E0243: High Performance Computer Architecture

Assignment 1

Group ID - G13

Pranjal Naman
SR - 21246
pranjalnaman@iisc.ac.in

Yash Patel
SR - 22759
yashj@iisc.ac.in

Anjali Chauhan
SR - 22703
anjalic@iisc.ac.in

September 23, 2024

QUESTION 1

Cost-effective Branch Predictors

ChampSim [4] is an open-source trace-based simulator maintained at Texas A&M University, which was initially created to serve as a foundation for microarchitectural competitions like DPC3, DPC2, CRC2, IPC1, etc. Champsim allows users to integrate and simulate their own branch predictors, prefetchers and cache replacement policies. In this assignment, we investigate various branch predictors, and their combinations, and simulate these on three different SPEC2017 [2] benchmarks (namely `644.nab_s`, `648.exchange2_s` and `654.roms_s`). We discuss the individual branch predictors briefly in the following sections.

GShare Branch Predictor

GShare is a type of global branch predictor, i.e. it uses the global branch history to predict a branch. The pattern history table (PHT) is indexed using a hash (generally, XOR) of the program counter (PC) and the global history (GH). The *GShare* branch predictor reduces the effect of *aliasing* present in *bimodal* predictors, and, hence, outperforms the general *bimodal* predictor.

Perceptron Branch Predictor

Proposed by Jimenez and Lin [6], the *Perceptron* branch predictor is a global history register that uses long history lengths to make predictions. It comprises a table of *Perceptron* weights, which are *learnt* (adjusted) as the predictor warms up. It can provide an improvement of up to 10.1% over *GShare*, and 8.2% over *bimodal* for a fixed 4KB hardware budget [6].

TAGE Branch Predictor

The TAGE branch predictor comprises a simple base predictor (T_0), which provides an initial prediction, along with multiple partially tagged predictor components (T_i), which improve upon the predictions of T_0, which is indexed using the program counter (PC). The tagged tables use history lengths varying in a geometric series of the form $L_i = \alpha^{i-1} L_1$, where L_i is the history length to index

in the ith tagged table. While the formula of the series does not matter, the general form of the series is important [10]. We use the previously stated formula with $\alpha = 2$ in all our experiments.

Combining Branch Predictors

Different branch predictors are proposed to address different issues, each with its own advantages and disadvantages. Intelligently combining branch predictors lets us harness the advantages of each while mitigating both their disadvantages to some extent. One of the simplest is combining *bimodal* and *GShare*, which increases the accuracy of the *bimodal* prediction while keeping the overall computational complexity low [9]. Jiménez and Lin [6] show that the complementary strengths of *GShare* and *Perceptron* branch predictors combine well into a hybrid strategy that outperforms them individually. In [5], Jiménez has proposed combining a Multiperspective *Perceptron* branch predictor with TAGE and reports promising accuracies for multiple benchmarks. For the purpose of this assignment, we evaluate two hybrid strategies, namely *GShare-Perceptron* and *TAGE-Perceptron* branch predictors and present a comparative study between the three individual and two combined strategies. We have considered a 1-way 2048-entry branch target buffer for all our experiments.

Q1 (a)

We run the benchmarks on the three aforementioned branch predictions in Champsim with the hardware restriction of 128KB. We use the already available implementations of *GShare* and *Perceptron* predictors in ChampSim. The TAGE predictor is implemented using the open source implementation by [7]. The specifications of each of the branch predictions are listed below -

- **GShare** - We use a 2-bit *bimodal* table of size 512K bits, totalling 128K bytes. The other registers can be considered a part of the overhead budget of 2KB allocated for this question.
- **Perceptron** - We use a 48 bit long history register, with each *Perceptron* weight being 8 bits. The perceptron table consists of 2680 entries, and the update table has 100 entries, each adding up to 200 bits. The update table is used to keep track of the predictions made, in order to reverse them accurately if the speculation turns out wrong. This adds up to a budget of slightly more than 128KB, but within the allowed 2KB buffer.
- **TAGE** - We use a *bimodal* predictor with 64K entries as the base predictor and 4 TAGE tables with 8K entries for the first two, and 16K entries for the other two. Each TAGE entry for the first two tables is 16 bits (11-bit tag, 2-bit usefulness, and 3-bit counter), and 20 bits (15-bit tag) for the other two tables. This adds up to the allowed 128KB budget. We set the minimum history length L_1 at 4, and the α is 2, resulting in the following geometric history lengths - $\{4, 8, 16, 32\}$.

We evaluate the branch predictors for 2 different sets of experiments, one for 250 million (warmup of 50 million) and the other for 500 million (warmup of 200 million) simulation instructions. We report the performance for each of the aforementioned branch predictors on the provided benchmarks. The results for both experiments on all benchmarks are reported in Table 1. We show the MPKI, IPC and prediction accuracy for each branch predictor in Fig 1. The metrics to assess the branch predictors are MPKI (mispredictions per kilo instructions) and prediction accuracy. We list the observations for the 3 benchmark sets below -

