Control Hazards

This is lecture from my old class notes; it is more in line with my research point of view and less consistent with your text, but it is a good alternate introduction to branch prediction

Control Hazards

Instructions that disrupt the sequential flow of control present problems for pipelines. The effects of these instructions can't be exactly determined until late in the pipeline, so instruction fetch can't continue unless we do something special. The following types of instructions can introduce control hazards:

- Unconditional branches.
- Conditional branches.
- Indirect branches.
- Procedure calls.
- Procedure returns.

Let's see how control hazards are manifested with an example:

```
        ld      r2, 0(r4)       // r2 := memory at r4
        ld      r3, 4(r4)       // r3 := memory at r4+4
        sub     r1, r2, r3      // r1 := r2 - r3
        beqz    r1, L1          // if r1 is not 0, goto L1
        ldi     r1, 1           // r1 := 1
L1:     not     r1, r1          // r1 := not r1
        st      r1, 0(r5)       // store r1 to memory at r5
```

This code has the effect of comparing two memory locations and storing the result of that comparison (1 for equal, 0 for not equal) to another location. If the `beqz` branch is taken, then a 1 is stored; otherwise, a 0 is stored. Let's assume we have a five-stage pipeline as in the last class.

```
   Cycles----->
 ________________________                               Instructions
|_IF_|_ID_|_EX_|_MM_|_WB_|_____                         ld      r2, 0(r4)
     |_IF_|_ID_|_EX_|_MM_|_WB_|_________                ld      r3, 4(r4)
          |_IF_|_EX |_ID_|_EX_|_MM_|_WB_|____           sub     r1, r2, r3
               |_IF_|_EX |_ID_|_EX_|_MM_|_WB_|____      beqz    r1, L1
                    |_??_|_?? |_??_|_??_|_??_|_??_|____ ???
                         |_??_|_?? |_??_|_??_|_??_|_??_|???
```

beqz and `sub` instructions. Since the `beqz` instruction is a control instruction, there are two problems here:

- When the `beqz` instruction is in the decode stage, the `sub` instruction is in the execute stage. The branch can't read the output of the `sub` until it has been written to the register file; if it reads it early, it will read the wrong value. How do we fix this?
- More importantly, one cycle after the `beqz` instruction is fetched, what instruction should be fetched next? We don't know, because that depends on the outcome of the branch instruction. At this point, we don't even know that the branch instruction depends on the previous instruction, since it hasn't been decoded yet, so bypass can't help us. Even if we did know the condition, we still don't know where to fetch from if the branch is taken because the effective address computation for branches doesn't happen until the EX stage.

This little example illustrates the problem with branches in pipelines. Two unresolved issues in the pipeline cause control hazards: branch outcome (e.g., taken or not taken) and branch target (i.e. computing the next value for the PC register).

Solutions for Control Hazards

The following are solutions that have been proposed for mitigating aspects of control hazards:

- Pipeline stall cycles. Freeze the pipeline until the branch outcome and target are known, then proceed with fetch. Thus, every branch instruction incurs a penalty equal to the number of stall cycles. This solution is unsatisfactory if the instruction mix contains many branch instructions, and/or the pipeline is very deep.
- Branch delay slots. The ISA is constructed such that one or more instructions sequentially following a conditional branch instruction are executed whether or not the branch is taken. The compiler or assembly language writer must fill these *branch delay slots* with useful instructions or NOPs (no-operation opcodes). This solution doesn't extend well to deeper pipelines, and becomes architectural baggage that the ISA must carry into future implementations.
- Branch prediction. The outcome and target of conditional branches are predicted using some heuristic. Instructions are speculatively fetched and executed down the predicted path, but results are not written back to the register file until the branch is executed and the prediction is verified. When a branch is predicted, the processor enters a *speculative mode* in which results are written to another register file that mirrors the architected register file. Another pipeline stage called the *commit* stage is introduced to handle writing verified speculatively obtained results back into the "real" register file. Branch predictors can't be 100% accurate, so there is still a penalty for branches that is based on the branch misprediction rate.
- Indirect branch prediction. Branches such as virtual method calls, computed `goto`s and jumps through tables of pointers can be predicted using various techniques.
- Return address stack (RAS). Procedure returns are a form of indirect jump that can be perfectly predicted with a stack as long as the call depth doesn't exceed the stack depth. Return addresses are pushed onto the stack at a call and popped off at a return.

Branch Prediction Strategies

Let's take a look at some strategies that have been proposed for conditional branch prediction. There are two issues: outcome prediction and target prediction. The most important aspect of a branch prediction strategy is accuracy. Higher levels of sophistication in branch prediction strategies lead to higher accuracy.

Branch Direction Prediction

Here are some ideas for branch outcome prediction (or "branch direction prediction"), the problem of predicting whether a conditional branch will be taken or not taken.

