A Quantitative Comparison of Static vs Dynamic Branch Prediction Schemes

Javier Lombillo
2016-03-29
CET3126C Lab2
School of Engineering and Technology
Miami-Dade College

Abstract

Three branch prediction schemes were compared against a suite of benchmarks using a superscalar Alpha ISA simulator. A dynamic 2-bit bimodal predictor was shown to outperform two static predictors across all simulated workloads. A possible connection was discovered between a program's percentage of floating-point instructions and its sensitivity to branch prediction performance.

1. Introduction

Historically, the earliest computers used processors with a **single-cycle datapath**, in which every instruction completes in one clock cycle. While simple in both conception and implementation, the flaw with this model is that the minimum clock period depends on the slowest instruction—typically the memory load instruction—which can be significantly slower than the average instruction time. As such, the processor is idle for much of the clock cycle on most instructions, thereby violating the design principle of *making the common case fast*.

To counter this inefficiency, computer architects leveraged the fact that the instruction cycle consists of functionally distinct fetch, decode, and execute stages, each with independent hardware access requirements. For example, in the fetch stage the processor requires access to the program counter (PC) and an ALU, while in the decode stage it requires access to the register file. Since neither stage uses the hardware of the other, there is no reason that the *next* instruction cannot be fetched once the current instruction is in the decode stage.

Thus, by partitioning the instruction cycle into some small number of functional steps—an **instruction pipeline**—multiple instructions can be executing concurrently in assembly line-like fashion. Furthermore, the minimum clock period is no longer the length of the longest instruction, rather it is the length of the longest pipeline stage. This combination of instruction-level parallelism and increased processor utilization greatly improves instruction throughput, at the cost of datapath complexity.

However, a critical issue with any implementation of parallelism is *correctness*: Is the result of a paralleled operation guaranteed to be the same as the result of the equivalent serial operation? In the case of instruction pipelines, the circumstances under which correctness could fail are called **pipeline hazards**. This paper explores particular schemes that serve to handle a particular type of hazard known as *control hazards*.

2. Background

A branch instruction is a conditional jump. That is, depending on some condition to be evaluated, the result of a branch instruction will be either a jump to an out-of-sequence instruction, or a NOP, causing the next instruction in the sequence to be fetched. To illustrate the effect of a control hazard, consider the consequence of a branch instruction as it progresses through a pipeline. While the branch is being fetched, the previous instruction resides in the decode stage. On the next clock cycle, the branch enters the decode stage; at this point, the *next* instruction—the instruction following the branch—must be fetched. However, *which* instruction this is depends on the result of the branch condition, which has yet to be evaluated.

One method of eliminating this hazard is to simply *stall* the pipeline, i.e., add one or more NOP instructions after the branch to delay the next instruction until the condition in the branch has had enough time to be evaluated. While effective, this solution is inefficient: real-world programs typically branch every few lines of code; in the vernacular of the field, *basic blocks* tend to be small. A better solution uses **branch prediction**, wherein the branch target is *assumed* before the condition is evaluated. When the prediction is correct, the pipeline continues at its normal rate; otherwise, the state of the system must be reverted and the pipeline restarted.

Research and real-world implementations have shown that—despite the slight overhead penalty incurred by the pipeline itself, and the significant penalty from a misprediction—branch prediction results in a large overall performance gain. Prediction schemes abound, including sophisticated approaches that achieve success rates of better than 90% [1].

3. Experimental Methodology

Three branch prediction schemes were compared against a 'perfect' predictor on four representative workloads. Predictor performance was measured by comparing the cycles per instruction (CPI) ratio across the group, with the 'perfect' predictor serving as the baseline.

All simulations were performed in a Linux environment (kernel 3.2.0-4-amd64, gcc 4.7.2-5) running on a XEN virtual machine.

3.1. Simulation Framework

Simulations were performed using the `sim-outorder` program from SimpleScalar v3.0, implementing the Alpha ISA. As stated

in its output, `sim-outorder` "implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support."