- **644.nab_s**

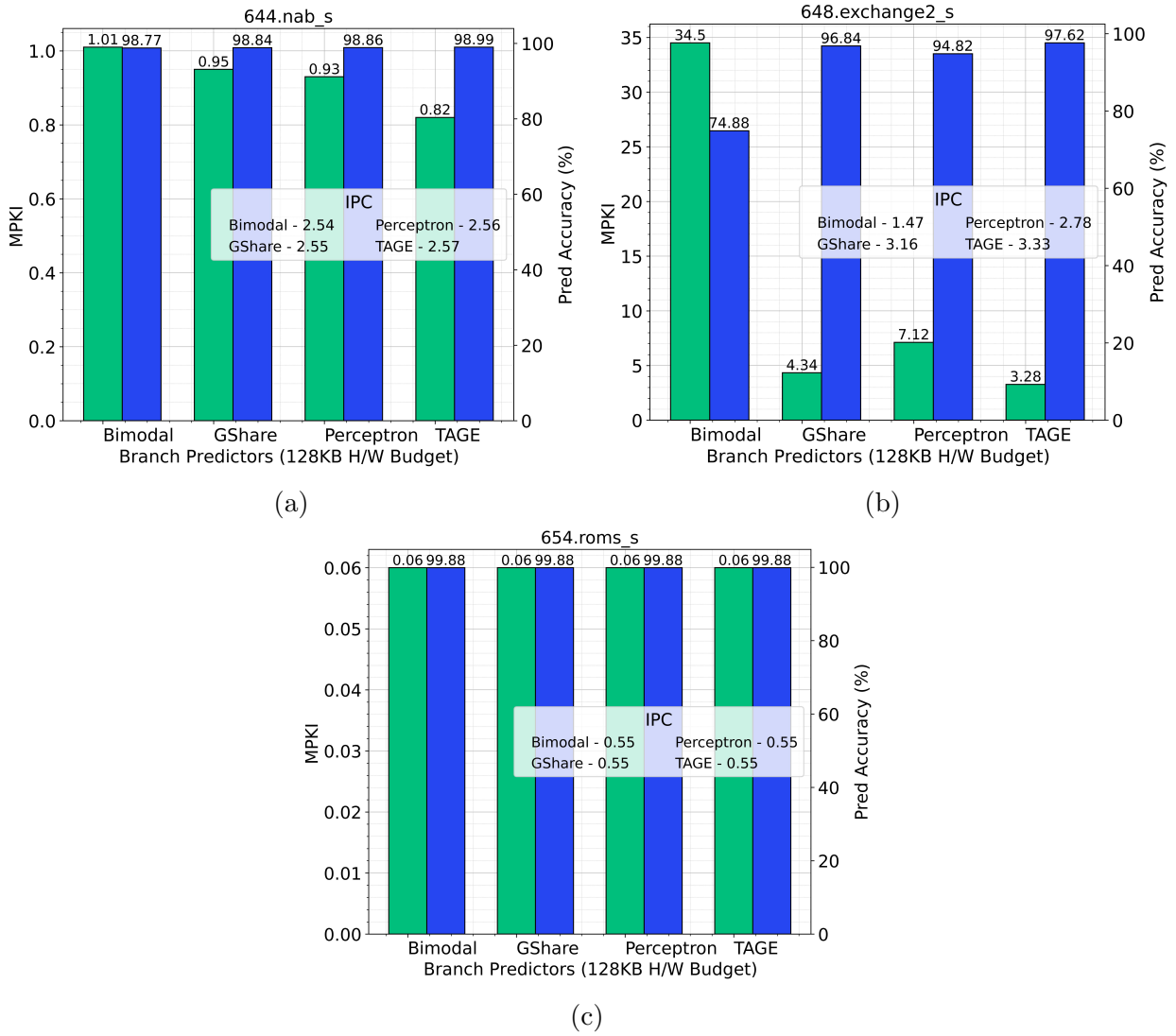


Figure 1: Branch Predictor Results on 3 Benchmarks (500 simulation instructions)

- It is evident from Fig 1a that TAGE performs best for this benchmark, while *bimodal* performs the worst with the highest MPKI.
- The maximum achieved prediction accuracy is 98.99%.
- The misprediction rate for the 3 correlated branch predictors is significantly low when compared to that of the *bimodal* predictor.
- The *bimodal* predictor performs poorly when branches have strong dynamic behaviour.

- **648.exchange2_s**
  - We see a dramatic drop in the MPKI when moving from *bimodal* predictor to other schemes. The reason behind this drop is most likely the presence of a large number of correlated branches. As we incorporate the global history for branch prediction, we notice the MPKI dropping significantly.
  - Since the *bimodal* predictor doesn't consider the correlation between dynamic branches for prediction, it performs poorly in this case.

GShare branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.07984	4.847	0.0633
ipc	3.696	3.096	0.5643
pred acc	99.92	96.51	99.88
ins	250M	250M	250M
cycles	67.64M	80.74M	443M
Perceptron branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.06963	7.707	0.06267
ipc	3.701	2.715	0.5643
pred acc	99.93	94.46	99.88
ins	250M	250M	250M
cycles	67.55M	92.07M	443.02M
TAGE branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.0619	3.687	0.06256
ipc	3.701	3.274	0.5643
pred acc	99.94	97.35	99.88
ins	250M	250M	250M
cycles	67.54M	76.37M	443.02M

(a) Benchmark metrics for 250M instructions

GShare branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.9462	4.343	0.061
ipc	2.548	3.159	0.5518
pred acc	98.84	96.84	99.88
ins	500M	500M	500M
cycles	196.23M	158.28M	906.16M
Perceptron branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.9279	7.115	0.06086
ipc	2.558	2.776	0.552
pred acc	98.86	94.82	99.88
ins	500M	500M	500M
cycles	195.45M	180.15M	905.77M
TAGE branch predictor (128KB)			
Spec. Benchmarks	**644.nab_s**	**648.exchange2_s**	**654.roms_s**
mpki	0.8232	3.275	0.06066
ipc	2.567	3.334	0.5517
pred acc	98.99	97.62	99.88
ins	500M	500M	500M
cycles	194.77M	149.96M	906.26M

(b) Benchmark metrics for 500M instructions

Table 1: Branch Predictor Results on All Benchmarks

  - The MPKI drop can also be attributed to some level to a reduction in aliasing in *GShare* predictor.
  - Since the *Perceptron* predictor is indexed on the program counter, we can attribute the rise in the MPKI from *GShare* to *Perceptron* to aliasing effects.

