Raghav Gupta
Divija Hasteer
Parshant Juneja

Lab 3 Open Ended Report - 4.1

Branch Predictor Model

Our goal was to improve the branch predictor from the default given bimodal predictor, as well as create a branch predictor model robust enough to compete in the class competition. Originally, we attempted implementing the suggested Alpha21264 branch predictor model; however, we ran into some issues that we will discuss in the following section. It also did not perform consistently better than the default bimodal branch predictor.

One of the branch predictors we ended up implementing is the Gshare/Bimodal branch predictor, an idea we got from office hours and the Combining Branch Predictors paper.

The idea behind the gshare branch predictor is to combine the global history of branch prediction with the program counter of the current branch instruction via an XOR to leverage various valuable pieces of information to index into a counter array, which then tells us whether to take the branch or not (like a bimodal table). This reduces the destructive aliasing you may encounter when solely using the lower bits of the PC to index into the counter array because we pack in some notion of global history into the index. (The branches that align via global history likely have similar behavior.) Adding the PC to the index provides effectiveness by usually indexing into the counter for the relevant branch instruction; the global history alone does not give enough information to do this effectively.

We pair this with a bimodal table and an arbiter, as imaginably, the gshare branch predictor itself is not particularly effective at predicting branches which are independent of global history. The global predictor table may take a little more time to train (fill up with enough data to be useful), and the bimodal table can generally give a solid verdict if there is a strongly biased branch. The bimodal table is indexed into by just the lower bits of the PC of the branch instruction (as discussed above); this can be trained faster but is also prone to destructive aliasing. The arbiter is a table that tracks via a saturating counter which predictor between the bimodal predictor and the gshare predictor is more reliable for a given branch (indexed into with PC), and it directs the speculation to be based on the more reliable predictor.

Below are the results from some of our runs for the same 7 benchmarks used in the directed portion of the lab. We included the default bimodal branch predictor as a baseline.

Raghav Gupta
Divija Hasteer
Parshant Juneja

Bimodal Predictor Results

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.544	0.401	0.540	0.494	0.375	0.545
branch	0.033	0.165	0.096	0.175	0.180	0.075	0.036
total	0.037	0.194	0.110	0.178	0.198	0.131	0.105

Gshare/Bimodal Predictor Results (4096, 2-bit entries)

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.540	0.387	0.538	0.492	0.372	0.542
branch	0.023	0.185	0.111	0.191	0.207	0.082	0.057
total	0.028	0.213	0.123	0.194	0.223	0.138	0.124

Gshare/Bimodal Predictor Results (4096, 3-bit entries)

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.545	0.403	0.538	0.492	0.373	0.544
branch	0.035	0.140	0.089	0.177	0.171	0.085	0.044
total	0.038	0.171	0.104	0.180	0.189	0.139	0.112

Gshare/Bimodal Predictor Results (4096, 4-bit entries)

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.536	0.399	0.533	0.492	0.368	0.537
branch	0.037	0.166	0.086	0.180	0.183	0.096	0.060
total	0.040	0.195	0.100	0.183	0.200	0.148	0.125

