Only Problem Set Two will be graded. Turn in only Problem Set Two before December 4, 2008, 11:00am.

1  Problem Set One

• Hennessy & Patterson (4th Ed) 5.2
• Hennessy & Patterson (4th Ed) 5.4
• Hennessy & Patterson (4th Ed) 5.5
• Hennessy & Patterson (4th Ed) 5.6
• Hennessy & Patterson (4th Ed) 5.8
• Hennessy & Patterson (4th Ed) 5.18

• A cache may use a write buffer to reduce write latency and a victim cache to hold recently evicted (nondirty) blocks. Would there be any advantages to combining the two into a single piece of hardware? Would there be any disadvantages?

• As caches increase in size, blocks often increase in size as well.

  1. If a large instruction cache has larger data blocks, is there still a need for prefetching? Explain the interaction between prefetching and increased block size in instruction caches.

  2. Is there a need for data prefetch instructions when data blocks get larger? Explain.
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 the 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?
2 (Way prediction) (40 points)

Using the *way prediction* technique, only one way in an associative cache needs to 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 $T_{one}$ ns, while accessing all the ways simultaneously takes $T_{all}$ ns. Furthermore, assume the way predictor itself needs $T_{pred}$ 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.

for $i = 0$ to 1000 by step 1
  $A[i] = i \times B[i+1]$;
  $B[i] = i + B[i+3]$;
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 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 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 (Compiler-based prefetching)  (30 points)

In this question, you are going to evaluate the prefetching technique for the following Control Flow Graph (CFG), which consists of 7 Basic Blocks. All the memory references within the code fragment are shown in the CFG. The CFG also shows the T/NT probabilities, generated by static profiling, of both branches.

Assume the program is going to be executed on a particular processor with the following memory configuration: L1 cache is a direct-mapped cache with 1 word/block, 6 bit index, and 1 cycle hit time. On a cache miss due to a load instruction, the memory would need to be accessed, which takes an additional 20 cycles. On the other hand, a cache miss of a store instruction will not block the processor.

Because of the significant cache miss penalty, you are considering adding an explicit prefetch instruction to reduce the miss rate. If the word to be prefetched is already in the cache, the prefetch instruction will transform itself into a noop.

For the following parts of this question, assume the starting addresses (in byte) of arrays A[], B[] and C[] to be 260, 0, and 400, respectively. All the data stored in A[], B[] and C[] are 32-bit integers. The loop body itself is going to be executed 30 times.
(Part A) Since the basic block $B_4$ is on the most frequently executed trace, you want to add an explicit prefetch instruction for the load of $A[i]$ in $B_4$. Since the prefetch instruction needs to be inserted at least 20 cycles ahead, the only possibility is to insert it in $B_1$ (prefetch instructions across loop iterations are not considered). However, it is also possible that sometimes the insertion of prefetch instructions may cause performance degradation. For this particular program, please analyze whether the inserted prefetch instruction will cause performance degradation or not. How much speedup (or degradation) can be achieved if the prefetch instruction is going to be inserted? Your answer should take into account the T/NT probabilities of branches.

(Part B) Since inserting a prefetch instruction for $A[i]$ may cause performance degradation, another possible optimization is suggested to reduce the execution time of this particular loop. Instead of inserting a prefetch instruction in $B_1$ for the load of $A[i]$ in $B_4$, a prefetch instruction is inserted in $B_1$ for the load of $B[i+1]$ in $B_5$. Compared with the idea in (Part A), is this idea better or not? Compared with the original code, how much speedup (or degradation) can be achieved if this prefetch instruction is going to be inserted? Your answer should take into account the T/NT probabilities of branches.