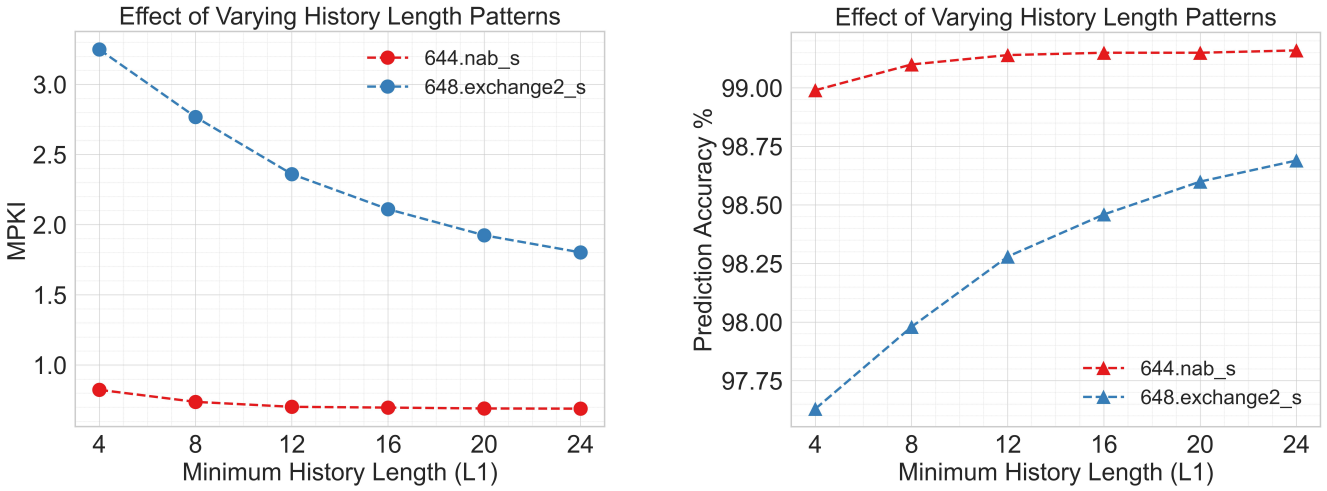
- **654.ros_s**
  - The fact that we do not see any improvement when moving from *bimodal* to *GShare* predictors can be attributed to the branches not exhibiting any or very small correlation.
  - We do not see any improvement for this benchmark across all the branch predictors. The MPKI for all predictors remains the same and is also very low compared to other benchmarks. This is most likely due to the number of branches present in this benchmark being very few. It is also possible that the dynamic branches follow the same specific pattern.

Q1 (b)

We first study the effect of history lengths on the performance (in terms of MPKI) of the TAGE branch predictor. We fix the hardware budget of the predictor at 128KB and increase the minimum history length (L_1) from 4 to 24 in order to see the effect of varying history lengths. The α for these experiments is set as 2 resulting in history length sets of $\{4, 8, 16, 32\}$ for $L_1 = 4$, $\{8, 16, 32, 64\}$ for $L_1 = 8$, and so on. For consistency across experiments, we also fix the size of the base *bimodal* predictor at 16KB. We perform the experiments on 2 benchmark sets to identify a pattern. The results for the same are shown in Fig 2. The key observations are listed here -

- We notice that the MPKI drops sharply for the first increase in the minimum history length, but quickly flattens out as we increase the minimum history length beyond 16. This is most likely because the conditional branches in the given benchmark set are not significantly affected by branches older than history 16.

- The results obtained for the third benchmark set `654.roms_s` do not change when the history length increases. MPKI and prediction accuracy hold steady at 0.0607 and 99.88% for all the history lengths. This again follows from our earlier hypothesis that the branches in `654.roms_s` are few and highly correlated to the point that increasing history length or predictor size or type does not affect the prediction accuracy or MPKI. Hence, we do not represent this case in the plots.



(a) MPKI vs History Len

(b) Pred Accuracy vs History Len

Figure 2: Effect of Varying History Length on TAGE Performance

For the second part of the question, we need to identify the effect different table sizes could have on the performance of TAGE predictors. We again fix the hardware budget at 128KB. The history lengths are also kept the same across experiments for consistency ($L_1 = 4$, $\alpha = 2$, histories $= 4, 8, 16, 32$). The individual table size, however, is dependent on 2 parameters, the number of entries and the size of the tag. We therefore fix the number of entries for the 4 tables at 13K, 13K, 14K and 14K, respectively. We vary the tag length of these tables to change the table sizes. We list down the configurations considered here -

- Config 1: For tables T_1, T_2, T_3, T_4 - the tag lengths are $11, 11, 15, 15$, and the table sizes are therefore $16KB, 16KB, 40KB, 40KB$
- Config 2: For tables T_1, T_2, T_3, T_4 - the tag lengths are $13, 13, 14, 14$, and the table sizes are therefore $18KB, 18KB, 38KB, 38KB$
- Config 3: For tables T_1, T_2, T_3, T_4 - the tag lengths are $17, 17, 12, 12$, and the table sizes are therefore $22KB, 22KB, 34KB, 34KB$
- Config 4: For tables T_1, T_2, T_3, T_4 - the tag lengths are $19, 19, 11, 11$, and the table sizes are therefore $24KB, 24KB, 32KB, 32KB$

The results for the aforementioned 4 configs are mentioned in Table 2. We observe that for both benchmarks, we do not see any noticeable difference in either the MPKI or the prediction accuracy for all the varying tag lengths (and, therefore, the varying table sizes). As per our observations, increasing the individual table sizes while keeping the overall predictor size the same does not offer any performance benefits.

Metrics	Config 1	Config 2	Config 3	Config 4
MPKI	0.8236	0.8236	0.8243	0.8233
Pred Acc %	98.99	98.99	98.99	98.99

(a) **644.nab_s**

Metrics	Config 1	Config 2	Config 3	Config 4
MPKI	3.249	3.245	3.263	3.243
Pred Acc %	97.63	97.64	97.62	97.64

(b) **648.exchange2_s**

Table 2: Performance Metrics with Varied Table Sizes (Q1 (b))

Q1 (c)

We know that using a single complex branch predictor might not always be great for reasons such as a longer warm-up times and more [5]. As mentioned in the aforementioned section, intelligently combining branch predictors can prove advantageous. In this section, we explore combining the *Perceptron* and TAGE branch predictors. We keep the hardware budget of the total hybrid predictor fixed at 128KB (size of combiner table not included) and study the effect of changing the allocated size to each predictor component. In particular, we study three configurations - 30:70, 50:50 and 70:30 splits of the hardware budget among *Perceptron* and TAGE predictors, respectively. We still use a 4-level TAGE predictor for the hybrid predictor. The specifications of the hybrid predictor are listed below -

