CHAPTER 9 Homework Solutions

1.

(a) For this branch, an always taken static prediction will always result in a 47% misprediction rate. This is not much better than random guessing.

(b) A bimodal predictor will provide very accurate predictions, assuming that the 53% taken branches are clustered separately from the 47% not-taken branches. For example, if the first 53% of all branches are taken, then the bimodal counters will quickly saturate and provide the correct prediction for this portion of the program. When the dominant direction of the branch switches, then the counter will mispredict a few times and then learn to predict not-taken. Overall, if the branch exhibits the same outcome many times in a row, the bimodal predictor will perform well. The less often the branch direction changes, the fewer mispredictions the bimodal predictor will make.

(c) The branch outcomes must repeat in a relatively short pattern for the local history predictor to learn and predict well. If the pattern is too complicated or too long to be captured by the local history register, then the local history predictor will not be able to learn the entire pattern, which will result in mispredictions. On the other hand, if the pattern is short enough, than a local history predictor can achieve a 100% prediction rate after the initial period that it takes to learn the pattern. The branch pattern may change part way through the program, and there will be a period where the local history predictor will make many more mispredictions as it tries to learn the new pattern. This is analogous to the bimodal predictor making mispredictions when the dominant branch direction changes. If the patterns do not change very often, then the local history predictor will make fewer mispredictions. If the 53% taken branches do not repeat in any discernable pattern, than the local history predictor will not be effective.

(d) Eager execution is not effective at predicting branches because it is not a branch predictor! Rather, it is a means for dealing with hard to predict branches. For branches that are seemingly "random", eager execution may provide a performance benefit since no matter which path is the correct one, some work on the correct path will always be done by the time the processor has resolved the actual branch direction. The flip-side of this is that the processor is also guaranteed to execute instructions on the wrong-path thus wasting power and execution resources. If the branch is very predictable, eager execution may actually slow down execution because the correct path instructions must now compete with the wrong path instructions for processor resources (e.g. functional units, register read ports, etc.).

2.

Counter State	Prediction	Actual Outcome	Correct?	Next State
ST	T	T	✓	ST
ST	T	T	✓	ST
ST	T	T	✓	ST
ST	T	N	✗	WT
WT	T	N	✗	WN
WN	N	T	✗	WT
WT	T	T	✓	ST
ST	T	T	✓	ST
ST	T	N	✗	WT
WT	T	N	✗	WN
WN	N	T	✗	WT
WT	T	T	✓	ST
ST	T	T	✓	ST
ST	T	N	✗	WT
WT	T	N	✗	WN

ST = Stongly Taken, WT = Weakly Taken, WN = Weakly Not-Taken

This table shows the progression of the counter state, the counter's predictions, the actual outcomes, whether the prediction was correct, and the transition to the next state. The column labeled "Actual Outcome" lists the original branch outcome sequence. For this sequence, there were 7 correct predictions out of 15 predictions overall for a 46.67% prediction rate (53.33% misprediction rate).

3. Even though the branch outcome pattern from Problem 2 repeats every five branches, a branch history register of only the last **three** outcomes is needed. This PHT is illustrated below:

PHT Index	Corresponding Prediction
000 (NNN)	X
001 (NNT)	T
010 (NTN)	X
011 (NTT)	T
100 (TNN)	T
101 (TNT)	X
110 (TTN)	N
111 (TTT)	N

For example, whenever the previous three branch outcomes were NTT, the following outcome is always T. Therefore the PHT stores a T in the entry for 011 (NTT). While this table only uses T's and N's, a real predictor would use two-bit saturating counters and so each T-prediction in the example PHT above would really have a counter in either the WT or ST state.

If a history register of length 5 was used, then the PHT would have 32 entries, where only five of the entries were in use.

4. Using a larger history register than necessary may increase the *training time* of the predictor. If a particular branch has an outcome that is always the opposite of the sixth previous branch, then the PHT must learn the patterns 1xxxxx → 0 and 0xxxxx → 1 (where 'x' is either 0 or 1). Since there are six bits to the pattern, it may take $2^6 = 64$ PHT entries to store the entire mapping. It takes time just to observe all of the patterns, and then it may require observing each pattern more than once to learn the corresponding prediction. Adding additional unnecessary history bits increases the number of patterns, and thus the training time. This in turn may lead to more mispredictions.

For a traditional Two-Level global history predictor, increasing the history length means decreasing the number of branch address bits used in the PHT index. This may lead to an increase in aliasing between unrelated branches. For a gshare predictor, these effects are not as pronounced and the disadvantages of a longer history register are mostly limited to what has already been discussed.