The prediction schemes tested were as follows:

- *taken* always take the branch jump
- *not-taken* never take the branch jump
- *bimod* a 2-bit dynamic prediction scheme with history (table size was 2048)
- *perfect* an idealized predictor that is never wrong

Except for the choice of branch predictor, all configuration values were left at defaults. Total simulation run time, as reported by `time(1)`, was:

```
real    48m25.508s
user    48m10.641s
sys     0m10.821s
```

3.2. Simulated Workloads

Four workloads were tested using SPEC benchmarks, each listed below with the percentage of conditional branches in its respective instruction mix:

- gcc (13.33%)
- Anagram (10.32%)
- Compress95 (5.70%)
- Go (10.96%)

4. Results

Post-simulation data was provided by `sim-outorder`. CPI for each benchmark are shown in Table 2 (lower numbers better). In order to place CPI in proper context, each predictor's total number of cycles used for the benchmarks are shown in Table 1.

	perfect	bimod	taken	not-taken
Anagram	10.83E6	11.78E6	24.57E6	24.76E6
Compress95	47,677	48,921	68,001	68,048
gcc	228.7E6	273.3E6	428.2E6	436.9E6
Go	309.8E6	413.2E6	573.9E6	580.0E6

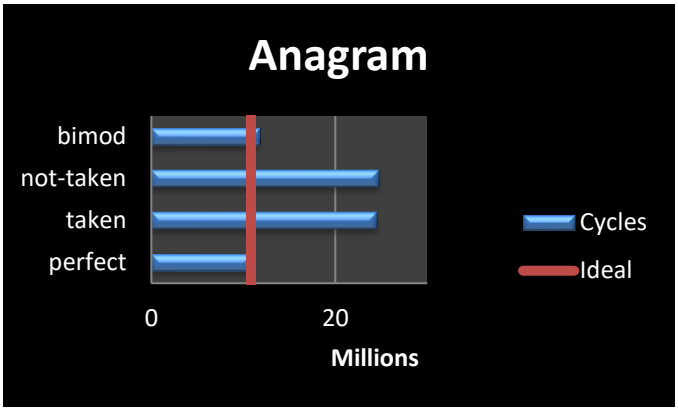
Table 1: Total Cycles

	perfect	bimod	taken	not-taken
Anagram	0.4209	0.4579	0.9550	0.9623
Compress95	0.5374	0.5514	0.7665	0.7670
gcc	0.6778	0.8101	1.2695	1.2950
Go	0.5677	0.7570	1.0514	1.0625

Table 2: CPI

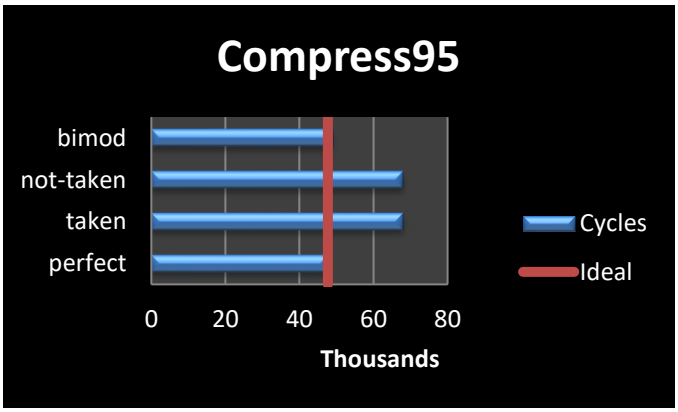
The following sections present graphs of cycle counts for each predictor for the respective benchmark. As a visual aid, a red vertical line indicates the baseline threshold of the 'perfect' predictor. Also included are prediction hit rates for the static and dynamic predictors. [1] The final metric in each category, "CPI spread," represents the percentage change in CPI from the 'perfect' predictor to the worst-performing predictor, with respect to the former, providing a measure of the sensitivity of each benchmark to branch prediction performance. (See Conclusion section for more.)

4.1. Anagram



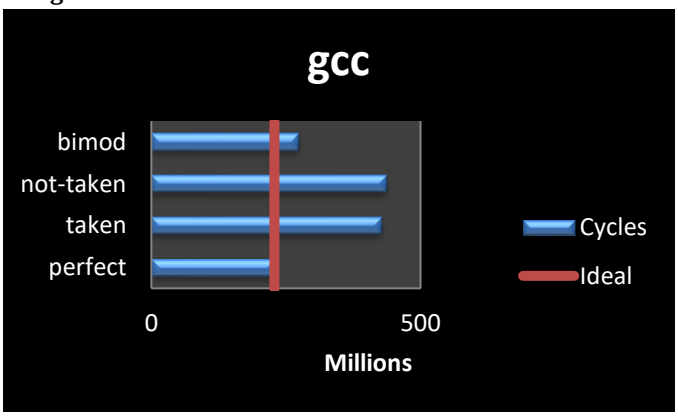

```
Static hit-rate : 0.3139
Bimodal hit-rate: 0.9610
CPI spread      : 129%
```