- Static branch prediction. Static branch prediction means that the prediction is the same every time a branch is fetched. There are three levels of static branch prediction strategies:
  - Always taken/Always not taken. Every branch is predicted to go a particular way. Implementing an "always not taken" strategy is the simplest thing to do, since it eliminates the need for target prediction. Instructions are fetched and speculatively executed down the not-taken path. The accuracy of this technique is low; however, it can be improved with compiler cooperation.
  - Heuristics depending on the instruction. For instance, a popular strategy is "backward taken/forward not taken." Backward branches are assumed to be loop back edges that are frequently taken, while forward branches are assumed to be frequently not taken. This strategy is more accurate.
  - Hint bits in the ISA. Some ISAs allow conditional branch instructions to include hint bits that let the microarchitecture know which direction the branch is likely to be taken. These hints bits can be set through smart compiler heuristics, or through training in a profiling run of the program. This technique achieves about 90% accuracy in the SPEC CPU integer benchmarks.
- Dynamic branch prediction. Every time a branch is executed, a machine algorithm gives a prediction for the branch. Dynamic branch prediction can predict branches that static branch prediction can't, for instance, loop back edges with fixed trip counts. Let's look at some increasingly complex dynamic branch predictors to see how they achieve high accuracy:

- Bimodal (or "Smith") predictors. A table of bits, indexed by branch PC, is kept. Each time a branch executes, the corresponding bit is set to 1 if the branch was taken, 0 otherwise. The next time the branch needs to be predicted, the corresponding bit is taken as the prediction. This is like the hint bits in the ISA for static prediction, but the hint bits are learned on-line, without a separate profiling run. Note that this system will predict highly biased branches well, but for instance a loop back edge with a low trip count will be mispredicted twice: once at the loop exit, and once the very next time the back edge is encountered.
- Bimodal prediction with hyseresis. Instead of keeping a table of bits, we keep a table of two-bit counters that saturate at 3 and 0. When a branch is taken, the corredponding counter is incremented. When the branch is not taken, the counter is decremented. When a branch needs to be predicted, the high bit of the corredponding counter is taken as the prediction. The low bit provides *hysteresis*, i.e., takes into account the history of the branch beyond just the last time the branch was executed. So a loop back edge will incur only one misprediction at a loop exit.
- Two-level adaptive branch prediction. A history shift register is kept, either for all branches (global) or for each branch (local). The length of this register is called the *history length*. As branches are executed, their outcomes (1 for taken, 0 for not taken) are shifted into this register. The register is used to index a table of two-bit saturating counters to do prediction with hysteresis. Once this scheme has gone through a sufficient (small) training period, it can perfectly predict loop back edges with small fixed trip counts. This scheme is highly susceptible to *aliasing* in which two unrelated branches contend for the same counter, resulting in reduced accuracy. Many ways to address aliasing have been proposed. Two-level branch prediction is present in all modern general-purpose microprocessors. How many table entrie are required for a history length of *N*?
- Hybrid branch prediction. Two or more predictors are combined to provide a more accurate estimate. For instance, local and global two-level predictors can be combined, or two global predictors with different history lengths.
- Other predictors. Neural predictors, predictors based on decision trees, or finite state machines learned off-line, combinations of static and dynamic prediction, etc.

Let's look at an example of how two-level prediction works. We'll assume a history length of 4. Consider the following C code:

```
int A[100][3];
int main () {
        int     i, j;

        for (i=0; i<100; i++) for (j=0; j<3; j++) A[i][j] = 0;
}

The history of the inner loop looks like this:

110110110110110110...


Let's assume the 4-bit history register for that branch is initialized to
all bits 0.  As we go through the loops, the history would shift through
these combinations:

history         outcome
0000            1
0001            1
0011            0
0110            1
1101            1
1011            0
0110            1
1101            1
1011            0
0110            1
1101            1
```

```
1011            0
...


So the table of 16 counters for that branch would look like this:
0000    01      // only on warm-up
0001    01      // only on warm-up
0010    00      // never happens
0011    00      // never happens
0100    00      // never happens
0101    00      // never happens
0110    11      // predict the back edge taken
0111    00      // never happens
1000    00      // never happens
1001    00      // never happens
1010    00      // never happens
1011    00      // predict the back edge not taken
1100    00      // never happens
1101    11      // predict the back edge taken
1110    00      // never happens
1111    00      // never happens
```

What about the outer loop? To learn that loop's behavior perfectly, we would need a history length of at least 100. That would be impossible with two-level prediction. However, it doesn't hurt us much if we mispredict that branch once in the entire run of the function; we will predict it correctly 99% of the time.

Branch Target Prediction

Predicting branch targets can be just as important as predicting branch directions. For jumps and taken branches, or any other time when the flow of control is no longer sequential, the address of the next instruction must be given to the fetch stage as quickly as possible. Some techniques for predicting targets:

- Return address stack (we have seen).
- Branch target buffer (BTB). Let's think of conditional branches only for now. A BTB keeps a table of branch targets indexed by the branch PC. When a branch is taken, the computed target is associated with the branch PC in the BTB. When the branch needs to be predicted, the predicted target is taken from the BTB. Note that the BTB only stores taken targets; non-taken branches simply proceed to the next instruction, like any other non-control instruction.

  The table keeps *tags*, much like a cache, to ensure that the branch PC and target address are not just coincidentally aliased to one another. The tags may be some or all of the bits of the branch PC. BTBs can also be set-associative, that is, the BTB can be a table of sets of entries, to get around aliasing problems. Note that this scheme can be directly extended to non-conditional branches that have only one target.

- Indirect branch prediction. Predicting the targets of indirect branches such as virtual method calls can be done using a mechanism similar to two-level adaptive branch prediction. Instead of indexing a table of two-bit counters, we can index a table of BTB entries with a combination of history and address. By storing a possibly different address with each history, we can store multiple targets and get good correlation between history and target. Note that BTBs that use full branch addresses as tags don't need to have their targets verified, while indirect branch predictors do.