- **30:70** - The *Perceptron* predictor is given a budget of 38.4KB (24-bit history length, 8-bit per weight, 1500 perceptrons and 100 200-bit update table entries), while the TAGE predictor is allocated a budget 89.6KB (128K-entry *bimodal* base table, 4K-entry 17-bit per entry T_0, T_1, and 8K-entry 20-bit per entry T_2, T_3).
- **50:50** - Both *Perceptron* and TAGE predictors are allocated 64KB hardware budget each. The *Perceptron* predictor comprises a 32-bit history, 8-bit per weight, 2000-entry perceptron table. There are 100 200-bit update table entries. The TAGE predictor is allocated a budget 64KB (128K-entry *bimodal* base table, 2K-entry 15-bit per entry T_0, T_1, 4K-entry 16-entry T_2, and 8K-entry 20-bit per entry T_3)
- **70:30** - The *Perceptron* predictor is comprised of 1860-perceptron table with each entry being 48 8-bit weights, while the TAGE predictor is made up of 64K *bimodal* base table, 1K-entry 16-bit per entry T_0, 2K-entry 16-bit per entry T_1 and 4K-entry 17-bit per entry T_2 and T_3.
- We use a *bimodal* table with 4096 entries (1KB) as the meta predictor for the proposed hybrid predictor. This is not included in the hybrid predictor hardware budget.

We tabulate the results for the three configurations over all 3 benchmark data sets in Table 3. Some key observations from these experiments are listed below -

- We notice that for the **644.nab_s** benchmark, there is no significant change in performance across configurations. However, the 70 : 30 seems to perform the best for this benchmark. This configuration uses a *Perceptron* predictor with 48 bits of history length while the maximum history length for stage component is 32. This could be one of the reasons for the slight increase in performance as the perceptron component size increases.

- For the **648.exchange2_s** benchmark, the 30 : 70 provides the best performance and the overall performance declines as the size of the TAGE predictor is reduced. Clearly, TAGE predictor increases the effectiveness for this branch predictor-benchmark pair more than the *Perceptron* predictor.
- As has been the trend, we do not see any significant change for the **654.roms_s** benchmark. As stated above, the reason could be a poor correlation between branches and fewer branches in general.

Benchmark	Config (P:T)	MPKI	Pred Acc	IPC
644.nab_s	**30:70**	0.856	98.95	2.57
	50:50	0.867	98.94	2.57
	70:30	0.831	98.98	2.58
648.exchange2_s	**30:70**	3.894	97.17	3.23
	50:50	5.109	96.28	3.05
	70:30	6.536	95.24	2.87
654.roms_s	**30:70**	0.061	99.88	0.55
	50:50	0.061	99.88	0.55
	70:30	0.061	99.88	0.55

Table 3: Benchmark Results for Hybrid Perceptron-TAGE Configs

Q1 (d)

We implement the remaining two possible combinations with the three mentioned branch predictors, with a fixed predictor size of 128KB. Branch predictors like perceptron and TAGE have been shown to suffer from longer warm-up times [6, 10], and this becomes a serious bottleneck when context switching is involved. Hence, we incorporate a simple *GShare* predictor with both these predictors to handle long warm-up times. We report the results for both these combinations in Tables 4a and 4b. We see that for both the branch predictors, the best performance is achieved as we increase the share of the *GShare* predictor in the hybrid predictor. This is most likely due to the shorter warm-up times of the *GShare* predictor. We show that the hybrid *GShare*-Perceptron branch predictor (with fixed budget of 128KB) performs the best out of all other implementation discussed above for the benchmark **644.nab_s**, although the performance improvement is marginal (99.4% as compared to 98.99% of the TAGE branch predictor of the same size.)

Benchmark	Config (G:P)	MPKI	Pred Acc	IPC
644.nab_s	**15:85**	0.539	99.40	2.85
	25:75	0.539	99.40	2.85
	50:50	0.539	99.40	2.85
648.exchange2_s	**15:85**	9.52	93.07	2.53
	25:75	9.396	93.16	2.54
	50:50	8.402	93.88	2.64

(a) Benchmark Results for Hybrid GShare-Perceptron Configs (128KB)

Benchmark	Config (G:T)	MPKI	Pred Acc	IPC
644.nab_s	**15:85**	0.57	99.36	2.84
	25:75	0.57	99.37	2.84
	50:50	0.57	99.37	2.84
648.exchange2_s	**15:85**	6.09	95.57	3.23
	25:75	5.29	96.15	3.05
	50:50	4.73	96.56	2.87

(b) Benchmark Results for Hybrid GShare-TAGE Configs (128KB)

Table 4: Benchmark Results for Explored Hybrid Strategies

QUESTION 2

CPI Stack Architecture

Cycles-Per-Instruction (CPI) stacks are an intuitive way to visualize and identify processor core performance bottlenecks. These bottlenecks, once identified, can guide the programmers to make better software choices. A CPI stack comprises a base CPI component (representing the cycles where useful work occurs) and additional *lost* cycle components stemming from events like cache misses, branch mispredictions, and so on, which consume cycles without contributing to productive work. Calculating CPI stacks on modern superscalar processors is a challenging task due to the presence of various overlapping effects. Because stall events can occur concurrently (e.g., an instruction cache miss and a data cache miss), these CPI stacks often do not paint the complete picture of the breakdown of the CPI A common and simple method for calculating the different components in a CPI stack involves multiplying the number of specific miss events by the average penalty associated with each of these events [1, 8]. This approach has multiple pitfalls, as pointed out by [3], including the fact that the average penalty for a given miss event may vary across programs and that certain penalties associated with miss events can be concealed or overlapped by employing out-of-order processing of independent instructions and handling miss events concurrently. This, however, adds another level of complexity to the construction of the CPI stack, as mentioned above. Moreover, the simplistic approach does not differentiate between miss events occurring along mispredicted control flow paths and those along correctly predicted control flow paths.