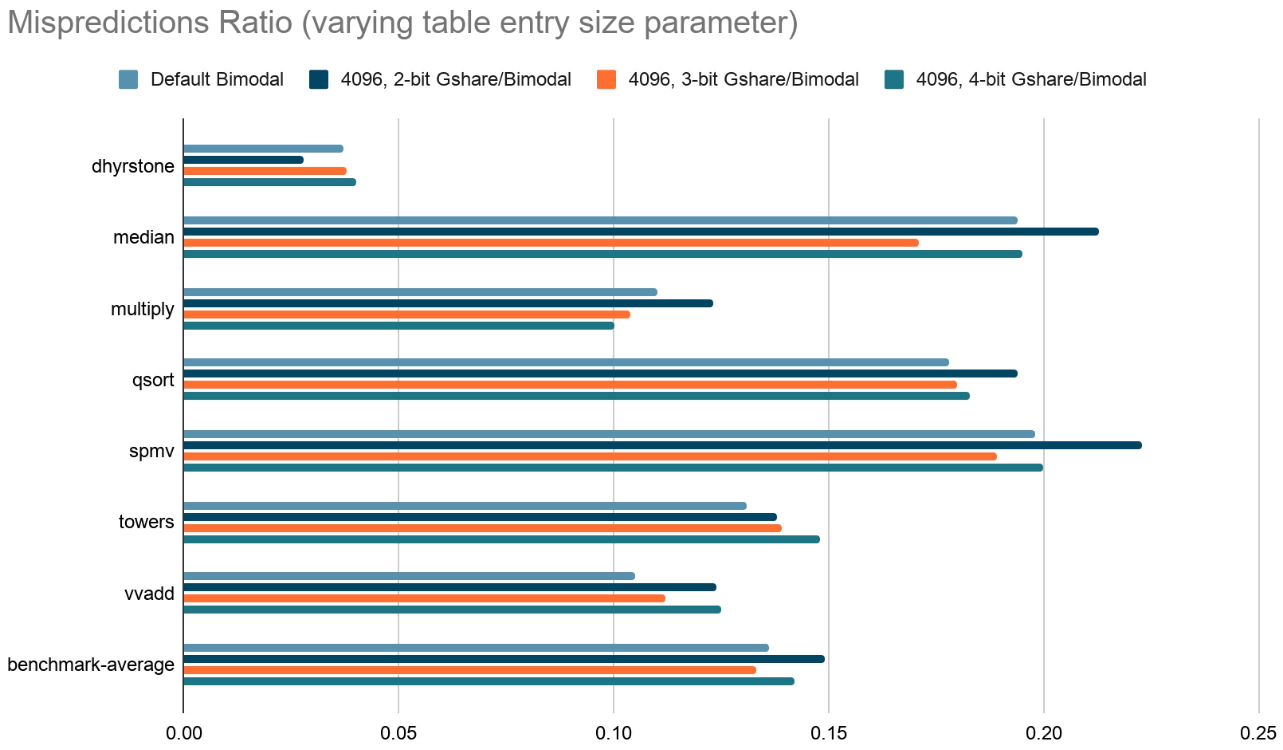
Raghav Gupta
Divija Hasteer
Parshant Juneja

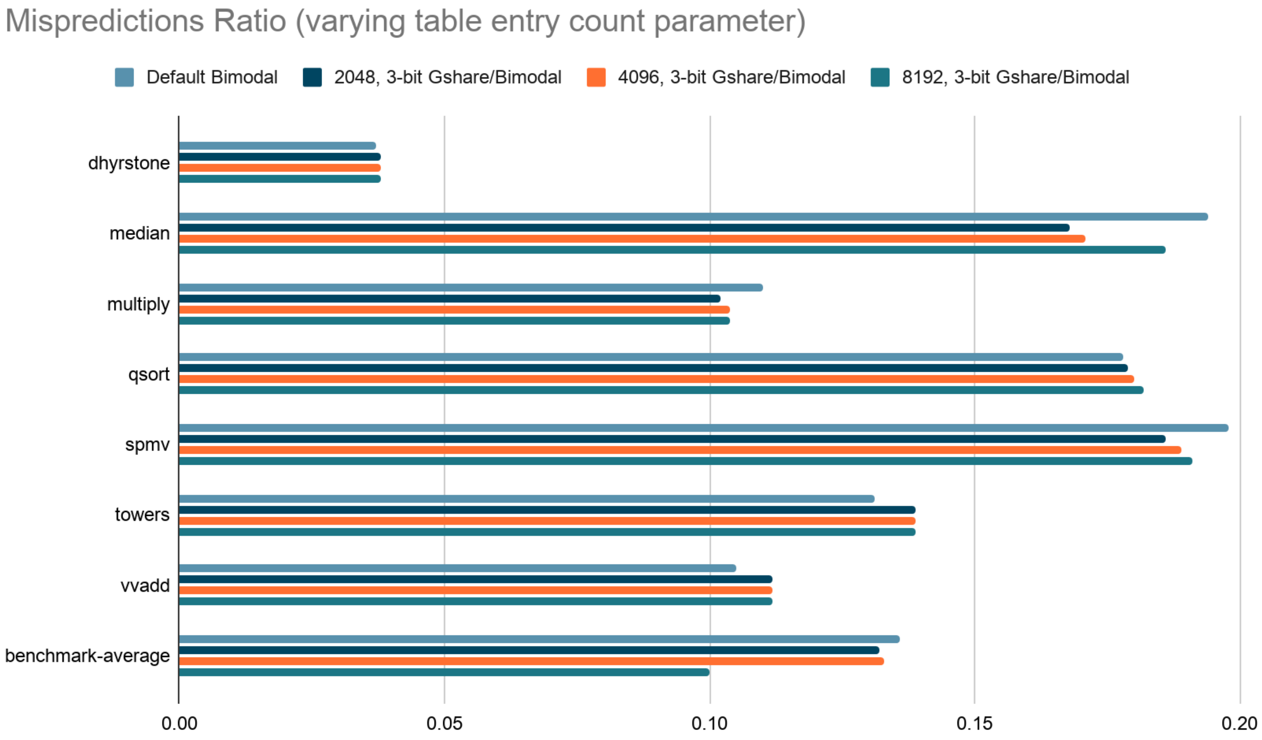
Gshare/Bimodal Predictor Results (2048, 3-bit entries)