The problem states that *most* of the branches only require six bits of history, which implies that there are some that require more. Whether or not it is a good idea to increase the history register length to help these other branches depends on how much the additional training time hurts the prediction rates of the other branches that do not require as much history, and it depends on how much performance is being lost to the branches that do require longer histories.

What makes choosing an appropriate history length difficult is that different programs exhibit different behaviors. A history length of four may be sufficient for one application, but another may require twelve. Choosing a history length of twelve may cause the first program to suffer from increased predictor training times. Choosing a history length of four may result in a high misprediction rate for the second program due to insufficient prediction context.

5. The main tradeoff is one between capacity and conflict aliasing. Adding bits for tags (the tags may be partial tags, similar to partial resolution for BTBs) decreases conflict aliasing, but this comes at the cost of additional bits. These bits could have alternatively been used to increase the number of entries in the branch prediction table (e.g. larger PHT), which in turn decreases interference due to insufficient capacity. For example, instead of adding a 6-bit tag to a 2-bit counter, one could instead implement an untagged table that is four times larger than the original (i.e. each 6-bit tag could instead be used to implement three separate 2-bit counters). For larger hardware budgets, further adding capacity has diminishing returns, and so adding associativity is likely to be more beneficial. At smaller hardware budgets, the amount of capacity aliasing is likely to be the predominant cause for interference, and so using the bits to increase the table size instead is likely to be the better choice.

6. You can write a specially designed program in a low level language such as C (without compiler optimizations enabled) or even directly in assembly that is composed of carefully crafted branches that have taken/not-taken patterns chosen specifically to test the branch predictor. Such *microbenchmarks* would be written to exhibit branch patterns that your new predictor can handle that other predictors should not be able to predict well, and there should also be patterns that you know will cause problems for your predictor. By running these microbenchmarks on your predictor simulator, you can compare the reported prediction accuracy to what is expected. For example, a global history predictor with a 3-bit history register should be able to handle the following segment nearly perfectly:

```
start:
        x=0;
        if(x)
           goto end;
        x=0;
        if(x)
           goto end;
        x=1;
        if(x)
           goto start;
end:
```

This code would exhibit a repeating branch pattern of 001. The technique of crafting specialized programs to test certain behaviors is a useful method for testing not only branch predictors, but all kinds of microarchitectural functionalities.

7. The path information provides different information than the branch history. The example of Figure 8-19 showed a case where the path history provided more information than the branch history. Now consider the following code:

```
if(condition1)   /* branch A */
   x = 30;
if(condition2)   /* branch B */
   x = -1;
if(condition3)   /* branch C */
   x = 42;
if( x>15 || x < 0)   /* branch of interest */
   do_something();
```

In this example, each branch will be executed regardless of the whether branches A, B and C are taken or not-taken. If branch A is taken, then the variable x will be set to 30, and then the code continues to branch B. If branch A is not-taken, then the code will immediately proceed to branch B. In any case, by the time the program reaches the last branch, the global path history for the last three branches will be ABC, which does not provide any useful information for making the final prediction. On the other hand, global branch history could potentially enable perfect prediction of the final branch. Therefore the path history does not provide a superset of the branch history information. There are some cases where path history is needed, others where branch history is required, and there could even be branches that require some of both types of information.

8. Intuitively, combining a global-history and local-history predictor may seem to make more sense because some branches may need one type of history to predict, while others may need the other type. Combining two predictors that use the same type of history may seem to be completely useless since it does not appear that the second predictor makes use of any new information. On the other hand, two global history predictors (even if they have the exact same history length) that use different prediction algorithms (e.g. Bi-Mode vs. gskewed) will result in different interference patterns in their respective PHTs. There may be cases where branch aliasing in a gskewed predictor causes a misprediction, but since the hashing function for a Bi-Mode predictor is different, there may not be any interference, and this results in a correct prediction. Using two global-history predictors in this situation may result in a decrease in interference-based mispredictions.

9. Using only the branch address to predict the target is insufficient as there are many possible targets for the branch. A possible solution is to include more (different) information than just the branch address in making the prediction. One could possibly use global branch history or path history, as the outcomes of the last several branches may have determined the computed branch's target such as in the following example:

```
f = &function1;          /* f is a function pointer */
if(some_condition)
   f = &function2;
f(x);                    /* computed branch */
```

If the conditional branch is not-taken, then the computed branch's target is the address of function1. If the conditional branch is taken, then the computed branch's target is the address of function2. In this case, the global branch history provides enough additional information to determine the target of the computed branch. Other possibilities include the incorporation of the history of register values, or even predicted register values, in the computation of the branch target prediction.