Counting Events

We use `perf`, a Linux-based profiling tool, to find the count of a number of pre-selected events in a specified interval of time. We use the `perf stat` command to capture the count of the number of hardware events that occur at a suitable interval while executing the four benchmark programs provided. We use the `stat` command provided by the `perf` utility to sample the events at a suitable time interval. These intervals are chosen to give us sufficient data points for regression analysis. We record 8 events using `perf stat` to construct the regression model for the CPI stack of our benchmark programs. These eight events are listed below. The data is collected by dumping the `perf` output to a file, which is then parsed, and data is extracted and stored in `pandas` data frames. The exact command used to collect data is mentioned below. We use the `taskset` command to set the CPU affinity so that the program runs on the specified single CPU core.

```
perf stat -I <interval size> -o <output file> -e cycles, instructions,
L1-dcache-load-misses, L1-icache-load-misses, dTLB-load-misses,
dTLB-store-misses, iTLB-load-misses, branch-misses, LLC-load-misses,
LLC-store-misses <benchmark executable>
```

Data Visualization We visualize the correlation between individual events and the target CPI by plotting the same for each of the given benchmark programs (Fig 3)

Linear Regression

We perform a multiple linear regression on the extracted data points in order to construct the CPI stack. We make use of the `sklearn` library in Python to perform the multiple linear regression. The steps followed are mentioned below -

- The events recorded for the linear regression model are - `L1-dcache-load-misses`, `L1-icache-load-misses`, `dTLB-load-misses`, `dTLB-store-misses`, `iTLB-load-misses`, `branch-misses`, `LLC-load-misses`, `LLC-store-misses`
- We start by normalizing our feature values (the independent variables) by the number of instructions for that sample. Thus, each of the features (miss event) has a unit of $\frac{event-counts}{instructions}$.
- We use the `Lasso` regression class for building our model. The Lasso Regression is a popular linear regression method with an additional L1 penalty on the loss function.
- We build a multiple linear regression model to fit the hyperplane equation mentioned below. The target variable y is CPI, while the independent variables are the miss event counts per instruction. The coefficients β_i, therefore assume the unit $\frac{cycles}{event-counts}$.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_8 x_8 \tag{1}$$

- It is important to note that we force the model to only look at positive coefficients minimizing the loss function since we want to look at the additive effects of the events on the CPI stack.
- The parsed data is split into a training and testing data set. We do so to verify how the estimated frame generalizes to new data points (test set).
- We report various performance metrics (detailed description in Table 5) in order to evaluate the efficacy of the model.
- We then construct the CPI stack for each benchmark data set. We do so by using the coefficients β_i received from the linear regression model. The x_i is calculated by taking the mean of the respective feature over the entire data set for a benchmark. Substituting these into Equation 1, we get the predicted average CPI for a benchmark data set.
- Each $\beta_i x_i$ term represents the CPI breakdown for the event x_i, while the β_0 represents the base CPI.

The following section shows the results achieved for all four benchmark data sets - 526.*blender_r*, 527.*cam4_r*, 531.*deepsjeng_r* and 541.*leela_r*.

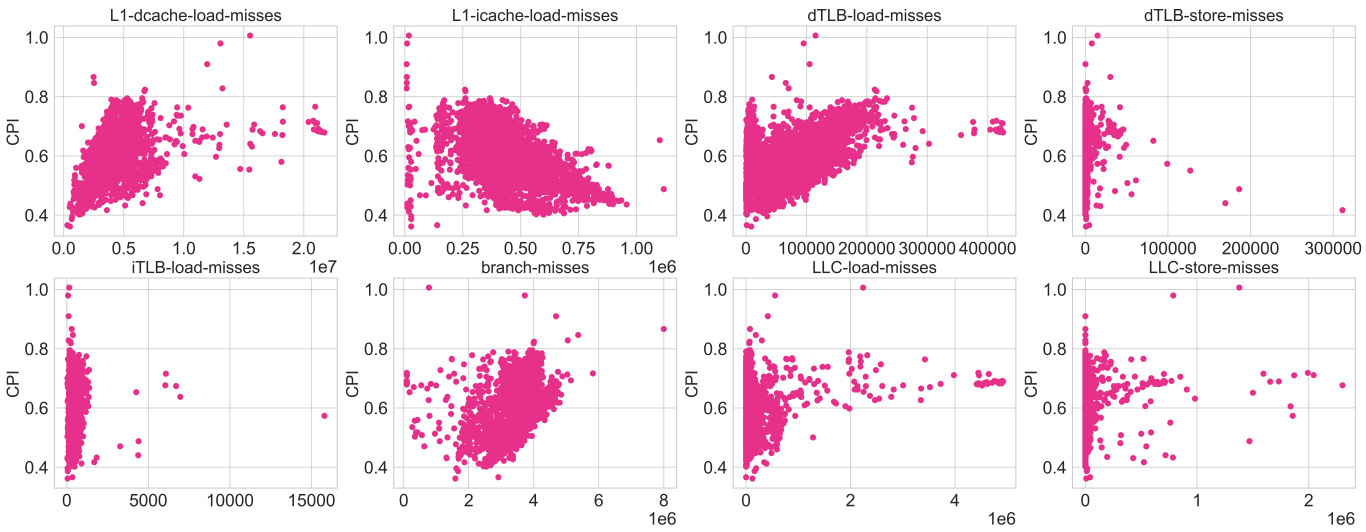


Figure 3: Correlation of Miss Events With the Target CPI for 526.blender_r Benchmark Set

Metric	Description
Residual	The difference between the actual value and the value predicted by the model for any given data point.
RMSE	The standard deviation of the residuals (prediction errors) for all associated data points.
R^2	It is the coefficient of determination (or the coefficient of multiple determination in this case) and is the ratio of the variance explained by the model to the total variance in the data.
Adjusted R^2	The R^2 value increases as the number of independent variable in the linear model increases. The adjusted R^2 value normalizes the R^2 to the number of independent variables.
F-statistic	It is defined as the ratio between the total mean square error and the residual mean squared error, normalized to the total mean square error and helps assess the significance of the entire regression model.
p-value	Identifies the significance of the regression coefficients.