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.545	0.403	0.538	0.492	0.373	0.544
branch	0.035	0.137	0.088	0.175	0.167	0.085	0.044
total	0.038	0.168	0.102	0.179	0.186	0.139	0.112

Gshare/Bimodal Predictor Results (8192, 3-bit entries)

Misprediction Ratio	dhrystone	median	multiply	qsort	spmv	towers	vvadd
jalr	0.051	0.544	0.403	0.538	0.492	0.373	0.544
branch	0.035	0.155	0.089	0.179	0.173	0.085	0.044
total	0.038	0.186	0.104	0.182	0.191	0.139	0.112





Our Gshare/Bimodal branch predictor implementation can be found here.

Note to Instructors for Tournament

The gshare/bimodal design that we talk about in this lab report is not the same branch predictor that we are using for the tournament. We have made a lot of adjustments to the gshare/bimodal branch predictor, and we may even be using gskewed.

Raghav Gupta
Divija Hasteer
Parshant Juneja

Alpha21264 Predictor Discussion

Our implementation of the Alpha21264 was not particularly effective, and this was down to issues with updating local history. Local history requires a lot of state, and is quite challenging to maintain with speculative prediction. Given the interface exposed by this lab, we do not know when something gets killed, can result in using the wrong branch entry from the vector to update the tables. However, these difficulties are representative of the issues faced in industry and the corresponding move away from local history tables over the last two decades.

Parshant's suggestion to the staff was to add another parameter to the predict and update functions with some “instruction/cycle ID” that is not subject to aliasing and can help differentiate executed and quashed instructions to enable correct local history updates. This could also be a good learning opportunity for students in future semesters in regards to identifying aliasing.

Raghav Gupta
Divija Hasteer
Parshant Juneja

4.1.6 Analysis Questions

How did you calculate the amount of state your branch predictor has?

For the implementation of the gshare/bimodal predictor shared in the Github gist, we had 3 tables, all the same size with 4096 3-bit entries: the bimodal table, the “global predictor” table, and the arbiter. So our state consists of **3 tables * 2^12 entries/table * 3 bits/entry = (2^12 * 9) bits**. From here, we know that **2^10 bytes = 2^13 bits = 1 KiB**. Thus, **(2^12 * 9) bits * 1 KiB/ 2^13 bits = 4.5 KiB**.

We also use a vector to store branch data between prediction and update, but this data is freed as soon as an update has been made. Thus, this should be a small amount of additional state but we cannot effectively bound it as it is program dependent.

How do certain parameters (e.g., number of entries) impact accuracy?

Originally, we expected that additional entries in our table would help us cut down on aliasing and always benefit us since we were under the assumption that aliasing tends to be destructive. However, from our runs we realized that this may not always be the case; there is such a thing as **constructive aliasing**. We hypothesize that since the gshare branch predictor aligns branches with similar behaviors (by including some notion of global history in the index) the results of different branches that “hash” the same (or have the same index) actually is accurate behavior in reinforcing the other branches that “hash” the same. This allows faster training for the gshare predictor, which is probably the more accurate predictor between gshare and bimodal. By increasing the entries, we reduce all aliasing, including constructive aliasing; this may have a net negative effect on our performance. This is actually what seems to be happening overall as we increase the number of entries from 2048 to 4096 to 8192.