10. The shortcoming of a two-cycle branch prediction loop is that it will increase the penalty of a branch misprediction by one cycle. Normally, as soon as a branch prediction is known, the branch predictor is fed the new PC and a next PC can be predicted for the next cycle. The two-cycle branch prediction loop would be reset by the branch misprediction, but two cycles would have to pass before the next PC is available (as opposed to only a single cycle in the case of the normal single-cycle predictor loop).

The key advantage of a two-cycle branch prediction loop is that the longer prediction latency may enable a larger (and therefore more accurate) branch predictor to be implemented. This predictor may be slightly more complicated since it must provide *two* predictions every other cycle (either by simultaneously predicting the next PC and the next-next PC, or perhaps through pipelining). Similar to the overriding branch predictor configuration, it may be better to provide a more accurate prediction with occasional pipeline bubbles than to provide a wrong prediction really quickly.

11. The next-trace predictor typically predicts the next trace address is a fashion similar to a traditional BTB. Each return instruction that occurs in a trace will likely result in a wrong next-trace prediction. This may not actually affect the *hit rate* of the trace cache because the correct next trace may actually be cached. In this situation, after the original trace has been decoded, the processor may notice the presence of a return instruction, and then consult the return address stack predictor for a better prediction. The processor can then ignore/flush the originally predicted next-trace and try to fetch a trace that starts at the address specified by the RAS. Whether this trace hits or misses simply depends on whether the trace has been seen before and whether or not the trace has been evicted. Unless the next-trace predictor is adapted to incorporate predictions from the RAS, the next-trace prediction rate will be decreased by the presence of return instructions that jump to different call sites.

12. Constructing traces in the front-end of the processor can directly affect the branch misprediction penalty, because after a branch misprediction, a new trace must be fetched which may not even be present. Thus, the *latency* of the trace builder may greatly affect overall performance due to an increased misprediction recovery time. An alternative is to allow some sort of bypassing of the trace cache construction (i.e. fetch instructions directly from a conventional instruction cache). Back-end trace construction occurs after instruction retirement, and therefore the latency is not as critical. The processor can spend many cycles slowly constructing traces, and the only real penalty is an opportunity loss; that is, before the trace has been constructed, the front-end would not have been able to deliver as great of an instruction fetch bandwidth because the trace was not yet available and the instructions would instead have to come from the instruction cache.

An advantage of front-end trace construction is that current branch predictions can be incorporated. The back-end basically has to rely on building traces based on the most recently observed path (which may not be the same the next time around). The next time the starting block of this trace is reached, the program control flow may take a different path. The front-end constructor can incorporate the current branch predictions that may make use of better information such as branch history. This has an impact on the effective prediction rates of intra-trace branches.

13. One of the major difficulties of a deep layering of overriding predictors is design complexity. Even with a single overriding predictor, the processor must be able to intercept instructions that have been fetched with the quick-and-dirty prediction when the overriding predictor predicts that a different path should be taken. Properly tracking and flushing out these instructions is somewhat tricky even for a simple overriding organization. In a multi-layered overriding configuration, there are now many more "intercept points", and it is possible that multiple overrides that correspond to different branches arrive at the same time. The processor must track which of these correspond to

the oldest branch and then resteer the fetch engine accordingly. Another difficulty is that the last predictor in a ten-layered overriding prediction scheme would likely have a latency that is 10 cycles or more. If fetched instructions have already been renamed by the time this last prediction is made available, then resteering the frontend is much more complicated because the register renaming table must also be repaired.

14. [No Answer]

15. [No Answer]

16. Up to a certain size, a large window may provide more parallelism which in turn may result in the faster detection of a mispredicted branch. Sometimes loads that are executed on a wrong-path may cause cache misses that bring in data which will later be useful; this wrong-path prefetching effect may be beneficial in some situations, and a larger window can potentially uncover more of these wrong-path prefetches. A branch predictor with a relatively poor misprediction rate may still benefit from a large window depending on how the mispredictions are clustered. For example, if most of the mispredictions are tightly clustered (they all occur near each other), that means that there will still be long stretches where there are no or very few mispredictions. In these regions, a large window can still be filled with useful work which may result in a performance benefit.

A large window may go largely unused if the branch predictor is not accurate enough. If a branch misprediction occurs once every 40 instructions, and the instruction window supports 100 instructions, then most of the time 60% of the instruction window will be idle. Instead of building such a large window, it may have been possible to build a smaller one that achieved a higher clock frequency that could have resulted in greater performance. Besides potential lost performance opportunities, an over-designed instruction window wastes a lot of power. For example, power may still be burned due to driving the clock signal to all of the unused entries, and transistor leakage also consumes power.