Branch Prediction

Lecture #3: Wednesday, 5 April 2000
Lecturer: Mattan Erez
Scribe: Mahesh Madhav

1 How to Read a Paper

Professor Dally was getting requests on tips for reading papers, so he went over his method in class:

1. Before diving in, think about what your goals are, and what you want to get out of the paper. You need to approach the reading with an aim to extract info; you may want to target specific areas of the paper in case you are looking for something particular. Keeping these objectives in hand will help you while reading.

2. Read the abstract first. This should be an advertisement for the paper.

3. Read the conclusions. Figure out what the authors accomplished, so you know what their goals were.

4. Find the "paper overview" section in the introduction and read it if you need to. If you don't know about a certain topic, you may find background material here.

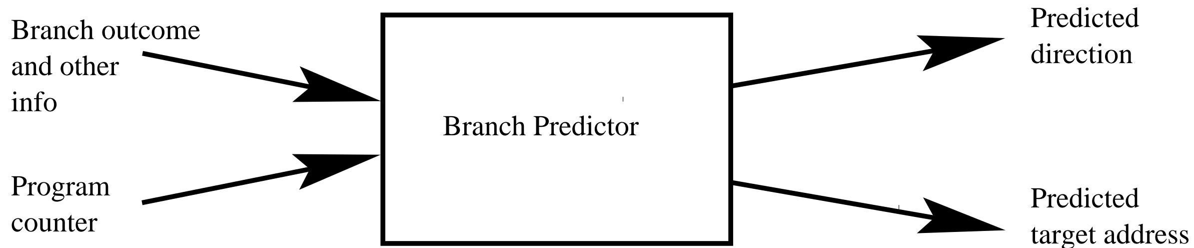
5. Make a quick first pass through the paper, so you understand their presentation of the topics. Figure out how the equations fit into the discussion, but don't try to understand the math. On the second pass, read the paper more carefully.

6. Highlight important parts of a paragraph so you don't have to reread the entire paragraph each time. This helps in class discussions as well.

7. Write down an outline as you go- this will help you remember what you have read. You can refer back to this outline as you go through the paper.

8. Look up key references if you are looking for a particular topic. This paper may not have what you are looking for, but it may build on top of a topic you need to research. In the library, you can find guides that list forward references for each paper, which may also be useful in your quest.

2 Introduction



A branch predictor can be thought of as a black box, as shown above. We feed it info about the instruction pointer (PC) and outcome of that branch, and it uses it to speculate a course of action.

- What is branch prediction and what are we actually predicting?

  We are predicting where the next instruction is in the instruction stream. We speculate two items: the direction of the branch (taken/not-taken) and the branch target (where the branch is going).

- Why do we need to compute the target (wouldn't the branch target be the same everytime)?

  The target address changes if a branch is computed, or if it is an offset branch. A fetch may block, and may need to predict the target before going to the decode stage. A large latency will need to be filled with bubbles, which is not resourceful.

- Why do we predict branch direction? Are there other solutions?

  One solution is to explicitly execute both instruction paths when we reach a branch. This has the problem that the number of paths grow exponentially. There are lots of branches in a program - a one out of five instructions is a typical cited number.

  The Pentium Pro has about 50 instructions in flight at one time; applying the above ratio here gives us 10-20 branches in that window. This shows how impractical it is to use parallel computation of both paths. It is very inefficient if we don't use branch prediction.

  Side note: you can always write code to fool any branch predictor. One way is to build a model of the predictor in software, and use its own algorithm to guess incorrectly each time.

3 Lee/Smith Paper [1]

3.1 Branch Prediction Strategies

We discussed some of the existing approaches to the branch problem.

Loop buffers A small, very high speed buffer maintained by the instruction fetch stage of the pipeline. Essentially a very small I-Cache. The original i386 had 32 bytes of loop buffer on a block boundary, and programmers tried to hack their code to fit entire loops in there very tightly for faster execution. Does this solve the problem of branch prediction? Not really; it just alleviates the high memory latency for misses in the instruction cache. (May break for unrolling loops.)

Prefetch branch target Method of "eager fetch", doesn't solve any problem, just makes misses incur less delay.

Prepare to branch Software gives hints to the hardware about what the branch target will be. It saves us the target prediction since it has already been written into one of the target registers.

Branch delay slot A slot for executing an independent instruction while waiting for the branch to resolve. If you have 5 or 6 branches waiting to resolve, you may not have 30 independent instructions waiting to be executed. Using this feature breaks binary compatibility across a processor family, since branches may take a different number of clock cycles to resolve.

Branch target buffer A small cache memory associated with the instruction fetch stage of the pipeline. It contains the branch instruction address, prediction stats, and the branch target address associated with them. Its structure is similar to a TLB. The P6 has a BTB of size 512.

3.2 Methodology

- As in all papers, the authors select a set of benchmarks to test performance.
- Though a bit hard to decipher, Figure 5 shows that a small number of instructions are executed a large percentage of the time.
- Figure 6 is interesting as well. Certain patterns can be seen easily by a branch predictor. Line 5, for example, is a for loop which loops 14 times then breaks out. Line 8, however, is a pattern that any human can detect, but branch hardware would likely fail on.

3.3 Prediction Mechanisms

Static Prediction Base all prediction on what the compiler gives you for input. These tables do not change at run time. Many compilers also let you feed profile data back into your program so it runs faster by making informed decisions on how to predict branches, how to use the cache, etc. The Sun compilers in Sweet Hall have this capability. If you haven't tried it out, here is how you do it:

Compile the program by informing it to generate profile data

```
$ /opt/SUNWspro/SC5.0/bin/CC -p -xprofile=collect:a.out main.C
```

Run the program a few times to "train" it.

```
$ time ./a.out
```

Recompile it using the data we generated.

```
$ /opt/SUNWspro/SC5.0/bin/CC -xO5 -xprofile=use:a.out main.C
```

Execution should now be faster. (Compare the `time` outputs)

```
$ time ./a.out
```

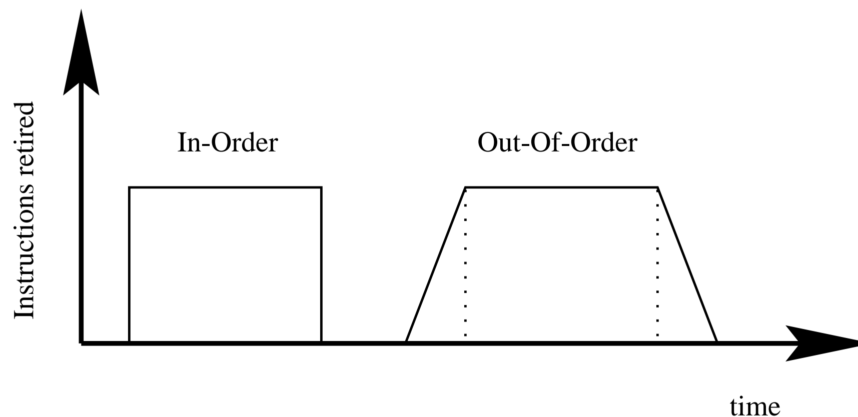
Dynamic Prediction In this case, executing the program "trains" the branch prediction hardware on the fly. Branch history may be kept to determine the best course of action to take when that branch is reached again.

Prediction based on Opcode The opcode of the branch instruction may affect its behavior. This method has almost disappeared since it isn't that effective. Professor Dally mentioned that the Alpha implements this, since all their branches are the same instruction but with different opcodes.

Return Address Stack On every `JTS` you push the calling address onto the RAS, and on every `RTS` you predict the target from the RAS. All current high-performance microprocessors have RAS's now. Usually they are 8 or 16 levels deep.

Early Resolution Dedicated logic for resolving branches out of order.

The authors tried to correlate prediction performance with actual performance. In their simulation, they had penalties measured in cycles. This is not accurate enough due to differences in the way Out-Of-Order and SuperScalar processors work compared to In-Order.



From the figure above, In-Order processors stop the instruction flow when a branch needs to be resolved, whereas OOO processors can still finish executing a few paths while waiting for the branch to resolve. There is more of a rampup-rampdown seen in OOO, while In-Order is all-or-nothing.

Quandary: All these methods are based on history– correlated with all other branches. How do you base your prediction on the outcome of that prediction? (your history of other predictions?) One can try to update the tables speculatively, since they are easy to flush anyway.

4 McFarling Technical Note [2]

Almost all branch prediction papers written after 1994 reference this one.

Yeh/Patt name	McFarling name	description
per address (PA)	local	Keep info for each specific branch to find patterns Table of 2-bit counters indexed by program counter to get prediction.
GA	global	Take advantage of other recent branches to make prediction. All branches update the same BHR, and we ignore the program counter

In each of these ideas, we index into a table of 2-bit saturating counters that dictate if the branch should be taken or not. If we take a branch we increment the counter, if we don't take the branch, the counter is decremented. The counters are initialized at 10 or 01, states which are more neutral than 00..

state	interpretation
11	strongly taken
10	weakly taken
01	weakly not-taken
00	strongly not-taken

Let's use 3 bits of history to index the counters which are initialize to 01. If we encounter have a pattern that looks like

$$101\underline{1}\ldots101\underline{0}\ldots101\underline{1}\ldots101\underline{1}\ldots$$

we use 101 to index into the counter table, and the next time we encounter a 101, we should predict taken since the counter state is at 11.

McFarling introduces a schemes he calls gshare. It is based on a scheme called gselect which adds some bits of the instruction pointer to the index before doing a table lookup.

Gshare goes one step further by using twice as many bits from the IP as gselect. This way, it is unlikely that there will be aliasing in the counter table. Gshare removes most interference between different branch instructions.

He then combines two predictors, a global and a local, and keeps a tally for each branch that records the success rate of both predictions. The final prediction is chosen by taking the prediction most accurate for that branch. This method is implemented in the Alpha 21264.

5 Federovsky et.al. Paper [3]

The authors try reducing the branch prediction problem to a previously solved one (data compression). They are asking, what is the incremental information in each bit (of the predict taken/not-taken bitstream). If there is no incremental information, then that means we can compress the data, and predict the next branch. But if there is new information each time, predicting successive branches is hard.

6 Ending Thoughts

- Nobody really touches upon how branch prediction improves performance of the entire computer system as a whole. Maybe because there are too many variables?

- Nobody evaluates context switching in these papers. Most simulations are assumed to be a single thread running with full context, no switching. How can we speed up context switching by saving the least amount of branch prediction state for each thread?

- Researchers have stopped trying new branch prediction algorithms, and instead try reducing the learning time (compulsory misses) of predictors, and increasing accuracy at the same time. Predictors that need a longer learning time may not be useful for small programs.

- What are good ways to reduce interference and aliasing?

- In the YAGS branch prediction scheme, the warm-up time is very small. Aliasing is also reduced.

- Partial Matching. Keep history, but only use as much history as you have collected for the prediction; don't use the unitialized portions. This is similar to Classless InterDomain Routing (CIDR) in IP routers that forward packets to the most specific outgoing interface they know about.

References

[1] Johnny K. F. Lee, Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer, Vol. 17 Issue 1*, pp 6-21, January 1984.

[2] Scott McFarling, "Combining Branch Predictors", *Technical Note TN-36, Digital Equipment Corporation Western Research Laboratories*, June 1993.

[3] Eitan Federovsky, Meir Feder, Shlomo Weiss, "Branch Prediction Based on Universal Data Compression Algorithms", Proceedings of the *25th International Symposium on Computer Architecture*, June 1998.

[4] Marius Evers, Sanjay J. Patel, Robert S. Chappel, Yale N. Patt, "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work", Proceedings of the *31st International Symposium on Microarchitecture*, December 1998.