Changing the number of bits in our entries was a bit more ambiguous to us. A smaller counter enables more flexibility in changing between predictions, whereas a larger counter provides more resistance in changing between predictions. Which one do we want? We don’t want to easily shift between predictions because we may often have a one-off branch that improperly indexes into this entry of the counter array (aliasing). At the same time, our program may be actually changing patterns (either because a different branch is using the counter more frequently now or because of the nature of the program), so the counter should be flexible to that possibility. Various benchmarks perform better with different entry sizes. One may have fewest mispredictions with a 2-bit entry (dhrystone); another may have fewest mispredictions with a 3-bit entry (median); and yet another may have fewest mispredictions with a 4-bit entry (multiply).

Right now, it seems that our sweet spot would be a gshare/bimodal predictor with 2048, 3-bit entries.

Raghav Gupta
Divija Hasteer
Parshant Juneja

Which branches or patterns were easier or harder to predict?

Branches with a strong bias are easier to predict with a bimodal predictor. Branches that show correlation in behavior (taken/not taken) with global history are easier to predict with a gshare predictor. Branches that have highly local and unbiased behavior are the hardest ones to predict as neither the gshare predictor nor the bimodal predictor do well on such instructions. Such branches are also most prone to aliasing with other branches across multiple entries of the gshare table.

What kind of application code do you expect your predictor to perform better or worse on?

It's interesting to see how the plain bimodal predictor performs better than the gshare/bimodal predictor for some benchmarks. This is probably an indication that either these benchmarks are too short for the training on the gshare predictor to show its benefits, like towers with 6172 instructions in total and vvadd with 2419 instructions in total. Or, it is an indication that these benchmarks are truly random in their branches that using the pattern of global history to make predictions (via gshare) is actually potentially detrimental, like maybe with the quicksort benchmark. Based on this description, we expect our predictor to perform better on longer benchmarks that follow patterns in their branches (for instance, 10 takens, then a not taken within a loop).

Do you foresee any challenges with the implementation of your predictor algorithm as a hardware block within a superscalar, out-of-order core?

The actual implementation should still be able to work in a superscalar, out-of-order core (no errors). Our predictor is also lightweight in terms of state usage and prediction complexity, and should be able to meet the performance demands of an out-of-order core. But its effectiveness might take a hit.

This is because our implementation relies partially on global history to make the branch prediction. Speculative prediction means that the branches don't see an accurate recent global history for a given branch that is being speculatively predicted. After all, the importance of global history is to rely on a pattern of past execution to help guide the prediction of a branch. But, the "pattern" in a superscalar, out-of-order core becomes a bit distorted.

There is also a significant delay between prediction and update, especially when starting up. In the absence of speculative updates (instead of updates on commit), this leads to large training time.

Raghav Gupta
Divija Hasteer
Parshant Juneja

What changes or alternative approaches would you pursue as future work if more time were available?

Regarding the gshare/bimodal predictor itself, we would explore various hashing combinations for the gshare aspect of the predictor. Here, we would try to understand how much global history itself should play a role in indexing into the entry in our counter arrays. We might also try to enhance the bimodal predictor itself by making it a two-level table. We would also look into using our state more efficiently by having tables of different sizes (like a bimodal table that is half the size of our global predictor table) and determining where the state makes the most impact. The Combining Branch Predictors paper calls this bimodalN/gshareN+1. We were also curious about trying various entry sizes in the table (increasing or reducing the resistance in tipping over to the other prediction in the counter – as mentioned earlier, the best entry size is a bit tricky to predict). We would also take some creative liberties and potentially further combine predictors with our current combined gshare/bimodal predictor (maybe something with a “3-way arbiter”). We would be interested in branching out to other types of predictors, as well, like the set-associative predictor and even the gskewed predictor. After all, for the limited section, we have 64 KiB of state, and gskewed seems like the predictor that best takes advantage of state via its multiple tables (used in majority vote for the prediction) that enable faster training and reduced aliasing.

Raghav Gupta
Divija Hasteer
Parshant Juneja

Sources

"Combining Branch Predictors", McFarling, DEC WRL Technical Note TN-36, 1993.

R. E. Kessler, “The alpha 21264 microprocessor,” IEEE Micro, vol. 19, no. 2, pp. 24– 36, Mar. 1999. doi: 10.1109/40.755465.