Table 5: Description of Metrics to Evaluate Our Regression Model

Results

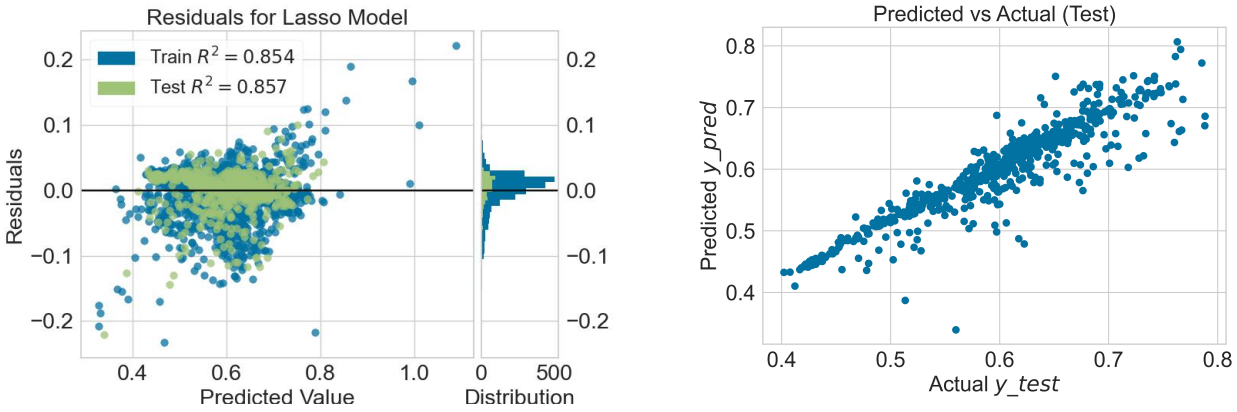
For each of the 4 SPEC2017 benchmark data sets provided, we present the linear regression model results and the resulting CPI stack for each benchmark set below. The complete data set for each benchmark is split 80-20 as training and testing data. The error metrics for the linear models trained for the 4 benchmarks are specified in Table 6.

526.blender_r

The `526.blender_r` benchmark completes its run in about 225 seconds, which allows us to sample the events at a large sampling time of 75 ms. This allows us to get about 3000 data points, which we split into training and testing data (80:20 split). The correlation of CPI with all the events recorded is plotted in Fig 3. Listed below are the key observations -

- We notice that the model fit to this benchmark data performs reasonably well (Table 6) with the RMSE around 0.033. This tells us that our model predictions are not far from the true values, and hence, the model is a good fit.
- The residue values in Fig 4 (a) are fairly clustered together, and their distribution varies between -0.1 and 0.1. A good regression model's residues follow a normal distribution and are clustered around 0. We see that this is clearly the case for our linear model.
- We can also deduce the same from the fact that both train and test R^2 are more than 85% (Fig 4 (a)). In other words, the model is able to explain 85% of the variance of the CPI can be explained by the chosen features.
- Fig 4 (b) also shows how the model generalizes to unseen data. We can observe that the actual and predicted CPI from the model on the test data set are highly correlated.

- The p-values for the 8 coefficients are - $[0, 0, 0, 1, 0, 0, 0, 1]$. These indicate that the dependent variables `dTLB-store-misses` and `LLC-store-misses` are statistically insignificant in calculating independent variable CPI for this particular benchmark.
- We can deduce from the p-values that the stall cycles induced by the events `dTLB-store-misses` and `LLC-store-misses`, are therefore being masked or overlapped and, therefore, their effective stall CPI contribution to the CPI stack is negligible.



(a) Residual Plot

(b) Test Set Performance

Figure 4: Residual Plot and Performance on Test Set (526.blender_r)

527.cam4_r

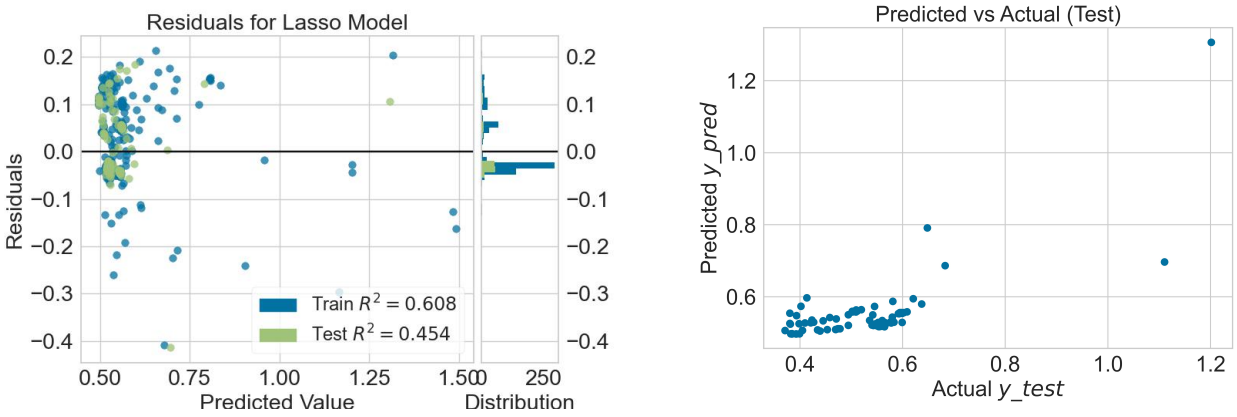
The `527.cam4_r` benchmark completes its execution in about 16 seconds. In order to sample enough data points to fit a regression model, the sampling time has to be fairly small (25 ms, in our case). This small sampling value for recording events using `perf stat` imposes a severe overhead on the program execution. We list down the key observations from the performance metrics -

- We observe that the fit value we receive for regression on this benchmark is not as great as the others. This is most likely due to the overhead imposed by the frequent `perf stat` calls. A larger sampling period, on the other hand, would restrict the number of data samples, in turn, again leading to a badly fit model.
- Another reason for a badly fit model could be the correlation of CPI to the events we have chosen as features. We observe the CPI. The p-values we receive for the model for this benchmark are indicative of this hypothesis -$[1, 0, 0.05, 0, 0.25, 0.39, 1, 0.1]$

Linear Regression Error Metrics				
Benchmark / Metrics	**526.blender_r**	**527.cam4_r**	**531.deepsjeng_r**	**541.leela_r**
RMSE	0.033	0.066	0.005	0.004
R^2	0.854	0.608	0.994	0.966
Adj R^2	0.853	0.604	0.994	0.968
F-statistic	4.19M	0.11M	3.99M	1.75M

Table 6: Linear Regression Model Performance Metrics

- Fig 5 shows us the spread of the residue values is quite large. However, the residues do still follow a somewhat normal distribution.
- The p-values indicate that the first and the seventh events (`L1-dcache-load-misses` and `LLC-load-misses`) do not contribute to the CPI stack. We can therefore infer that their stall cycles are being masked due to the overlapped processing of independent instructions and miss events.