4.2. Compress95




```
Static hit-rate : 0.0432
Bimodal hit-rate: 0.9725
CPI spread      : 43%
```

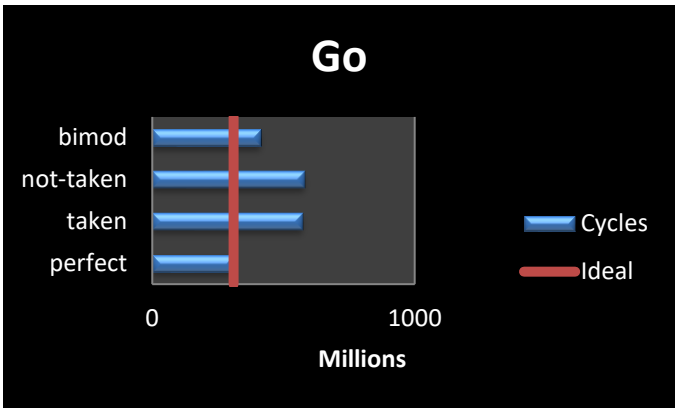
4.3. gcc




```
Static hit-rate : 0.4049
Bimodal hit-rate: 0.8932
CPI spread      : 91%
```

[1] As calculated by sim-outorder in the 'bpred_dir_rate' statistic. In all benchmarks, both the 'taken' and 'not-taken' predictors had the same bpred_dir_rate value, which seems suspicious. The author, however, could not find details on the various bpred hit-miss metrics provided by sim-outorder to make a more informed choice.

4.4. Go




```
Static hit-rate : 0.3782
Bimodal hit-rate: 0.7912
CPI spread      : 87%
```

5. Conclusion

Before discussing the performance of the various branch prediction schemes, it is important to establish some measure of context among the benchmarks. While a given processor will handle any individual branch prediction miss in a consistent and predictable manner, each benchmark presents a unique sequence of instructions, where the aggregate consequences of branch prediction misses are likewise unique. As such, predictor success is relative to the *prediction sensitivity* of its workload. To account for this, the author devised "CPI spread" as a measure of a benchmark's sensitivity to branch prediction performance.

For example, the Compress95 benchmark score showed a 43% CPI spread:

$$\frac{|0.5374 - 0.767|}{0.5374} = 0.427$$

indicating poor prediction sensitivity. Thus, though Compress95 exhibited an enormous difference in its raw dynamic-vs-static predictor hit rate— 97% to 0.03%, respectively—the seemingly tremendous advantage of the bimodal predictor was not reflected in its modest CPI improvement.

One possible explanation for Compress95's low sensitivity is its low usage of conditional branches, which represent only 5.70% of its instruction mix. However, Anagram, with the next lowest branch usage (10.32%), had the *highest* branch sensitivity at 129%. To investigate this further, the table below shows the instruction mix by percentage for each of the benchmarks:

(CPI spread)	**Load**	**Store**	**JMP**	**BRCH**	**Int**	**FP**	**Trap**
AGM (129%)	25.30	9.92	4.48	10.32	44.68	5.29	0.01
GCC (91%)	24.67	11.47	4.12	13.33	46.30	0.11	0.00
GO (87%)	30.62	8.17	2.58	10.96	47.64	0.03	0.00
C95 (43%)	1.74	78.77	0.28	5.70	13.50	0.00	0.02

The benchmarks have been arranged in descending order by CPI spread (shown in parentheses). Both unconditional jumps and floating-point instructions show positive correlation with CPI spread, while memory store instructions exhibit negative correlation. Further research may prove fruitful here.

Finally, the predictors themselves may be discussed. Both the 'taken' and 'not-taken' schemes are static predictors, with the obvious meaning that their "predictions" are *a priori* determined. This simplistic approach is improved by the 'bimod' scheme, a dynamic predictor that uses a small history of previous predictions for its decisions. This is effective because any particular branch may be heavily biased toward one path or the other, e.g., the condition in a long loop. A predictor that is sensitive to this *temporal bimodality* will generally outperform the predetermined guess of a static predictor. This was borne out in the experiment.

6. References

[1] David A. Patterson and John L. Hennessy. 2013. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (5th ed.), p. 284. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA