ACA PROJECT

Name: M V V V S Raju **Roll No: 22cs02003**
T Pranav **22cs02006**

Project : Branch Target Buffer Replacement policy Analysis

Abstract:

Branch prediction plays a crucial role in improving the performance of modern pipelined processors by minimizing control hazards. In this work, we extended the SimpleScalar simulator to evaluate the impact of different Branch Target Buffer (BTB) replacement policies on overall branch prediction accuracy and processor performance. The default Least Recently Used (LRU) policy was augmented with two additional strategies — First-In First-Out (FIFO) and Random replacement — to explore their relative effectiveness under varying conditions. The modified simulator was compiled and executed using both sim-bpred and sim-outorder tools.

Experiments were conducted using four benchmark programs — *anagram*, *go*, *compress*, and *gcc* — across five branch predictors: *taken*, *not-taken*, *bimodal*, *two-level*, and *combined* for each of the replacement policies to measure branch prediction hit rate and Instructions Per Cycle (IPC).

Results show that LRU consistently achieved the highest prediction accuracy and IPC, as it effectively exploits temporal locality by retaining recently used branch targets. FIFO and Random policies, though simpler to implement, exhibited slightly lower performance. Consequently, LRU remains the preferred and widely adopted replacement strategy in most modern processor designs due to its balance between accuracy and complexity.

Introduction:

In modern pipelined processors, branch prediction is essential to maintain high instruction throughput by minimizing stalls caused by control hazards. When a branch instruction is encountered, the processor must predict the next instruction address before the branch outcome is known. To assist in this process, processors use a Branch Target Buffer (BTB) — a cache-like structure that stores the target addresses of previously executed branch instructions. When a branch is fetched, the BTB is consulted to quickly supply its predicted target, thereby allowing the pipeline to continue without interruption.

Since the BTB has limited storage, replacement policies determine which entry should be evicted when new branch information needs to be inserted. The effectiveness of this policy directly influences branch prediction accuracy and, consequently, processor performance. Common replacement strategies include:

- Least Recently Used (LRU): Replaces the entry that has not been accessed for the longest time, leveraging temporal locality.

- First-In First-Out (FIFO): Replaces the oldest entry in insertion order, without considering recent usage.

- Random: Selects a random entry for replacement, offering simplicity but potentially evicting useful entries.

Branch Prediction Overview:

Branch predictors are mechanisms that anticipate the direction of a branch (taken or not taken) before it is resolved, allowing the processor to maintain a continuous flow of instructions. Different prediction strategies vary in complexity and accuracy. In this study, the following predictors were used:

- **Always Taken Predictor:** Assumes every branch will be taken. It performs well for programs where most branches are taken but suffers otherwise.

- **Always Not-Taken Predictor:** Assume branches are never taken. It is simple but often inaccurate in real-world code.

- **Bimodal Predictor:** Uses a single-level table of 2-bit saturating counters indexed by branch address to capture branch behavior history.

- **Two-Level Adaptive Predictor (2-Level):** Utilizes two levels of history — a branch history register and a pattern history table — to detect recurring branch patterns.

- **Combined Predictor (Comb):** Merges the outputs of bimodal and two-level predictors using a meta-predictor to select the more accurate prediction dynamically.

Performance Metrics

Two key performance metrics were used to evaluate the branch prediction mechanisms and replacement policies:

- **Branch Prediction Hit Rate (Accuracy):**
  Indicates how accurately the predictor predicted the branch outcomes.

Hit Rate=(Number of Correct Predictions/Total number of predictions) x 100

- **Instructions Per Cycle (IPC):**
  Represents the average number of instructions executed per clock cycle, serving as a measure of processor throughput.

IPC= Total Instructions Executed / Total Cycles

Results:

Anagram :

Predictor	LRU Hit Rate	FIFO Hit Rate	Random Hit Rate	LRU IPC	FIFO IPC	Random IPC	LRU CPI	FIFO CPI	Random CPI
Taken	0.7664	0.7667	0.7664	0.3654	0.3652	0.3654	2.7369	2.7379	2.7369
Not Taken	0.6066	0.6059	0.6066	0.3614	0.3621	0.3614	2.7667	2.7614	2.7667
Bimodal	0.6678	0.6648	0.6644	0.4057	0.4047	0.4057	2.4650	2.4712	2.4650
2-Level	0.6644	0.6610	0.6644	0.4054	0.4042	0.4054	2.4670	2.4743	2.4670
Combined	0.6701	0.6655	0.6701	0.4054	0.4042	0.4054	2.4670	2.4743	2.4670

Gcc :

Predictor	LRU Hit Rate	FIFO Hit Rate	Random Hit Rate	LRU IPC	FIFO IPC	Random IPC	LRU CPI	FIFO CPI	Random CPI
Taken	0.6004	0.6004	0.6004	0.7878	0.7878	0.7878	1.2694	1.2694	1.2694
Not Taken	0.6357	0.6357	0.6357	0.7722	0.7722	0.7722	1.2949	1.2949	1.2949
Bimodal	0.8919	0.8347	0.8899	1.2431	1.1518	1.2401	0.8044	0.8682	0.8064
2-Level	0.8718	0.8228	0.8704	1.1482	1.0876	1.1465	0.8710	0.9195	0.8722
Combined	0.9119	0.8523	0.9099	1.2693	1.1719	1.2660	0.7878	0.8533	0.7899

Compress:

Predictor	LRU Hit Rate	FIFO Hit Rate	Random Hit Rate	LRU IPC	FIFO IPC	Random IPC	LRU CPI	FIFO CPI	Random CPI
Taken	0.9729	0.9729	0.9729	1.2973	1.2973	1.2973	0.7709	0.7709	0.7709
Not Taken	0.0601	0.0601	0.0601	1.2965	1.2965	1.2965	0.7713	0.7713	0.7713
Bimodal	0.9679	0.9678	0.9678	1.8405	1.8402	1.8402	0.5433	0.5434	0.5434
2-Level	0.9651	0.9651	0.9651	1.8362	1.8362	1.8362	0.5446	0.5446	0.5446
Combined	0.9678	0.9676	0.9676	1.8395	1.8392	1.8392	0.5436	0.5437	0.5437

Go:

Predictor	LRU Hit Rate	FIFO Hit Rate	Random Hit Rate	LRU IPC	FIFO IPC	Random IPC	LRU CPI	FIFO CPI	Random CPI
Taken	0.6221	0.6221	0.6221	0.9512	0.9512	0.9512	1.0513	1.0513	1.0513
Not Taken	0.5689	0.5689	0.5689	0.9412	0.9412	0.9412	1.0624	1.0624	1.0624
Bimodal	0.7894	0.7481	0.7891	1.3218	1.2694	1.3214	0.7565	0.7878	0.7568
2-Level	0.7565	0.7275	0.7561	1.2154	1.1930	1.2153	0.8228	0.8382	0.8229
Combined	0.8083	0.7653	0.8079	1.3401	1.2855	1.3397	0.7462	0.7779	0.7465

Observations:

- LRU gives the highest hit rate and IPC overall among all replacement policies.
- FIFO and Random perform slightly worse but follow similar trends to LRU.
- Static predictors (Taken, Not Taken) show low hit rates and IPC values for all policies.
- Dynamic predictors (Bimodal, 2-Level, Combined) perform much better and show very close results to each other.
- For small benchmarks (like Anagram), all policies perform almost the same because the BTB is not heavily used.
- For large benchmarks (like GCC and Go), LRU performs noticeably better because it keeps recently used branches that are likely to be reused soon.
- Compress shows almost equal performance across all policies, as its branch behavior is very regular and predictable.
- As program size and branch count increase, the difference between replacement policies becomes more visible — LRU adapts best under these conditions.
- Overall, LRU remains the best choice due to its consistent and adaptive performance across all branch predictors and benchmarks.

Conclusion:

From the experimental results obtained using different benchmarks and branch predictors, it is evident that the LRU (Least Recently Used) replacement policy consistently achieves the highest hit rates and IPC values across all test cases. While FIFO and Random show comparable performance for smaller benchmarks, their efficiency decreases as program complexity and branch activity increase.

The superior performance of LRU stems from its ability to retain recently accessed branch targets, which are more likely to be reused soon. This makes it more adaptive and effective in handling real-world program behaviors compared to simpler or probabilistic policies.

Therefore, based on both theoretical reasoning and practical simulation results, LRU stands out as the most efficient and reliable replacement policy, which justifies its widespread use as the default BTB replacement mechanism in modern processor architectures.

Additional works done:

1. Addition of replacement policy in command line:

   Initially unlike in caches , we cannot directly change the replacement policy in the branch target buffer through commands in the terminal. In order to test for newly implemented policies like FIFO / Random we need to change some code in sim-bpred.c . So we can run commands for one policy at a time without changes in code. Thus the addition of a replacement policy in command line.

   This is done with the addition of a new variable bpred_btb_repl which stores the replacement policy and initialises respective policy. All necessary changes in function definitions and calls have been made.

2. A hybrid replacement policy

   From the results we have seen Lru has performed better . But in some cases Random has come close / similar to that of LRU. When there is a scope of utilizing temporal locality in code lru excels and when there is no specific relation between branches Random excels. So we implemented a new policy, a hybrid of LRU and Random , an adaptive replacement policy.

   Here are some results of it run for anagram benchmark:

Predictor \ policy	LRU	Adaptive
Taken	0.7667	0.7667
Not Taken	0.6059	0.6059
Bimod	0.6682	0.6682
2lev	0.6648	0.6648
Comb	0.6704	0.6704

As we can see, the results are the same as LRU. This is because of the benchmark we are running on. All the benchmarks we have are relatively small . Thus we cannot see much improvement. We can try on a large benchmark which we can custom design using a cross compiler . We couldn't complete the cross compiler installation in time and thus were unable to show results for them in the report.

All other necessary code changes required are already made in the files uploaded.