(a) Residual Plot

(b) Test Set Performance

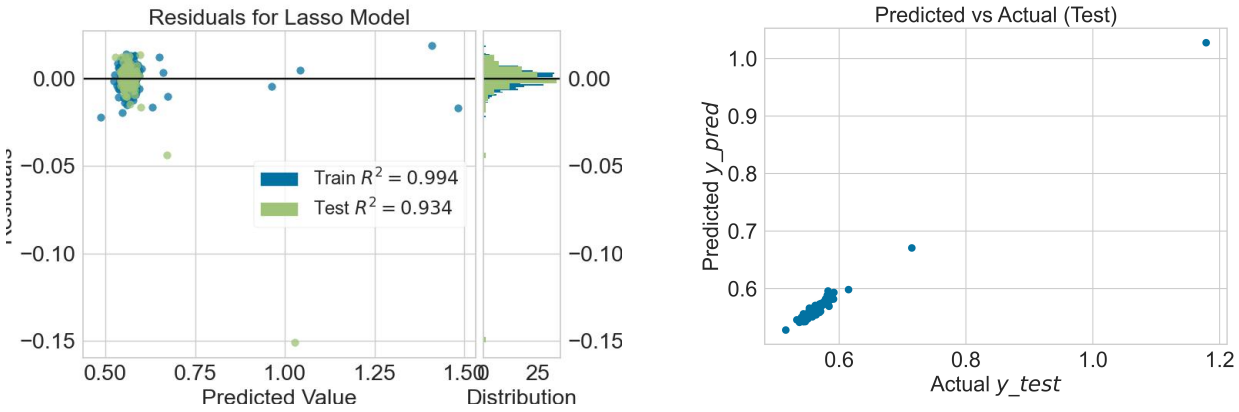
Figure 5: Residual Plot and Performance on Test Set (527.cam4_r)

531.deepsjeng_r

The `531.deepsjeng_r` finishes execution in about 46 seconds, giving us enough time to perform sampling of events at a large enough period to avoid `perf stat overheads`. We decided on 75 ms as an apt sampling period for this benchmark. The data obtained is again split 80:20 into training and testing sets. Listed below are the key observations for this benchmark -

- The observed p-values for the linear model for this benchmark are - $[0, 0.92, 0, 0, 0, 0, 0, 0]$. It is evident that most features (except the second feature `L1-icache-load-misses`) used for the regression analysis are significant towards the target CPI.
- All features being significant towards the target CPI points to the fact that this benchmark could do better if software changes to overlap the effect of stall cycles were made.
- The fit of the model is quite good. This is evident from the low RMSE and high R^2 and adjusted R^2 values.
- It is also clear from Fig 6 (a) that the residues are tightly scattered around 0 and follow a normal distribution, which implies to model was well fit to the data.

We observe that the model fit for this benchmark performs the best out of the 4. The residue values in Fig 6 are very clustered, and the R^2 and adjusted R^2 values are very high.



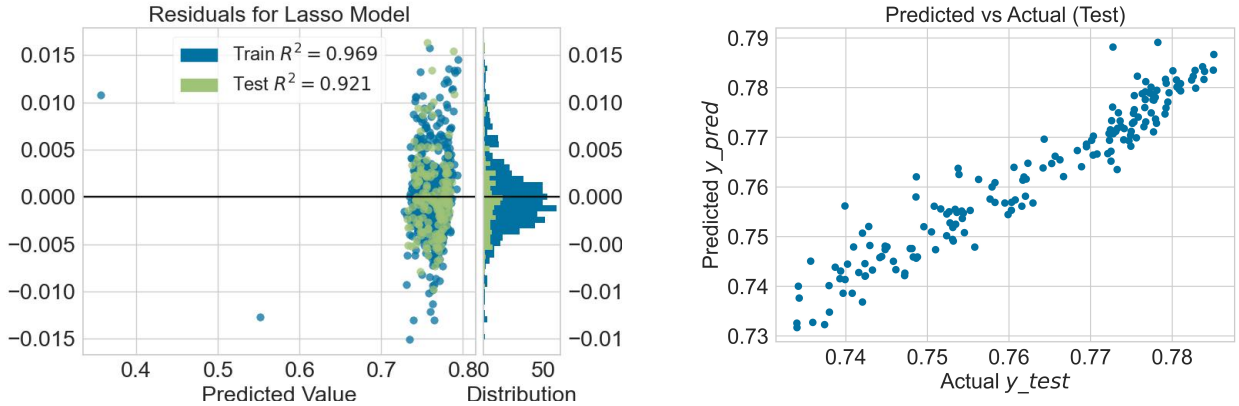
(a) Residual Plot (b) Test Set Performance

Figure 6: Residual Plot and Performance on Test Set (531.deepsjeng_r)

541.leela_r

`541.leela_r` benchmark finishes execution in about 63 seconds. We again see a very good model fit for this benchmark. The sampling time chosen for this benchmark is 75 ms. The data set is split in a manner similar to that of the other benchmarks. All the performance metrics are listed in Table 6 We list down the key observations from the performance metrics -

- High R^2 and adjusted R^2 values (Table 6) and a very small scatter (between -0.015 and 0.015) in the residue plot (Fig 7 (a)) indicate a good model fit. The residue scatter also follows a normal distribution, as the figure shows.
- We observe (Fig 7 (b)) that the predicted and the true values on the test data set are also highly correlated, indicating a good model fit.
- The p-values obtained for this benchmark are - $[1, 1, 1, 0.57, 0, 0, 0, 1]$. This indicates that the independent variables `L1-dcache-load-misses, L1-icache-load-misses, dTLB-load-misses, LLC-store-misses` are not statistically significant while calculating the target CPI.



