DSP code with little branch behavior & highly analyzable memory accesses, which instruction fetch queue loading policy will you suggest?

For a *preferred* application with heavy conditional utilization & pointer chasing, which instruction fetch queue loading policy would you suggest?

(Part C) Another critical issue in computer architecture is the increasing performance gap between processors and memories. Although more and more levels of caches are being inserted between the processor and memory, the L1 cache access still cannot be accomplished within a single clock cycle. Consequently, the design team is thinking of furthermore providing a dedicated L0 cache for each thread in this multi-threading superscalar processor, which can execute 3 threads in parallel. These L0 caches are organized as small direct-mapped caches to reduce hit time. The threads share the L1 cache in common, as shown in the following figure.



During execution, each thread will access its own L0 cache to obtain data. The shared L1 cache is only accessed for L0 cache misses. Assume the threads are independent, and the data presented in each L0 cache are also captured in the shared L1 cache. Furthermore, assume the small L0 cache can be accessed in 1 cycle, while the L1 cache takes 2 cycles to access. If for each thread the average miss rate of the L1 cache is M , what should the miss rate of the L0 cache be, in order for the L0 cache to be beneficial in improving the overall performance of load/store accesses?

2 (Way Prediction vs. LRU Replacement) (30 points)

A way predictor employs extra hardware to record cache access histories. If a way predictor correctly predicts the way of the next cache access, only one way in an associative cache needs to be accessed. On the other hand, a mis-prediction results in the check of the other ways for matches in the next clock cycle, thus increasing the hit time.

An associative cache usually employs a *least recently used* (LRU) replacement policy, which requires extra hardware to keep track of the recently accessed ways. Since there exists a strong correlation between the LRU data and the way to be accessed next, in this question we consider the idea of merging the two pieces of hardware together, that is, *to use the replacement policy to attain way prediction*.

The following loop is going to be executed exclusively on a processor. An interesting aspect about this loop is that the computation at each iteration is composed of **three** parts, each of which is initiated by a load and terminated by a store. Furthermore, there are no register-level true dependences among the three groups; perhaps another way of saying it is that each store instruction indirectly depends on the preceding load through a register chain, while load instructions have no true register dependences on any of the preceding store instructions within the same iteration.

The L1 cache is a **two-way** associative cache with 1 word/block and 128 sets. Assume that the cache is empty at the beginning of this loop, while if both ways are empty the data is always placed in way 0. All the data stored in $A[]$ and $B[]$ are 4 bytes long. The starting addresses (in bytes) of arrays $A[]$ and $B[]$ are 0 and 520 (in bytes), respectively.

```
for (i=1;i<2000;i++) {  
    L.D R1, A[2i+4];  
    ...  
    S.D R2, B[2i-2];  
    L.D R3, A[2i+1];  
    ...  
    S.D R4, A[2i];  
    L.D R5, B[2i-1];  
    ...  
    S.D R6, B[2i+1];  
}
```

(Part A) Assume a single bit is added to each cache line to record the LRU block. This bit is also used to generate way prediction since this cache is only 2-way associative; the way of the MRU block is always predicted as the way of the next access to that cache line. Please comment on the accuracy of this prediction mechanism for the loop presented above. Specifically, what percent of the hit accesses can be correctly predicted by the way predictor in terms of the way to be accessed? *For partial credit, you may want to analyze the hit/miss history of each load/store instruction, as well as the access pattern of each cache line.*

(Part B) One common optimization strategy is to use compiler techniques to transform the code so as to improve dynamic cache performance, for example, to reduce miss rate.

As you may have noticed, the prediction accuracy of a way predictor is strongly affected by the order of the memory accesses in a repetitive pattern. For this loop, do you think that the *prediction accuracy* of the way predictor described in (Part A) can be improved by compiler techniques without hurting the cache miss rate? If you think it can, please address the technique needed, the code transformation performed, and the amount of improvement in prediction accuracy. If you think that compiler techniques cannot help the way predictor without hurting the cache miss rate, please clearly state your reasoning.

3 (Skewed Associative Caches) (40 points)

One technique for reducing the number of conflict misses is to use *skewed associative* caches that employ distinct indexing mechanisms for various ways. For example, in a 2-way skewed-associative cache, way 0 is indexed using the least significant address bits, the same as in a regular cache. The indices of way 1, however, are generated through XORing the original indices with a set of bits selected from the tags. In this way, data originally mapped to the same cache line can be mapped to different lines in way 1, thus reducing the number of conflicts.

(Part A) In a 2-way skewed associative cache design, a crucial issue is the selection of suitable XORing functions for accessing way 1. Specifically, assume that the 32-bit address is partitioned into 3 segments, 4-bit block offset, 10-bit index, and 18-bit tag. Furthermore, it has been found that the program only accesses data within the following two disjoint address ranges: [0002 0000: 002F FFFF] and [1040 0000: 107F FFFF].

To effectively reduce potential conflict misses, which part of the tag segment is more appropriate to be selected for XORing; the most significant bits or the least significant bits? Please clearly state your choice and briefly explain your reasoning.

(Part B) A common technique employed to reduce hit time is to avoid address translation during cache indexing. The cache size can be determined according to memory page sizes, so that the index fields of the virtual and physical addresses are identical.

In a 2-way skewed-associative cache, however, not only the index field but furthermore a subpart of the tag field is needed while indexing way 1. Therefore, the design should consider a crucial issue regarding whether the *virtual* or the *physical* tags should be used during cache indexing.

The following table shows 3 pairs of design decisions of a 2-way skewed-associative cache. These decisions can produce 8 distinct design configurations. For each configuration, please evaluate its validity according to 1) whether an access is a hit or miss can be correctly identified, and 2) whether the replaced block, if any, can be successfully identified.

If you think a configuration is invalid, please provide a brief rationale. Otherwise if you think it is valid, please comment on the *cache access time* compared to the traditional cache of the same configuration.

A Store <i>virtual</i> tags in cache	a Use <i>virtual</i> tags for XORing	α Use a write through policy
B Store <i>physical</i> tags in cache	b Use <i>physical</i> tags for XORing	β Use a write back policy

- $A + a + \alpha$

- $A + a + \beta$

- $A + b + \alpha$

- $A + b + \beta$

- $B + a + \alpha$

- $B + a + \beta$

- $B + b + \alpha$

- $B + b + \beta$

(Part C) A widely used replacement algorithm in a 2-way associative cache is the least-recently-used (LRU) algorithm. LRU can be simply implemented through adding a single bit to each cache line; the bit is set to 0 if way 0 is accessed most recently, and to 1 if way 1 is accessed most recently.

Can LRU replacement policy be attained using a single bit in a 2-way skewed associative cache? If you think it can, please give the mechanism for setting and resetting the bit. Otherwise, please give a brief reasoning regarding why LRU cannot be attained.