(a) Residual Plot (b) Test Set Performance

Figure 7: Residual Plot and Performance on Test Set (541.leela_r)

CPI Stack & Analysis

A CPI stack offers valuable insights into how an application behaves on a specific microprocessor. In this section, we construct the CPI stack for the 4 benchmarks provided to us. The coefficients of the linear regression model for all 4 benchmark sets were restricted to be positive since we only want the additive effects of the stall events considered. Fig. 8 shows the CPI stack of the 4 benchmarks. Listed below are the observations for CPI stacks for each of the 4 benchmark sets provided -

- **526.blender_r** It is evident that the base CPI is almost 50% and `branch-misses` are the major stall cycles inducing event for this benchmark. It can be deduced, therefore, that the branch predictor being used for running the benchmark does not perform very accurately for this program. the other major component contributing to the misses is `L1-dcache-load-misses`. This indicates that there is scope for the application developer to make use of data locality so as to further optimize this benchmark application.
- **527.cam4_r** The base CPI for this benchmark forms about 90%, which shows that stalls from other events are overlapped and only contribute to about 10% of the overall CPI.
- **531.deepsjeng_r** The base CPI for this benchmark is also quite high. The highest contributor to miss cycles again is the `branch-misses` which can be taken care of by using a better branch predictor for the processor.
- **541.leela_r** `branch misses` accounts for almost half of the CPI. The branch predictor of the processor is highly ineffective in predicting the branches.

The total CPI for all 4 benchmarks is less than 1, as it should be since the benchmarks are run on a super-scalar processor. The best CPI is achieved by the 527.*cam4_r* benchmark, which is able to execute about 1.6 instructions per second.

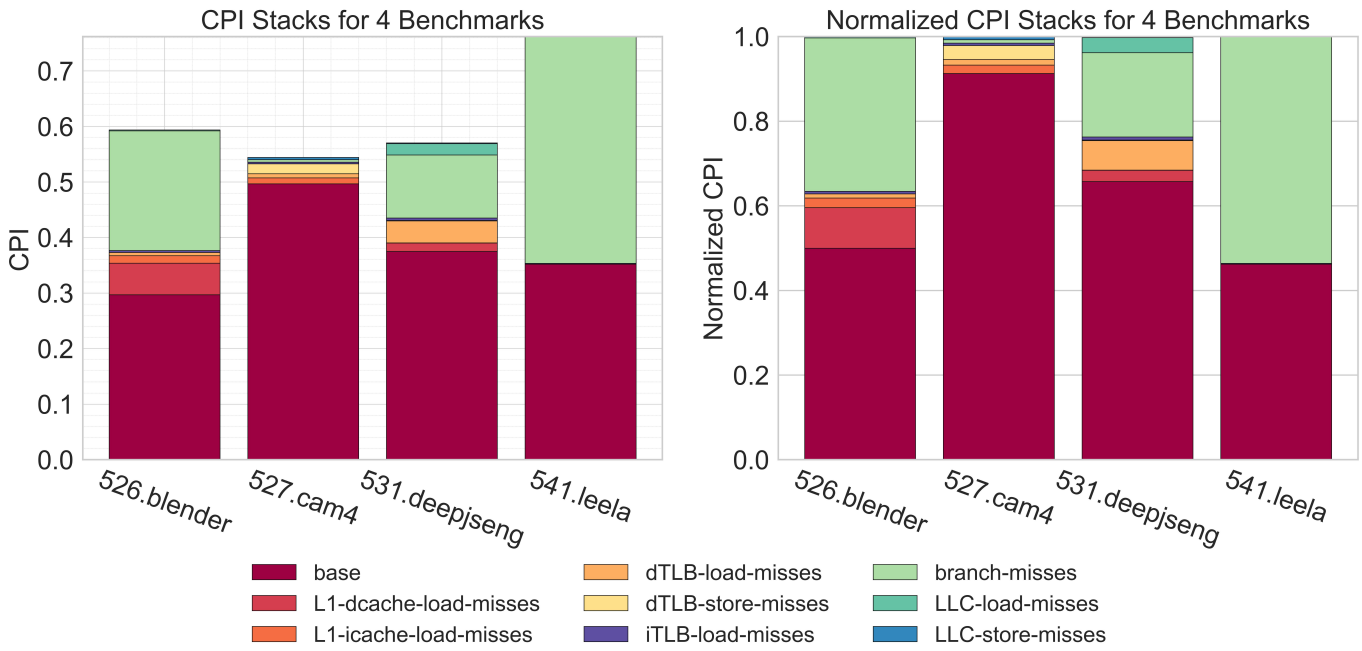


Figure 8: CPI and Normalized CPI Stacks For 4 Benchmarks

Arch	x86_64
Model	Intel(R) Core(TM) i5-10500 CPU
CPU MHz	3100
L1d cache	192 KB
L1i cache	192 KB
L2 cache	1.5 MB
L3 cache	12 MB
Address sizes	39 bits physical, 48 bits virtual

Table 7: CPU Specifications

References

[1] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., p. 266–277.

[2] Bucek, J., Lange, K.-D., and v. Kistowski, J. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (2018), pp. 41–42.

[3] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A top-down approach to architecting cpi component performance counters. *IEEE Micro 27*, 1 (2007), 84–93.

[4] Gober, N., Chacon, G., Wang, L., Gratz, P. V., Jiménez, D. A., Teran, E., Pugsley, S. H., and Kim, J. The championship simulator: Architectural simulation for education and competition. *ArXiv abs/2210.14324* (2022).

[5] Jiménez, D. A. Multiperspective perceptron predictor with tage.

[6] Jiménez, D. A., and Lin, C. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture* (2001), IEEE, pp. 197–206.

[7] KanPard005. RISCY_V_TAGE. https://github.com/KanPard005/RISCY_V_TAGE.

[8] Keeton, K., Patterson, D., He, Y. Q., Raphael, R., and Baker, W. Performance characterization of a quad pentium pro smp using oltp workloads. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)* (1998), pp. 15–26.

[9] McFarling, S. Combining branch predictors. Tech. rep., Citeseer, 1993.

[10] Seznec, A., and Michaud, P. A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism 8* (2006).